ZEUS

Utilities Manual

Zilog

An affiliate of EXXON Corporation

03-3196-01

April 1982

Copyright 1981 by Zilog, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Zilog.

The information in this publication is subject to change without notice.

Zilog assumes no responsibility for the use of any circuitry other than circuitry embodied in a Zilog product. No other circuit patent licenses are implied.

Portions of the material have been reproduced with the permission of Western Electric Company, Incorporated.

SADIE Diagnostic Tape
14-0009-03

The following items apply to Version 3.0 of SADIE release on cartridge tape with part number 14-0009-03. Please report any additional problems to Zilog immediately by recording them on your machine with the STR command. The listing resulting from the STRPRINT command can then be sent directly to Zilog.

DATA PRODUCTS INTERFACE TEST ERROR : DR.PRT, the Data Products Interface test does not work correctly. The routine times out while waiting for the printer to come on line. This problem will be fixed in the next release of SADIE.

MEMORY TESTS DO NOT CHECK PARITY : NEWMEM1, NEWMEM2, and NEWMEM3 tests do not detect parity errors. They detect only data-line and address-line errors. This problem will be corrected in the next release of SADIE.

ZEUS Software Release, 1.7
NOTE TO USER

P/N 03-0200-01

The following modes need to be corrected to accurately run ZEUS:

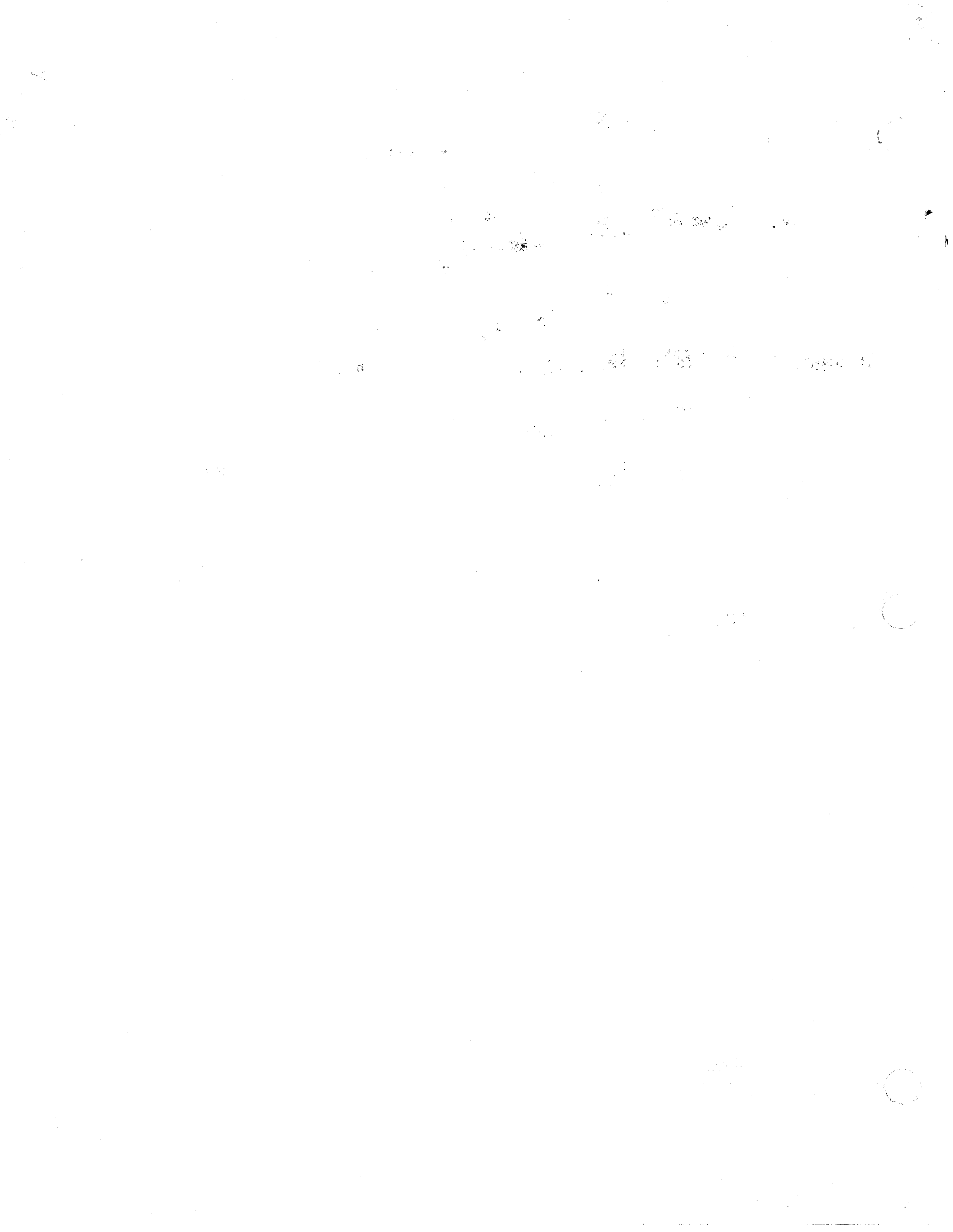
1. /zeus*; permission modes set as 751 needs to be changed to 644.

2. /usr/lib/tmac/tmac.an; permission mode set at 640 needs to be changed to 644.

These changes can be accomplished easily by issuing the following commands as super user:

```
cd /
chmod 0644 zeus*
upkeep -d (if there appears to be a difference at
           this point, you may need to re-initialize
           your contents file by "upkeep -i".  If
           you do not receive a difference then you
           may proceed to the next command.)

cd /usr/lib/tmac
chmod 0644 tmac.an
upkeep -d (same as above)
```



E3-0152-03, Errata, ZEUS Operating System, Version 1.7

The following items apply to Version 1.7 of ZEUS releases on cartridge tape with part number 14-0006-03. Please report any additional problems to Zilog immediately by recording them on your machine with the STR command. The listing resulting from the STRPRINT command can then be sent directly to Zilog.

1. The cartridge tape unit in your System 8000 is a very high recording density unit (6400 bits per inch). Zilog subjects cartridge tapes to additional screening before making them acceptable for shipment. Zilog recommends that users buy cartridge tapes for their systems directly from Zilog or contact the major cartridge tape vendors directly for tapes screened for 6400 bits per inch. Customers may find that they have tape reliability problems if they purchase standard tapes from distributors as these tapes are intended for 1600 bits per inch use. It is expected that within one year standard tapes certified for 6400 bits per inch will be available directly from the distributor.

2. The following programs normally found in Version 7 releases are not currently in ZEUS:

   cu   iostat   refer

3. The uucp program erroneously sets the tty device (/dev/tty3 by default) to mode 0600. This should be reset to mode 0666 for use by remote.

4. When using sysgen to generate new operating systems the /swap file system's size must remain 3200.

5. The following problems may occur in the ZEUS 1.7 Tape Release:

C. C Compiler

If two external names are identical in the first seven characters the two variables are mapped into one memory location. For example,

```
int     g123456x;
int     g123456y;
main()
{
        g1234567x = 1;
}
```

the two global variables are mapped into one location named _g123456. The C compiler truncates names after the first eight characters. The only way around this problem now is not to have external names that are identical in the first seven characters.

D. Secondary Bootstrapper Anomaly

Occassionally the secondary bootstrapper prompts with a message "no more file slots". This seems to occur after a large number of programs have been executed. This has not occured when following the steps given in the System Administrator Manual. If this should occur, you should reboot the system from tape.

NOTICE TO OWNER

FEDERAL COMMUNICATIONS COMMISSION RADIO FREQUENCY INTERFERENCE STATEMENT

Warning: This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. As temporarily permitted by regulation it has not been tested for compliance with the limits for Class A computing devices pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference. Operation of this equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

ZEUS UTILITIES

03-3196-01

PRELIMINARY VERSION

The information contained in this draft may undergo changes, both in content and organization, before arriving at its final form.

PREFACE

The ZEUS™ Utilities Manual documents, in handbook and tutorial form, important ZEUS features and complex programs that run under ZEUS. System 8000™ hardware and ZEUS software are used in the Zilog System 8000. This manual and the related manuals below provide the complete technical documentation of the System 8000 and ZEUS.

Title	Zilog Number
System 8000 Hardware Reference Manual	03-3198
System 8000 User Manual	03-3199
ZEUS System Administrator Manual	03-3197
ZEUS Reference Manual	03-3195

ZEUS™ and System 8000™ are registered trademarks of Zilog, Inc.

TABLE OF CONTENTS

Introduction to Zeus Utilities	INTRODUCTION
A Tutorial Introduction to ADB	ADB
ZEUS PLZ/ASM Assembler User Guide	AS
Awk: A Pattern Scanning and Processing Language	AWK
The C Programming Language	C
ZEUS Communication Package	COMM
An Introduction to the C Shell	CSH
The ZEUS Line-Oriented Text Editor, ed	ED
File System Integrity	FSCK
Learn	LEARN
Lex: A Lexical Analyzer Generator	LEX
Lint: A C Program Checker	LINT
Make	MAKE
Typing Documents on the ZEUS System ...	MS
Nroff/Troff Reference Manual	NROFF/TROFF
Zeus Programming	PGMG
S8000 PLZ/SYS User Guide	PLZ/SYS
SED: A Noninteractive Text Editor	SED
The ZEUS Shell	SHELL
A Troff Tutorial	TROFF
UUCP Installation	UUCP
Introduction to Display Editing with vi	VI
YACC: Yet Another Compiler-Compiler ..	YACC
ZEUS for Beginners	ZEUS

File System Integrity

File System Integrity .... FSCK

INTRODUCTION TO ZEUS UTILITIES

This volume contains manuals and tutorial describing the basic utility programs of ZEUS.

ZEUS for Beginners describes the basics of logging in, running programs, creating and modifying files, etc.

Learn is an teaching-machine program for practice in using ZEUS.

The ZEUS mechanism for running programs is itself a user program called a shell. Commonly used under ZEUS is csh, described in An Introduction to the C Shell. An alternative is sh (known simply as "The Shell,"); it is described in The ZEUS Shell.

There are two utilities for the maintaining of text files. They are the command-line oriented editor ed, and the screen oriented editor vi and are described in The ZEUS Line-oriented Text Editor, ed, and Introduction to Display Editing with vi.

Troff is a macro-oriented typesetting program; nroff approximates troff on typewriter-like devices. The Nroff/Troff Reference Manual describes these programs. They are used with a package of commands (macros); Typing Documents on the ZEUS System Using the -ms Macros with Troff and Nroff is a first-time document that describes a simple macro package. A Troff Tutorial describes problems of typesetting documents.

SED: A Noninteractive Text Editor describes a program which edits input of indefinite length; commands are similar to those of ed.

Awk: A Pattern Scanning and Processing Language describes a stream editor with a powerful command language.

The primary programming language on ZEUS is C. Special considerations of programming in C on ZEUS are listed in The C Programming Language. Lint: A C Program Checker detects implementation-dependent code and other bad features.

PLZ/SYS is another high-level ZEUS language; PLZ/ASM is the ZEUS resident assembler. They can be used together to design low-level programs.

ZEUS Programming explains how programs running under ZEUS interact with ZEUS; it describes how ZEUS programs handle

command arguments, input/output, etc.

A Tutorial Introduction to ADB describes a program which is used to examine core files resulting from aborted programs, patch object files, and run programs with embedded breakpoints.

Lex: A Lexical Analyzer Generator and _YACC: Yet Another Compiler-Compiler_ describe tools useful in developing programs which apply translation rules to input.

Make describes a program used to maintain a large group of interrelated files, such as the source code files and their associated object files that are behind a large C program.

ZEUS Communication Package describes a communications path between ZEUS and remote systems.

UUCP Installation describes a program that links to other ZEUS systems (or any other system that can run UUCP) via tty port-to-port connections or transient telephone connections.

File System Integrity Program (FSCK) Reference Manual describes how file systems can be protected against corruption upon reboot.

A Tutorial Introduction to ADB*

* This information is based on an article originally written by J.F. Maranzano and S.R. Bourne, Bell Laboratories.

PREFACE

This document contains information on ADB (A DeBugger), a new debugging program. With ADB, it is possible to examine core files resulting from aborted programs, print variable contents in a variety of formats, patch files, and run programs with embedded breakpoints.

This document is written as a tutorial. It is assumed that the reader is familiar with the C language.

The examples referenced in the text are located in Appendix A. For ease of reference, it is recommended that the examples be brought up on the terminal while the text is read from the hard copy.

TABLE OF CONTENTS

SECTION 1 A QUICK SURVEY 4

1.1 Basic Command Format 4
1.2 File Locations 4
1.3 Current Address 4
1.4 Formats 5
1.5 General Requests 6

SECTION 2 DEBUGGING C PROGRAMS 7

2.1 Debugging a Core Image 7
2.2 Calling Multiple Functions 8
2.3 Setting Basic Breakpoints 9
2.4 Setting Advanced Breakpoints 11
2.5 Using Other Breakpoint Facilities 14

SECTION 3 MAPS 15

SECTION 4 ADVANCED USAGE 17

4.1 General 17
4.2 Formatted Dump 17
4.3 Directory Dump 19
4.4 Ilist Dump 19
4.5 Value Conversion 19

SECTION 5 PATCHING 21

SECTION 6 CAUTIONS 23

APPENDIX A PROGRAM EXAMPLES 24

APPENDIX B ADB SUMMARY 41

SECTION 1

A QUICK SURVEY

1.1 Basic Command Format

The ADB command copies core to an output file. The command format is:

```
adb objfile corefile
```

where _objfile_ is an executable ZEUS file (default is _a.out_) and _corefile_ (default is _core_) is a core image file. When the defaults are used, the command appears as:

```
adb
```

The file name minus (-) means ignore an argument, as in:

```
adb - core
```

1.2 File Locations

ADB has requests for examining locations in the contents of _objfile_, (the ? request) or the _corefile_ (the / request). The general form of these requests is:

```
address ? format
```

or

```
address / format
```

where _format_ describes the printout (Section 2.4).

1.3 Current Address

ADB maintains a current address, called dot, similar in function to the current pointer in the ZEUS editor. The request:

```
.,10/d
```

prints ten decimal numbers starting at dot. Dot then refers to the address of the last item printed.

When an address is entered, the current address is set to that location, so that:

```
0126?i
```

sets dot to octal 126 and prints the instruction at that address.

When used with the ? or / requests, the current address can be advanced by typing a new line, and it can be decremented by typing ^.

Addresses are represented by expressions of decimal, octal, and hexadecimal integers, and symbols from the program under test. These can be combined with the operators +, -, *, % (integer division), & (bit and), | (bit inclusive or), # (round up to the next multiple), and ~ (not). All arithmetic within ADB is 32 bits. When typing a symbolic address for a C program, type _name_ or __name_; ADB recognizes both forms.

1.4 Formats

To print data, specify a collection of letters and characters that describe the format of the printout. Typing a request without a format causes the new printout to appear in the previous format. The following are the most commonly used format letters:

b	one byte in octal
c	one byte as a character
o	one word in octal
d	one word in decimal
f	two words in floating point
i	Z8000 instruction
s	a null terminated character string
a	the value of dot
u	one word as unsigned integer
n	print a new line
r	print a blank space
^	backup dot

Format letters are also available for long values (for example, D for long decimal and F for double floating point).

1.5 General Requests

Requests of the form

address,count command modifier

set dot to address and execute the command count times.

The following table gives general ADB command meanings:

Command	Meaning
?	Print contents from a.out file
/	Print contents from core file
=	Print value of "dot"
:	Breakpoint control
$	Miscellaneous requests
;	Request separator
!	Escape to shell

Use the request $q or $Q (or control-D) to exit from ADB.

SECTION 2

DEBUGGING C PROGRAMS

2.1 Debugging a Core Image

Example 1 (Appendix A) changes the string pointed to by _charp_, then writes the character string to the file indicated by argument 1. The common error shown is that a null character ends a character string. In the loop to print the characters, the ending condition is based on the value of the pointer _charp_, not the character that _charp_ points to. Executing the program produces a core file because of an out-of-bounds memory reference.

The following explanation refers to Example 2.

ADB is invoked by the command:

```
adb a.out core
```

The first debugging request:

```
$c
```

is used to give a C backtrace through the subroutines called.

The next request

```
$C
```

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in octal.

The next request

```
$r
```

prints the registers, including the program counter and an interpretation of the instruction at that location.

The request

```
$e
```

prints out the values of all external variables.

The request

```
$m
```

produces a report of the contents of the maps. A map exists for each file handled by ADB. The map for the _a.out_ file is referenced by ?, and the map for the _core_ file is referenced by /. Use ? for instructions and / for data when looking at programs.

To see the contents of the string pointed to by _charp_, enter

```
*charp/s
```

This uses _charp_ as a pointer in the _core_ file and prints the information as a character string. This printout shows that the pointer to the character buffer points to an address outside of the program's memory.

The request

```
.=o
```

prints the current address, not its contents, in octal. This has been set to the address of the first argument. The current address, dot, is used by ADB to keep the current location. It allows reference to locations relative to the current address; for example,

```
.-10/d
```

2.2 Calling Multiple Functions

The C program shown in Example 3 calls functions _f_, _g_, and _h_ until the stack is exhausted and a core image is produced. The following explanation refers to Example 4.

Enter the debugger with the command

```
adb
```

which assumes the names _a.out_ and _core_ for the executable file and core image file respectively. The request

```
$c
```

fills a page of backtrace references to _f_, _g_, and _h_. Entering _DEL_ terminates the output and returns to ADB request level.

The request

```
,5$C
```

prints the five most recently called procedures.

Each function (f,g,h) has a counter of the number of times it was called. The request

```
fcnt/d
```

prints the decimal value of the counter for the function f.

To print the the decimal value of x in the last call of the function h, type

```
h.x/d
```

It is not currently possible to print the value of local variables.

2.3 Setting Basic Breakpoints

The C program in Example 5 changes tabs into blanks (adapted from Software Tools by Kernighan and Plauger, pp. 18-27).

Run this program under the control of ADB (Example 6) by

```
adb a.out -
```

Set breakpoints in the program as:

```
address:b  [request]
```

The requests

```
settab:b
open:b
read:b
tabpos:b
```

set breakpoints at the start of these functions.

To print the location of breakpoints, enter

```
$b
```

The display indicates a count field. A breakpoint is bypassed count -1 times before causing a stop. The command field indicates the ADB requests to be executed each time

the breakpoint is encountered. In the example, no *command* fields are present.

Displaying the original instructions at the function *settab* sets the breakpoint to the entry point of the *settab* routine. Display the instructions using the ADB request

```
settab,5?ia
```

This request displays five instructions starting at *settab* with the addresses of each location displayed. Another variation is

```
settab,5?i
```

which displays the instructions with only the starting address.

The addresses are accessed from the *a.out* file with the ? command. When asking for a printout of multiple items, ADB advances the current address the number of bytes necessary to satisfy the request. In Example 6, five instructions are displayed and the current address is advanced 18 (decimal) bytes.

To run the program, enter

```
:r
```

To delete a breakpoint, for instance the entry to the function *settab*, enter:

```
settab:d
```

To continue execution of the program from the breakpoint, enter

```
:c
```

Once the program has stopped (in this case at the breakpoint for *open*), ADB requests can be used to display the contents of memory. For example, use

```
$C
```

to display a stack trace, or

```
tabs/8x
```

to print three lines of 80 locations each from the array called *tabs*. At location *open* in the C program, *settab* has been called to set a one in every eighth location of *tabs*.

Printing the _tabs_ array allows verification of _settab_.

2.4 Setting Advanced Breakpoints

Continue execution of the program (Example 6) with

```
:c
```

Read is called three times and the contents of the array _tabs_ is displayed each time. The single character on the left edge is the output from the C program.

Contine the program with the command

```
:c
```

The program hits the first breakpoint at _tabpos_ because there is a tab following the "This" word of the data.

Several breakpoints of _tabpos_ occur until the program changes the tab into equivalent blanks. Remove the breakpoint at that location by enteringg

```
tabpos:d
```

If the program is continued with

```
:c
```

it resumes normal execution after ADB prints the message

```
a.out:running
```

The ZEUS quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs, the program being debugged is stopped and control is returned to ADB. To save the signal and pass it to the test program, enter

```
:c
```

This can be useful when testing interrupt handling routines. Enter

```
:c  0
```

if the signal is not to be passed to the test program.

Now reset the breakpoint at _settab_ and display the instructions located there when the breakpoint is reached. This is accomplished by:

```
settab:b  settab,5?ia  *
```

* Owing to a bug in early versions of ADB (including the version distributed in Generic 3 ZEUS), these statements must be written as:

```
settab:b       settab,5?ia;0
read,3:b       main.c?C;0
settab:b       settab,5?ia;0
```

The ;0 sets dot to zero and stop at the breakpoint. To request each occurrence of the breakpoint and stop after the third occurrence, type:

```
read,3:b  tabs/8x
```

This request prints the local variable _c_ in the function _main_ at each occurrence of the breakpoint. The semicolon separates multiple ADB requests on a single line.

NOTE

Setting a breakpoint causes the value of dot to be changed. Executing the program under ADB does not change dot. For example, the commands

```
settab:b  .,5?ia
open:b
```

print the last value dot was set to (example _open_) _not_ the current location (example settab) at which the program is executing.

A breakpoint can be overwritten without first deleting the old breakpoint. Enter

```
settab:b  settab,5?ia;  *
```

The display of breakpoints

```
$b
```

shows the above request for the _settab_ breakpoint. When the breakpoint at _settab_ is encountered, the ADB requests are executed. The location at _settab_ has been changed to plant the breakpoint. All the other locations match their original value.

The execution of each function (_f_, _g_, and _h_ in Example 3) can be monitored by planting nonstop breakpoints. Call ADB

with the executable program of Example 3 as follows:

```
adb ex3 -
```

Enter the following breakpoints:

```
h:b  hcnt/d;  h.hi/;  h.hr/
g:b  gcnt/d;  g.gi/;  g.gr/
f:b  fcnt/d;  f.fi/;  f.fr/
:r
```

Each request line indicates that the variables are printed in decimal (by the specification d). The format is not changed and the d can be left off all but the first request.

The output in Example 7 illustrates two points. First, the ADB requests in the breakpoint line are not examined until the program under test is run. This means any errors in those ADB requests are not detected until run time. At the location of the error, ADB stops the program.

Example 7 also illustrates the way ADB handles register variables. ADB uses the symbol table to address variables. Register variables, like _f.fr_ in the previous example, have pointers to uninitialized places on the stack and print the message "symbol not found."

Another way of getting at the data in this example is to print the variables used in the call as with

```
f:b  fcnt/d;  f.a/;  f.b/;  f.fi/
g:b  gcnt/d;  g.p/;  g.q/;  g.gi/
:c
```

The operator / was used instead of ? to read values from the _core_ file. The output for each function, as shown in Example 7, has the same format. For the function _f_, for example, it shows the name and value of the _external_ variable _fcnt_. It also shows the address on the stack and value of the variables _a_, _b_, and _fi_.

The addresses on the stack continue to decrease until no address space is left for program execution. At this time the program under test aborts. A display with names is produced by requests

```
f:b  fcnt/d;  f.a/"a="d;  f.b/"b="d;  f.fi/"fi="d
```

In this format, the quoted string is printed literally and the d produces a decimal display of the variables. The results are shown in Example 7.

2.5 Using Other Breakpoint Facilities

Arguments and change of standard input and output are passed to a program as

```
:r  arg1  arg2 ... <infile  >outfile
```

This request aborts any existing program under test and restarts _a.out_.

The program being debugged can be single-stepped by

```
:s
```

If necessary, this request starts the program being debugged and stops after executing the first instruction.

ADB allows a program to be entered at a specific address by entering

```
address:r
```

The count field is used to skip the first _n_ breakpoints as

```
,n:r
```

The request

```
,n:c
```

is also used for skipping the first _n_ breakpoints when continuing a program.

A program is continued at an address different from the breakpoint by

```
address:c
```

The program being debugged runs as a separate process and is aborted by

```
:k
```

SECTION 3

MAPS

ZEUS supports several executable file formats that tell the loader how to load the program file. File type E707 is the most common and is generated by a C compiler invocation such as cc pgm.c. An E711 file is produced by a C compiler command of the form cc -i pgm.c. ADB interprets these different file formats and provides access to the different segments through a set of maps (see Example 8).

To print the maps, enter

    $m

In E707 files, both instructions and data (I & D) are intermixed. This makes it impossible for ADB to differentiate data from instructions, and some of the printed symbolic addresses look incorrect (for example, printing data addresses as offsets from routines).

In E711 files with separated I & D space, the instructions and data are also separated. However, in this case, since data is mapped through a separate set of segmentation registers, the base of the data segment is also relative to address zero. In this case, since the addresses overlap, it is necessary to use the ?* operator to access the data space of the _a.out_ file.

Example 9 shows the display of two maps for the same program linked as an E707 file and an E711 file respectively. The b, e, and f fields are used by ADB to map addresses into file addresses. The f1 field is the length of the header at the beginning of the file (020 bytes for an _a.out_ file and 02000 bytes for a _core_ file). The f2 field is the displacement from the beginning of the file to the data. For an E707 file with mixed text and data, this is the same as the length of the header; for an E711 files, this is the length of the header plus the size of the text portion.

The b and e fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either _a.out_ or _core_) is calculated as:

    b1≤A≤e1 => file address = (A-b1)+f1
    b2≤A≤e2 => file address = (A-b2)+f2

Locations can be accessed by using the ADB defined variables. The $v request prints the following variables

initialized by ADB:

```
b    base address of data segment
d    length of the data segment
s    length of the stack
t    length of the text
m    execution type (E707 and E711)
```

In Example 9 those variables not present are zero. These variables can be used by expressions such as

```
<b
```

in the address field. Similarly, the value of the variable can be changed by an assignment request such a

```
02000>b
```

which sets b to octal 2000. These variables are useful to know if the file under examination is an executable or core image file.

ADB reads the header of the core image file to find the values for these variables. If the second file specified is not a core file, or if it is missing, the header of the executable file is used.

SECTION 4

ADVANCED USAGE

4.1 General

It is possible with ADB to combine formatting requests to provide elaborate displays. Several examples follow.

4.2 Formatted Dump

To print four octal words followed by their ASCII interpretation from the data space of the core image file, enter

```
<b,-1/4o4^8Cn
```

The various request pieces mean:

<b	The base address of the data segment.
<b,-1	Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition, such as end of file, is detected.
4o	Print four octal locations.
4^	Back up the current address four locations (to the original start of the field).
8C	Print eight consecutive characters using an escape convention. Each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.
n	Print a new line.

The request:

```
<b,<d/4o4^8Cn
```

allows the printing to stop at the end of the data segment. The <d provides the data segment size in bytes.

The formatting requests can be combined with the ADB ability to read in a script to produce a core image dump script. Invoke ADB as:

```
adb a.out core < dump
```

to read in a script file, dump, of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8ona
```

The request 120$w sets the width of the output to 120 characters (normally, the width is 80 characters). ADB prints addresses as symbol + offset.

The request 4095$s increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095.

The request = can be used to print literal strings. Headings are provided in this dump program with requests of the form

```
=3n"C Stack Backtrace"
```

which spaces three lines and prints the literal string.

The request $v prints all nonzero ADB variables (Example 8). The request 0$s sets the maximum offset for symbol matches to zero, thus suppressing the printing of symbolic labels in favor of octal values. This is only done for the printing of the data segment. The request

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Example 11 shows the results of some formatting requests on the C program of Example 10.

4.3 Directory Dump

Example 12 dumps the contents of a directory made up of an integer _inumber_ followed by a 14-character name

```
adb dir -
=n8t"Inum"8t"Name"
0,-1? u8t14cn
```

In this example, the u prints the _inumber_ as an unsigned decimal integer, the 8t means that ADB spaces to the next multiple of 8 on the output line, and the 14c prints the 14-character file name.

4.4 Ilist Dump

The contents of the _ilist_ of a file system, such as /dev/src, is dumped with the following set of requests:

```
adb /dev/src -
02000>b
?m <b
<b,-1?"flags"8ton"links,uid,gid"8t3dn",
  size"8tDn"addr"8t20un"times"8t2YnY2na
```

In this example, the value of the base for the map was changed to 02000 (by saying ?m<b) because that is the start of an _ilist_ within a file system. The last access time, last modify time, and creation time are printed with the 2YnY operator. Example 12 shows portions of these requests as applied to a directory and file system.

4.5 Value Conversion

ADB can convert values from one representation to another. For example:

```
072 = odx
```

prints

```
072  58  %3a
```

which are the octal, decimal, and hexadecimal representations of 072 (octal). ADB keeps track of format so that as subsequent numbers are entered they are printed in the

previous formats. Character values are similarly converted. For example:

```
'a' = crb
```

prints

```
%0061
```

It can also evaluate expressions, but all binary operators have the same precedence, which is lower than for unary operators.

SECTION 5

PATCHING

Patching files with ADB is done with the _write_ (w or W) request, not to be confused with the _ed_ editor write command. This is often used in conjunction with the _locate_, (l or L) request.

The request syntax for l and w is:

```
address range  file designator  command  argument
```

where the address range gives the characters to be searched, the file designator is ? or /, the command is either a write or locate variation, and the argument is an expression and can support decimal and octal numbers or character strings. The address range can appear as zero, one, or two characters, including dot (current address). The request l is matched on two bytes, and L is used for four bytes. The request w writes two bytes, and W writes four bytes. For example,

```
0, 1000?l    searches the original file from 0 to 1000
   1000?l    searches the original file from 1000 to end
       ?l    searches the entire file
```

To modify a file, call ADB as

```
adb -w file1 file2
```

When called with this option, _file1_ and _file2_ are created and opened for both reading and writing.

For example, to change the word "This" to "The" in the executable file in Example 10, use the following requests:

```
adb -w ex7 -
 .?l 'Th'
 .?W 'The '
```

The request ?l starts at dot and stops at the first match of "Th" having set dot to the address of the location found. The use of ? writes to the _a.out_ file. The form ?* is used for an E711 file.

More frequently, the request is typed as:

```
?l 'Th'; ?s
```

This locates the first occurrence of "Th" and prints the entire string. Execution of this ADB request sets dot to the address of the "Th" characters.

Followng is an example of the utility of the patching facility that has a C program with an internal logic flag. The flag can be set through ADB and the program can be run.

```
adb a.out -
:s argl arg2
flag/w 1
:c
```

The :s request is normally used to single step through a process or start a process in single-step mode. In this case, it starts _a.out_ as a subprocess with arguments argl and arg2. If there is a subprocess running, ADB writes to it rather than to the file. The w request causes _flag_ to be changed in the memory of the subprocess.

SECTION 6

CAUTIONS

ADB has the following idiosyncrasies:

1. The value of local variables cannot currently be printed.

2. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.

3. When printing addresses, ADB uses either text or data symbols from the a.out file. This sometimes causes unexpected symbol names to be printed with data (for example, savr5+022). This does not happen if ? is used for text or instructions and / is used for data.

APPENDIX A

PROGRAM EXAMPLES

```
1
2    char *charp = "this is a sentence";
3
4    main( argc, argv )
5    int argc;
6    char **argv;
7    {
8         int     fd;
9         char cc;
10        if (argc < 2 )
11        {
12                printf("Input file missing\n");
13                exit(8);
14        }
15
16        if ( (fd = open(argv[1],0))== -1)
17        {
18                printf("%s : nor found\n", argv[1]);
19                exit (8);
20        }
21        charp = "hello";
22   printf("debug 1 %s\n", charp );
23        while( charp++ )
24                write (fd, *charp, 1);
25   {
***1*      **
```

Example 1

```
1     adb a.out core
2
3     ADB: S8000 1.1
4     ? $c
5     Stack backtracing not implemented
6     ? $C
7     Stack backtracing not implemented
8     ? $r
9     r0    %0000
10    r1    %0000
11    r2    %0000
12    r3    %0000
13    r4    %0000
14    r5    %0000
15    r6    %0000
16    r7    %0000
17    r8    %0000
18    r9    %0000
19    r10   %0000
20    r11   %0000
21    r12   %0000
22    r13   %0000
23    r14   %0000
24    sp    %0000
25    fcw   %0000
26    pc    %0000
27    _main:     jr          _main+%7c
28    ? $e
29    _charp:           %1400
30    __iob:            %1172
31    __sobuf:          %0000
32    __lastbu:         %0f5e
33    __sibuf:          %0000
34    _environ:         %ffa6
35    _end:             %0000
36    nd:               %1374
37    _errno:           %0009
38    ? $m
39    ? map      'a.out'
40                b1 = %0        e1 = %f72       f1 = %38
41                b2 = %0        e2 = %f72       f2 = %38
42    / map           'core'
43                b1 = %0        e1 = %1400      f1 = %400
44                b2 = %fa00     e2 = %10000     f2 = %1800
45    ? *charp/s
```

```
46    _end+%8c:
47    data address not found
48    ? charp/s
49    _charp:
50    ? main.argc/d
51    Sorry, local variable names not implemented
52    ? $q
***1*       **
```

Example 2

```
1     int fcnt, gcnt, hcnt:
2     h(x,y)
3     {
4          int hi; register int hr;
5          hi = x+1;
6          hr = x-y+1;
7          hcnt++;
8          f(hr,hi);
9     }
10
11    g(p,q)
12    {
13         int gi; register int gr;
14         gi = q-p;
15         gr = q-p+1;
16         gcnt++;
17         h(gr,gi);
18    }
19
20    f(a,b)
21    {
22         int fi; register int fr;
23         fi = a+2*b;
24         fr = a+b;
25         fcnt++;
26         g(fr,fi);
27    }
28
29    main()
30    {
31         f(1,1);
32    }
***1*       **
```

Example 3

```
1     adb
2
3     ADB:  S8000 1.1
4     ? $c
5     Stack backtracing not implemented
6     ? ,5$C
7     Stack backtracing not implemented
8     ? fcnt/d
9     _fcnt:          2156
10    ? gcnt/d
11    _gcnt:          2156
12    ? hcnt/d
13    _hcnt:          2157
14    ? h. x/d
15    Sorry, local variable names not implemented
16    ? $q
***1*       **
```

Example 4

```
1     #define MAXLINE     80
2     #define YES         1
3     #define NO          0
4     #define TABSP       8
5
6     char input[] = "data";
7     int tabs[ MAXLINE ];
8
9     main()
10    {
11         int fd;
12         int col, *ptab;
13         char c;
14
15         ptab = tabs;
16         settab(ptab);
17         col = 1;
18         if ((fd = open(input, 0 )) == -1 )
19         {
20              printf("%s : not found\n", input );
21              exit( 8 );
22         }
23
24         while(read(fd, &c, 1) > 0 )
25         {
26              switch(c)
27              {
28                   case '\t':
29                        while(tabpos(col) != YES)
30                        {
31                             putchar( ' ' );
32                             col++;
33                        }
34                        break;
35
36                   case '\n':
37                        putchar('\n');
38                        col = 1
39                        break;
40
41                   default:
42                        putchar(c);
43                        break;
44              }
45         }
46    }
47
48    tabpos(col)
49    int col;
50    {
51         if (col > MAXLINE )
52              return(YES);
```

```
53          else
54                return(NO);
55    }
56
57    settab(tabp)
58    int *tabp;
59    {
60          int i;
61
62          for ( i=0; i <=MAXLINE; i++ )
63                (i % TABSP ) ? (tabs[i] = NO : (tabs[i] = YES);
64    }
***1*        **
```

Example 5

```
1     adb a.out -
2
3     ADB: S8000 1.1
4     ? settab:b
5     ? open:b
6     ? read:b
7     ? tabpos:b
8     ? $b
9     breakpoints
10    count     bkpt     command
11    1     _tabpos
12    1     _read
13    1     _open
14    1     _settab
15    ? settab, 5ia
16    _settab:        jr   _settab%48
17    _settab+%2:     clr  %0002(sp)
18    _settab+%6:     cp   %0002(sp),#%0050
19    _settab+%c:     jr   gt,_settab+%44
20    _settab+%e:     ld   r3,%0002(sp)
21    _settab+%12:
22    ? settab,5?i
23    _settab:        jr   _settab+%48
24                    clr  %0002(sp)
25                    cp   %0002(sp),#%0050
26                    jr   gt,_settab+%44
27                    ld   r3,%0002(sp)
28    ? :r
29    fig5:  running
30    breakpoint      _settab:  jr   _settab+%48
31    ? settab:d
32    ? :c
33    fig5:  running
34    breakpoint      _open:    ld   r0,r7
35    ? $C
36    Stack backtracing not implemented
37    ? tabs/8x
38    _tabs:    %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
39              %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
40              %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
41    ? :c
42    fig5:  running
43    breakpoint      _read:        ld   r0,r7
44    ? :c
45    fig5:  running
46    breakpoint      _read:        ld   r0,r7
47    ? tabpos:d
48    ? settab:b settab,5?ia
49    ? settab,5:b settab,5?ia;  0
50    ? read,3:b tabs/8x
51    ? $b
52    breakpoints
```

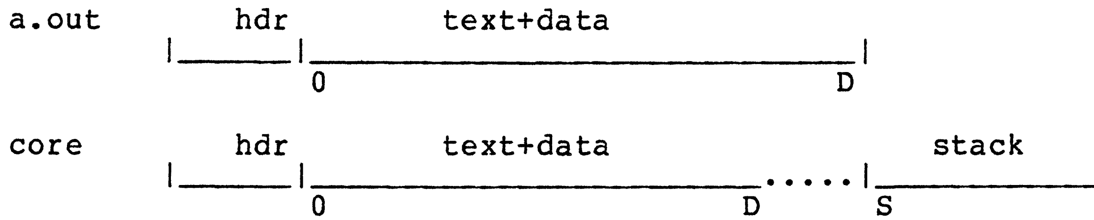
```
53   count       bkpt      command
54   3      _read    tabs/8x
55   1      _settab  settab,5?ia;  0
56   1      _open
57   ? fig5:  running
58   T_tabs:   %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
59   h_tabs:   %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
60   i_tabs:   %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
61   sbreakpoint    _read:          ld   r0,r7
62   ? $q
***1*      **
```

Example 6

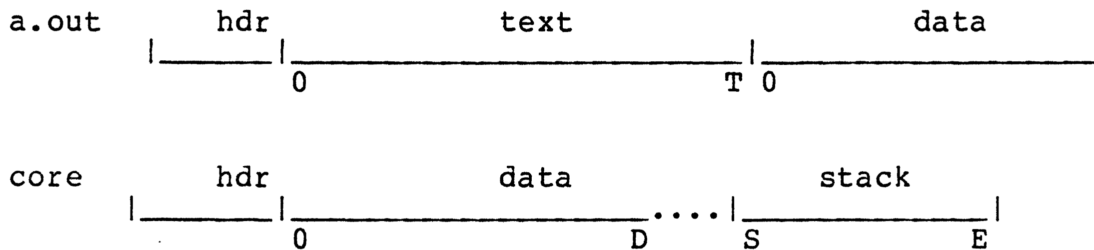
```
1     adb ex3 -
2
3     ADB: S8000 1.1
4     ? h:b hcnt/d; h.hi/; h.hr/
5     ? g:b gcnt/d; g.gi/; f.fr/
7     ? :r
8     ex3:  running
9     _fcnt:              0
10    Sorry, local variable names not implemented
11    ? f:b fcnt/d; f.a/"a = "d; f.h/"b = "d; f.fi/"fi = "d
12    ? g:b gnt/d; g.p/"p = "d; g.q/"q = "d; g.gi/"gi = "d
13    ? h:b hcnt/d; h.x/"x = "d; h.y/"y = "d; h.hi/"hi = "d
14    ? :r
15    ex3:  running       0
17    Sorry, local variable names not implemented
18    ? $q
***l*       **
```

Example 7

E707 files



E711 files (separated I and D space)



The following adb variables are set.

		E707	RM	E711
b	base of data	0	b	0
d	length of data	D	D-B	D
s	length of stack	S	S	S
t	length of text	0	T	T

Example 8

```
1     adb mapE707 coreE707
2
3     ADB:  S8000 1.1
4     ? $m
5     ? map           'mapE707'
6                       bl = %0       el = %dc       fl = %38
7                       b2 = $0       e2 = %dc       f2 = %38
8     / map           'coreE707'
9                       bl = %0       el = %100      fl = %400
10                      b2 = %200     e2 = %1000     f2 = %500
11    ? $v
12    variables
13      address
14    e = %a4
15      other
16    d = %100
17    m = %e707
18    s = %fe00
19    ? $q
20
21
22    adb mapE711 coreE711
23
24    ABD:  S8000 1.1
25    ? $m
26    ? map           'mapE711'
27                      bl = %0       el = %100      fl = %38
28                      b2 = %0       e2 = %0        f2 = %138
29    / map           'coreE711"
30                      bl = %0       el = %100      fl = %400
31                      b2 = %200     e2 = %10000    f2 = %500
32    ? variables
33      address
34    e = %a4
35      other
36    d = %100
37    m = %e711
38    s = %fe00
39    t = %100
40    ? $q
***l*     ***
```

Example 9

```
1    char      str1[] =  "This is character string";
2    int       one =     1;
3    int       number =  456;
4    long      lnum =    1234L;
5    char      str2[] =  "This is the second character string";
6    main()
7    {
8         one = 2;
9    }
***1*      **
```

Example 10

```
1   adb mapE711 coreE711
2
3   ADB: S8000 1.1
4   ? <b,-1/8oa
5   _strl       052150  064563  020151  071440  060440 061550 060562 060543
6
7   _strl+%10:    072145  071040  071564  071151  067147 000000 000001 000710
8
9   _lnum:      000000  002322  037640  000000  052150 064563 020151 071440
10
11   _str2+%8:    072150  062440  071545  061557  067144 020143 064141 071141
12
13   _str2+%18:    061564  062562  020163  072162  064556 063400 000000 177662
14
15   _environ+%2:  000000  000000  000000  000000  000000 000000 000000 000000
16
17   _environ+%12: 000000  000000  000000  000000  000000 000000 000000 000000
18
19   _environ+%22: 000000  000000  000000  000000  000000 000000 000000 000000
20
21   _environ+%32: 000000  000000  000000  000000  000000 000000 000000 000000
22
23   _environ+%42: 000000  000000  000000  000000  000000 000000 000000 000000
24
25   _environ+%52: 000000  000000  000000  000000  000000 000000 000000 000000
26
27   _environ+%62: 000000  000000  000000  000000  000000 000000 000000 000000
28
29   _environ+%72: 000000  000000  000000  000000  000000 000000 000000 000000
30
31   _environ+%82: 000000  000000  000000  000000  000000 000000 000000 000000
32
33   _environ+%92: 000000  000000  000000  000000  000000 000000 000000 000000
34
35   _environ+%a2: 000000  000000  000000  000000  000000 000000 000000 000000
36   ? <b,20/4on^8Cn
37   _strl:      052150  064563  020151  071440  This is
38          060440  061550  060562  060543  a charac
39          072145  071040  071564  071151  ter stri
40          067147  000000  000001  000710  ng@`@`@`@a@aH
41          000000  002322  037640  000000  @`@`@dR? @`@`
42          052150  064563  020151  071440  This is
43          072150  062440  071545  061557  the seco
44          067144  020143  064141  071141  nd chara
45          061564  062562  020163  072162  cter str
46          064556  063400  000000  177662  ing@`@`@`@2
47          000000  000000  000000  000000  @`@`@`@`@`@`@`@`
48          000000  000000  000000  000000  @`@`@`@`@`@`@`@`
49          000000  000000  000000  000000  @`@`@`@`@`@`@`@`
50          000000  000000  000000  000000  @`@`@`@`@`@`@`@`
51          000000  000000  000000  000000  @`@`@`@`@`@`@`@`
52          000000  000000  000000  000000  @`@`@`@`@`@`@`@`
```

```
53          000000  000000  000000  000000  @`@`@`@`@`@`@`@`
54          000000  000000  000000  000000  @`@`@`@`@`@`@`@`
55          000000  000000  000000  000000  @`@`@`@`@`@`@`@`
56          000000  000000  000000  000000  @`@`@`@`@`@`@`@`
57  ? <b,20/4o4^8t8cna
58  _strl:      052150  064563  020151  071440    This is
59  _strl+%8:    060440  061550  060562  060543    a charac
60  _strl+%10:    072145  071040  071564  071151    ter stri
61  _strl+%18:    067174  000000  000001  000710    ngH
62  _lnum:      000000  002322  037640  000000    R?
63  _str2:      052150  064563  020151  071440    This is
64  _str2+%8:    072150  062440  071545  061557    the seco
65  _str2+%10:    067144  020143  064141  071141    nd chara
66  _str2+%18:    061564  062562  020163  072162    cter str
67  _str2+%20:    064556  063400  000000  177662    ing2
68  _environ+%2:  000000  000000  000000  000000
69  _environ+%a:  000000  000000  000000  000000
70  _environ+%12: 000000  000000  000000  000000
71  _environ+%1a: 000000  000000  000000  000000
72  _environ+%22: 000000  000000  000000  000000
73  _environ+%2a: 000000  000000  000000  000000
74  _environ+%32: 000000  000000  000000  000000
75  _environ+%3a: 000000  000000  000000  000000
76  _environ+%42: 000000  000000  000000  000000
77  _environ+%4a: 000000  000000  000000  000000
78  _environ+%52:
79  ? <b,10/2b8t^2cn
80  _strl:      %0054   %0068    Th
81          %0069   %0073    is
82          %0020   %0069    i
83          %0073   %0020    s
84          %0061   %0020    a
85          %0063   %0068    ch
86          %0061   %0072    ar
87          %0061   %0063    ac
88          %0074   %0065    te
89          %0072   %0020    r
90  ? $q
*** 3168***
```

Example 11

```
1     adb dir -
2
3     ADB:  S8000 1.1
4     ? =nt"Inode"t"Name"
5     ? 0,-1?utl4cn
6                    Inode Name
7     %0000:            2  .
8                       2  ..
9                     102  bin
10                    101  usr
11                         157  lib
12                    164  dev
13                    148  etc
14                    197  pb.image
15                    957  tmp
16                    261  zeus3_1.2
17    ? $q
18
19
20
21    adb /dev/src -
22
23    ADB:  S8000 1.1
24    ? ?m 0 %1000000 1024
25    ? 0,-1?"flags"8ton"links,uid,gid"8t3dn"size"8tDn" \
        addr"8t20un"times"8t2Y2na
26    %0000:      flags    100000
27                links,uid,gid          0      0       0
28                size               0
29                addr          0      0      0       0       0      0      0
30                   0       0      0      0       0       0      0      0
31                   0       0      0      0       0
32                times    1981 Feb 12 13:50:17   1981 Feb 12 13:50:17
33
34                1981 Feb 12 13:50:17
35
36    %0040:     flags    040755
37               links,uid,gid          44      0       0
38               size               704
39               addr          3     9984    810       0     0      0     0
40                  0       0       0      0       0     0      0
41                  0       0       0      0       0
42               times    1981 Jul 17 16:58:42    1981 Jul 15 10:10:41
43
44               1981 Jul 15 10:10:41
45
46    %0080:     flags    100664
47               links,uid,gid           1      25      0
48               size               34
49               addr         52   12288     0      0      0      0     0
50                  0       0       0       0      0      0      0
51                  0       0       0       0      0
```

```
52          times    1981 Jul 16 17:06:34   1981 Jul 16 17:04:23
53
54          1981 Jul 16 17:94:23
```

Example 12

APPENDIX B

ADB SUMMMARY

Command Summary

- Formatted Printing

? *format*	print from *a.out* file according to *format*
/ *format*	print from *core* file according to *format*
= *format*	print the value of *dot*
?w expr	write expression into *a.out* file
/w expr	write expression into *core* file
?l expr	locate expression in *a.out* file

- Breakpoint and Program Control

:b	set breakpoint at *dot*
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run *a.out* file under ADB control
:s	single step

- Miscellaneous Printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

- Calling the Shell

!	call *shell* to read rest of line

- Assignment to Variables

  >name assign dot to variable or register name

Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	Z8000 instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
nt	move to next n space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

Expression Summary

- Expression Components

decimal integer	for example 256
octal integer	for example 0277
hexadecimal	for example %ff
symbols	for example flag _main main.argc
variables	for example <b
registers	for example <pc <r0
(expression)	for example expression grouping

- Dyadic Operators

+	add
-	subtract
*	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

- Monadic Operators

~	not
*	contents of location
-	integer negate

ZEUS PLZ/ASM ASSEMBLER

USER GUIDE

PREFACE

This manual describes how to use the Z8000 PLZ/ASM language translator (as) for the ZEUS Operating System. The Z8000 PLZ/ASM language is described in the Z8000 PLZ/ASM Assembly Language Programming Manual (03-3055). Implementation-dependent features are described in this document.

The S8000 version of PLZ/ASM depends on certain features of ZEUS. It uses the stream Input/Output (I/O) package to handle files, but otherwise is self-contained and system-independent. It is the resident assembler for ZEUS. A description of its exact invocation is contained in Section 1 of the ZEUS Reference Manual.

Refer to a.out(5) of the ZEUS Reference Manual, and to Section 7 of this manual, for a description of the object code format.

TABLE OF CONTENTS

SECTION 1 INTRODUCTION 4

1.1 General Description 4
1.2 Relocatability 4
1.3 Assembler Abort Conditions 4

SECTION 2 INPUT/OUTPUT 5

2.1 User Input 5
2.2 Assembler Output 5

SECTION 3 ASSEMBLER COMMAND LINE 6

3.1 Command Line 6
3.2 Options 6

SECTION 4 LISTING FORMAT 8

4.1 Format Description 8
4.2 Sample Listing 9

SECTION 5 MINIMAL PROGRAM REQUIREMENTS 10

SECTION 6 IMPLEMENTATION FEATURES AND LIMITATIONS 12

SECTION 7 OBJECT CODE 13

SECTION 8 PLZ/ASM ERROR MESSAGES 17

SECTION 1

INTRODUCTION

1.1 General Description

The Z8000 PLZ/ASM assembler (invoked by the command _as_) is the relocating assembler for ZEUS. It accepts a source file (a symbolic representation of a program in Z8000 assembly language) and translates it into an object module. It can also produce a listing file containing the source and assembled code.

1.2 Relocatability

Relocation refers to the ability to bind a program module and its data to a particular memory area after the assembly process. The output of the assembler is an object module that contains enough information to allow a loader or linker to assign a memory area to that module. Refer to the description of the ZEUS linker/loader in _ld_(_1_) of the _ZEUS Reference Manual_.

1.3 Assembler Abort Conditions

There are two assembler abort conditions.

1. If I/O errors are returned during a system call, an error is printed out and the assembly is aborted.

2. If error conditions cause the assembler to become completely lost, the assembly is aborted and an Assembler Abort error (error 255) is printed out to the standard error and the listing file.

SECTION 2

INPUT/OUTPUT

2.1 User Input

An editor is used to create a Z8000 PLZ/ASM source program. The source file should end with the file name extension .s (upper or lowercase). Instructions for invoking the assembler are defined in Section 3.

2.2 Assembler Output

The assembler creates two files: a listing file, with the default name of the source file and the extension .l in place of .s, and an object file, with a.out or t.out as the default name (Section 7). In creating the object file, the assembler uses a temporary, intermediate file that is deleted when the assembly is complete. The listing file contains the source statements and corresponding line numbers; any error message numbers are listed following the line on which the error occurred. Refer to Section 8 for explanations of error messages.

SECTION 3

ASSEMBLER COMMAND LINE

3.1 Command Line

The assembler is invoked by the following command line:

```
as filename [options]
```

The extension .s, which specifies that filename contains the source for a single Z8000 PLZ/ASM module, must be appended to filename.

3.2 Options

The following options are valid and can appear in any order, separated by delimiters such as a blank or tab.

-d string	in combination with the -l option, specifies a date (up to 19 characters) to be put in the listing header.
-f	allows assembly of floating point Extended Processor Unit (EPU) instructions.
-i	requests that the intermediate file the assembler uses be saved. The file name for the intermediate file is the input file name with the .i extension.
-l	requests a listing file. The file name for the listing file is the input file name with the .l extension. No listing is produced if this option is not used.
-o filename	allows the user to name the output file. If this option is not used, the default file name is a.out or t.out (Section 7).
-p	prints the listing file to the user console as it is being produced. Only source lines containing errors are printed to the console if this option is not specified.
-r	requests that the relocation information file be saved. The relocation file name is the input file name with the .r extension.

-u all undefined symbols are treated as external.

-v turns on the console message (name and version number, pass1 message, and assembly complete).

-z causes the assembler to produce type z object format rather than a.out. Also causes the default file name to be t.out rather than a.out (Section 7).

-^ turns on the pass1 trace facility.

SECTION 4

LISTING FORMAT

4.1 Format Description

The assembler produces a listing of the source program, along with generated object code. The various fields in the listing format are described in this section. Refer also to the sample listing in Section 4.2.

HEADING — The first page heading contains the assembler version number and column headings as explained below. In addition, the heading can contain a user-specified string that is usually the date of the assembly (see Date option, Section 3.2).

LOC — The location column contains the value of the reference counter for statements. The counter starts at zero for each different section.

OBJ CODE — The object code column contains the value of generated object code. It is blank if a statement does not generate object code.

Each byte or word of object code is followed by either a single quote ('), an asterisk (*), or a blank line. A single quote indicates that the value is relocatable. An asterisk indicates that the value is dependent on an external symbol. A blank indicates that the value will not change. A value that is either relocatable or dependent on an external is likely to be modified by either the linker or loader. The value in the listing can be different from the value during program execution. Three dots (...) indicate that the preceding byte, word, or long word is repeated (only in data initialization).

STMT — The statement number column contains the sequence number of each source line.

SOURCE — The remainder of the line contains the source text.

4.2 Sample Listing

```
Z8000ASM 3.0
LOC  OBJ CODE    STMT SOURCE STATEMENT
                    1 bubble_sort MODULE      ! Module declaration !
                    2
                    3 CONSTANT                ! Constant declarations !
                    4     FALSE  := 0
                    5     TRUE   := 1
                    6
                    7 EXTERNAL
                    8     list ARRAY [10 WORD]
                    9
                   10 INTERNAL
0000               11     switch BYTE           ! Loop control switch !
                   12
0000               13 sort PROCEDURE            ! Procedure declaration !
                   14 ENTRY                     ! Begin executable part !
                   15   DO                      ! Loop til EXIT !
0000 4C05 0000'    16    LDB switch,#FALSE      ! Initialize switch !
0004 0000
0006 8D18          17     CLR R1                ! Clear array pointer i !
                   18     DO
0008 0B01          19       CP  R1,R0           ! Done ?.!
000A E701          20       IF UGE THEN EXIT FI
000C E811
000E A112          21       LD  R2,R1           ! Initialize pointer j !
0010 A921          22       INC R2,#2           ! j = i+1 (dble for words)!
0012 6114 0000*    23       LD  R4,list(R1)
0016 6126 0000*    24       LD  R6,list(R2)
001A 8B64          25       CP  R4,R6           ! If list[i] > list[j]... !
001C E307          26       IF  UGT THEN        ! ..exchange to bubble... !
001E 4C05 0000'    27         LDB switch,#TRUE  ! ...largest to top !
0022 0101
0024 6F16 0000*    28         LD list(R1),R6
0028 6F24 0000*    29         LD list(R2),R4
                   30       FI
002C A911          31       INC R1,#2           ! Advance word pointer !
002E E8EC          32     OD                    ! End nested DO loop !
0030 4C01 0000'    33     CPB switch,#FALSE     ! Test switch !
0034 0000
0036 EE01          34     IF EQ THEN RET FI
0038 9E08
003A E8E2          35   OD                      ! End outer DO loop !
003C               36 END sort                  ! End of procedure !
                   37
                   38 GLOBAL                    ! New procedure declaration
003C               39 main PROCEDURE            ! Program entry procedure !
                   40 ENTRY
003C 2100 0012     41     LD  R0,#9*2           ! Initialize loop control !
                   42                           ! Double for word array !
0040 D021          43     CALR sort             ! Call sort procedure !
0042 9E08          44     RET
0044               45 END main                  ! End of main procedure !
                   46
                   47 END bubble_sort

    0 errors
Assembly complete
```

SECTION 5

MINIMAL PROGRAM REQUIREMENTS

The examples in this section illustrate the minimal amount of PLZ/ASM structuring required to make a working program. The first example shows the absolute minimal structuring required: a module definition, a declaration class, and a procedure definition. The second example shows the same program, but includes examples of how to use symbolic constants and data declarations.

EXAMPLE #1:

```
anyname MODULE

GLOBAL      ! or INTERNAL depending on whether !
            ! intermodule linking is desired. !

somename PROCEDURE
ENTRY

    ! The program goes here !
    RET

END somename

END anyname
```

EXAMPLE #2:

```
    anyname MODULE

    CONSTANT  ! Symbolic constants are declared here. !

        one := 1
        hexten := %10

    GLOBAL    ! or INTERNAL depending on whether !
              ! intermodule linkage is desired. !

       a BYTE ! Data declarations can go here. !
       b WORD
       buffer ARRAY [100 BYTE]

    GLOBAL    !Restate the declaration class [optional]. !

    somename PROCEDURE
    ENTRY

        ! The program goes here!
        RET

    END somename

    END anyname
```

SECTION 6

IMPLEMENTATION FEATURES AND LIMITATIONS

The Z8000 PLZ/ASM assembler limitations and implementation features follow.

1. The Z8000 PLZ/ASM assembler uses the standard ASCII character set. Upper or lowercase characters are recognized and treated as different characters; keywords are recognized only if they are either all upper or all lowercase (GLOBAL or global, but not Global). Hexadecimal numbers and special string characters can be either upper or lowercase (%Ab, '1st line%R2nd line%r').

2. Source lines longer than 132 characters are accepted, but only 132 characters are printed for error messages. Comments and quoted strings can extend over an arbitrary number of lines. Caution should be exercised to avoid unmatched comment delimiters (!) or string delimiters (').

3. Strings cannot be zero length ('').

4. Constants are represented internally as 32-bit unsigned quantities. Each operand in a constant expression is evaluated as though it were declared to be of type LONG. For example, 4/2 equals 2, but 4/-2 equals zero since -2 is represented as a very large unsigned number. There is no overflow checking during evaluation of a constant expression. Because constants are represented as 32-bit values, only the first four characters in a character sequence used as a constant are meaningful ('ABCD' = 'ABCDE'). An exception is a string used for array initialization, which can have a length of up to 127 characters.

5. Identifiers can be of any length up to a maximum of 127 characters.

6. After an error occurs within CONSTANT, TYPE, or variable declarations, the assembler skips ahead until it finds the next keyword that starts a new statement (an opcode, IF, DO, EXIT, REPEAT, or END). This skipping ahead may necessitate several assemblies before all errors are detected and removed.

SECTION 7

OBJECT CODE

Depending on command line options, the assembler produces object files in one of two formats: object code compatible with that produced by the MCZ Z8000 PLZ/ASM assembler (t.out) and ZEUS object code (a.out). Refer to Section 3 for the appropriate command-line options.

When producing ZEUS object code, a.out is the default file name. This object code format is fully described in a.out(5) of the ZEUS Reference Manual.

When producing MCZ object code, t.out is the default file name. Below is a list of the object tags, their functions, and the corresponding fields that make up this object code format. The tags are classified into three groups: control tags that are used to transfer control information, entry tags that define the code, and modifier tags that act as modifiers for the entry tags.

The following is a list of symbols used in the object code syntax:

| The vertical bar separates two mutually exclusive items. The user enters one or the other, but not both. Multiple vertical bars separate three or more mutually exclusive items. Parameters separated by a vertical bar can be delimited by brackets (see below).

* An asterisk placed after an item indicates that the item appears zero or more times in the syntax.

+ A plus sign placed after an item indicates that the item appears at least once in the syntax.

[ ] Brackets enclose an optional parameter--a parameter that can appear zero or more times.

( ) Parentheses enclose parameter pairs, or group items so that a repetition symbol (+ or *) can be applied to the group.

' ' Single quotes enclose character strings that must be entered with a particular parameter. However, the single quotes only delimit the required character string and must not appear in the command line.

OBJECT CODE SYNTAX

The object code format is still under development and is subject to change.

```
object_module      =>  [tagged_entry]*

tagged_entry       =>  control_entry | modified_entry

control_entry      =>  NOP
                   =>  SEGMODULE bcount size size name
                   =>  NONSEGMODULE bcount size name
                   =>  ENDMODULE
                   =>  SECTION bcount attr size name
                   =>  GLOB bcount secw loc attr typew name
                   =>  ABSGLOB bcount secw loc attr typew name
                   =>  EXTERN bcount typew name
                   =>  ENTRYPT sec loc
                   =>  ABSENTRYPT sec loc
                   =>  DEBUGSYMBOL bcount secw loc [bval]*
                   =>  DEBUGINFO bcount [bval]*
                   =>  MESSAGE bcount [bval]*
                   =>  SETDATA sec
                   =>  SETPROG sec
                   =>  BEGSEC sec
                   =>  LOCNT loc
                   =>  ABSLOCNI loc
                   =>  MODULEDEF secw loc wval size
                   =>  MODULEREF wval

modified_entry     =>  [REP bcount]
                       (modified_addr | modified_value)

modified_addr      =>  [SHORT] [SEGMENT | OFFSET]
                            [HIBYTE | LOBYTE]
                            [DISP offset] addr_entry

modified_value     =>  [(REL sec) | RELPROG | RELDATA]
                            [SEQUENCE bcount] value_entry

addr_entry         =>  EXREF ext
                   =>  SECREF sec
                   =>  SECADDR sec offset
                   =>  ZREF ext
```

```
value_entry          =>  LDBYTE bval
                     =>  LDWORD wval
                     =>  LDLONG lval

name                 =>  [byte]*
length               =>  byte
size                 =>  word
attr                 =>  byte
sec                  =>  byte
secw                 =>  word
loc                  =>  word
type1                =>  byte
type2                =>  byte
bval                 =>  byte
wval                 =>  word
lval                 =>  long
count                =>  word
ext                  =>  word
offset               =>  word
```

OBJECT CODE TAGS

CONTROL TAGS:

HEX		
00	NOP	Null operation
01	SEGMODULE	Segmented module definition
02	NONSEGMODULE	Nonsegmented module definition
03	ENDMODULE	End module
04	SECTION	Section definition
05	GLOB	Global symbol definition
06	ABSGLOB	Global symbol definition with absolute offset
07	EXTERN	External symbol definition
08	ENTRYPT	Entry point with relocatable offset
09	ABSENTRYPT	Entry point with absolute offset
0A	DEBUGSYMBOL	Debug symbol
0B	DEBUGINFO	Debug information
0C	MESSAGE	Variable length message
0D	SETDATA	Set current data section
0E	SETPROG	Set current program section
0F	BEGSEC	Begin section
10	LOCNT	Relocatable program counter
11	ABSLOCNT	Absolute program counter
12	MODULEDEF	Module definition for z-code
13	MODULEREF	Module reference used for z-code machines

ENTRY TAGS:

HEX		
20	LDBYTE	Load byte value
21	LDWORD	Load word value
22	LDLONG	Load long value
23	EXREF	External reference
24	SECREF	Section reference
25	SECADDR	Section address
26	ZREF	Z-code module reference

MODIFIER TAGS:

HEX		
40	REP	Repeat
41	SEQUENCE	Sequence
42	REL	Relocatable
43	RELDATA	Relocatable with respect to current data area
44	RELPROG	Relocatable with respect to current program area
45	DISP	Displacement
46	*LOBYTE	Low order byte of
47	*HIBYTE	High order byte of
48	**SHORT	Short segment address
49	**OFFSET	Offset of
4A	**SEGMENT	Segment of

* Z8/Z-UPC
** Z8000

SECTION 8

PLZ/ASM ERROR MESSAGES

ERROR	EXPLANATION
	WARNINGS
1	Missing delimiter between tokens
2	Array of zero elements
3	No fields in record declaration
4	Mismatched procedure names
5	Mismatched module names
8	Absolute address warning for System 8000
	TOKEN ERRORS
10	Decimal number too large
11	Invalid operator
12	Invalid special character after %
13	Invalid hexadecimal digit
14	Character_sequence of zero length
15	Invalid character
16	Hexadecimal number too large
	DO LOOP ERRORS
20	Unmatched OD
21	OD expected
22	Invalid repeat statement
23	Invalid exit statement
24	Invalid FROM label
	IF STATEMENT ERRORS
30	Unmatched FI
31	FI expected
32	THEN or CASE expected
33	Invalid selector record
	SYMBOLS EXPECTED
40	) expected
41	(expected
42	] expected
43	[expected
44	:= expected

ERROR EXPLANATION

INVALID VARIABLES

100 Invalid variable
101 Invalid operand for # or SIZEOF
102 Invalid field name
103 Subscripting of nonarray variable
104 Invalid use of period (.)

EXPRESSION ERRORS

110 Invalid arithmetic expression
111 Invalid conditional expression
112 Invalid constant expression
113 Invalid select expression
114 Invalid index expression
115 Invalid expression in assignment

CONSTANT OUT OF BOUNDS

120 Constant too large for 8 bits
121 Constant too large for 16 bits
122 Constant array index out of bounds

TYPE INCOMPATIBILITY

140 Character_sequence initializer used with array [*] declaration where component's base type is not 8 bits
141 TYPE incompatibility with initilization

SEGMENTATION ERRORS

170 Invalid operator in nonsegmented mode
171 Mismatched short address operator
172 Mismatched segment designator

DIRECTIVE ERRORS

180 Inconsistent area specifier
181 Invalid area specifier
182 Mismatched conditional assembly directives
183 Invalid conditional assembly expression
184 Attempt to mix segmented and nonsegmented code
185 Directive must appear alone on a single line
186 Invalid $CODE or $DATA directive

ERROR EXPLANATION

FILE ERRORS

198 EOF expected
199 Unexpected EOF encountered in source--possible unmatched ! or ' in source

IMPLEMENTATION RESTRICTIONS

224 Too many symbols--hash table full
226 Short segmented offset out of range
227 Object symbol table overflow
228 Relocation out of range (word overflow)
229 Unimplemented feature
230 Character_sequence of identifier too long
231 Too many symbols--symbol table full
234 Too many initialization values
235 Stack overflow
236 Operand too complicated

NOTE

Errors larger than 240 can occur. If there are no other errors in the program preceding one of these errors, this indicates an assembler bug that should be reported to Zilog along with any pertinent information concerning its occurence.

Awk - A Pattern Scanning and Processing Language *

* This information is based on an article originally written by Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, Bell Laboratories.

TABLE OF CONTENTS

SECTION 1 INTRODUCTION 3

1.1 Usage 3
1.2 Program Structure 4
1.3 Records and Fields 4

SECTION 2 PATTERNS 7

2.1 BEGIN and END 7
2.2 Regular Expressions 7
2.3 Relational Expressions 8
2.4 Combinations of Patterns 9
2.5 Pattern Ranges 9

SECTION 3 ACTIONS 11

3.1 Built-in Functions 11
3.2 Variables, Expressions, and Assignments 12
3.3 Field Variables 12
3.4 String Concatenation 13
3.5 Arrays 14
3.6 Flow-of-Control Statements 14

SECTION 4 DESIGN 17

SECTION 5 IMPLEMENTATION 19

SECTION 1

INTRODUCTION

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The basic operation of _awk_ is to scan a set of input lines in order, searching for lines which match any of a set of patterns which the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern.

Readers familiar with the ZEUS program _grep_ (see ZEUS Reference Manual, Section 1) will recognize the approach, although in _awk_ the patterns may be more general than in _grep_, and the actions allowed are more involved than merely printing the matching line. For example, the _awk_ program

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

prints the third and second columns of a table in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev     { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

1.1. Usage

The command

```
awk  program  [files]
```

executes the _awk_ commands in the string program on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file pfile, and executed by the command

```
awk  -f pfile  [files]
```

1.2. Program Structure

An *awk* program is a sequence of statements of the form:

pattern { *action* }

pattern { *action* }

...

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

1.3. Records and Fields

Awk input is divided into ``records'' terminated by a record separator. The default record separator is a newline, so by default *awk* processes its input a line at a time. The number of the current record is available in a variable named NR.

Each input record is considered to be divided into ``fields.'' Fields are normally separated by white space - blanks or tabs - but the input field separator may be changed, as described below. Fields are referred to as $1, $2, and so forth, where $1 is the first field, and $0 is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named NF.

The variables FS and RS refer to the input field and record separators; they may be changed at any time to any single character. The optional command-line argument -F*c* may also be used to set FS to the character *c*.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable FILENAME contains the name of the current input

file.

1.4. Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the _awk_ command print. The _awk_ program

```
{ print }
```

prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined variables NF and NR can be used; for example

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

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

```
{ print $1 >"foo1"; print $2 >"foo2" }
```

writes the first field, $1, on the file foo1, and the second field on file foo2. The >> notation can also be used:

```
print $1 >>"foo"
```

appends the output to the file foo. (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

```
print $1 >$2
```

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process for instance,

```
print | "mail bwk"
```

mails the output to bwk.

The variables OFS and ORS may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the print statement.

Awk also provides the printf statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in format and prints them. For example,

```
printf "%8.2f  %10ld\n", $1, $2
```

prints $1 as a floating point number 8 digits wide, with two after the decimal point, and $2 as a 10-digit long decimal number, followed by a newline. No output separators are produced automatically; you must add them yourself, as in this example. The version of printf is identical to that used with C.

SECTION 2

PATTERNS

A pattern in front of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

2.1. BEGIN and END

The special pattern BEGIN matches the beginning of the input, before the first record is read. The pattern END matches the end of the input, after the last record has been processed. BEGIN and END thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN     { FS = ":" }

... rest of program ...
```

Or the input lines may be counted by

```
END  { print NR }
```

If BEGIN is present, it must be the first pattern; END must be the last if used.

2.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete _awk_ program which will print all lines which contain any occurrence of the name ``smith''. If a line contains ``smith'' as part of a larger word, it will also be printed, as in

```
blacksmithing
```

Awk regular expressions include the regular expression forms found in the ZEUS text editor _ed_ (see ZEUS Reference Manual,

Section 1) and grep (without back-referencing). In addition, awk allows parentheses for grouping, | for alternatives, + for ``one or more'', and ? for ``zero or one'', all as in lex. Character classes may be abbreviated: [a-zA-Z0-9] is the set of all letters and digits. As an example, the awk program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

will print all lines which contain any of the names ``Aho,'' ``Weinberger'' or ``Kernighan,'' whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in ed and sed. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn of the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
/\/.*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and !~. The program

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches ``john'' or ``John.'' Notice that this will also match ``Johnson'', ``St. Johnsbury'', and so on. To restrict it to exactly [jJ]ohn, use

```
$1 ~ /^[jJ]ohn$/
```

The caret ^ refers to the beginning of a line or field; the dollar sign $ refers to the end.

2.3. Relational Expressions

An awk pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF % 2 == 0
```

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
$1 >= "s"
```

selects lines that begin with an s, t, u, etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

will perform a string comparison.

2.4. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example,

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with ``s'', but is not ``smith''. && and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

2.5. Pattern Ranges

The ``pattern'' that selects an action may also consist of two patterns separated by a comma, as in

```
pat1, pat2     { ... }
```

In this case, the action is performed for each line between an occurrence of pat1 and the next occurrence of pat2 (inclusive). For example,

```
/start/, /stop/
```

prints all lines between start and stop, while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

SECTION 3

ACTIONS

An *awk* action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

3.1. Built-in Functions

Awk provides a ``length'' function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

length by itself is a ``pseudo-variable'' which yields the length of the current record; length(argument) is a function which yields the length of its argument, as in the equivalent

```
{print length($0), $0}
```

The argument may be any expression.

Awk also provides the arithmetic functions sqrt, log, exp, and int, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function substr(s, m, n) produces the substring of s that begins at position m (origin 1) and is at most n characters long. If n is omitted, the substring goes to the end of s. The function index(s1, s2) returns the position where the string s2 occurs in s1, or zero if it does not.

The function sprintf(f, e1, e2, ...) produces the value of the expressions e1, e2, etc., in the printf format specified by f. Thus, for example,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets x to the string produced by formatting the values of $1 and $2.

3.2. Variables, Expressions, and Assignments

Awk variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns 7 to x. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most BEGIN sections. For example, the sums of the first two fields can be computed by

```
      { s1 += $1; s2 += $2 }
END   { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, -, *, /, and % (mod). The C increment ++ and decrement -- operators are also available, and so are the assignment operators +=, -=, *=, /=, and %=. These operators may all be used in expressions.

3.3. Field Variables

Fields in *awk* share essentially all of the properties of variables - they may be used in arithmetic or string operations, and may be assigned to. Thus one can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
      $3 = "too big"
  print
}
```

which replaces the third field by ``too big'' when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the the string s into array[1], ..., array[n]. The number of elements found is returned. If the sep argument is provided, it is used as the field separator; otherwise FS is used as the separator.

3.4. String Concatenation

Strings may be concatenated. For example

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a print statement,

```
print $1 " is " $2
```

prints the two fields separated by `` is ''. Variables and numeric expressions may also appear in concatenations.

3.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have _any_ non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the NR-_th_ element of the array x. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the _awk_ program

```
        { x[NR] = $0 }

END     { ... program ... }
```

The first action merely records each input line in the array x.

Array elements may be named by non-numeric values, which gives _awk_ a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like apple, orange, etc. Then the program

```
/apple/    { x["apple"]++ }

/orange/   { x["orange"]++ }

END        { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

3.6. Flow-of-Control Statements

Awk provides the basic flow-of-control statements if-else, while, for, and statement grouping with braces, as in C. We showed the if statement in section 3.3 without describing it. The condition in parentheses is evaluated; if it is true, the statement following the if is done. The else part

is optional.

The while statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1

while (i <= NF) {

     print $i

     ++i

}
```

The for statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)

     print $i
```

does the same job as the while statement above.

There is an alternate form of the for statement which is suited for accessing the elements of an associative array:

```
for (i in array)

     statement
```

does _statement_ with i set in turn to each element of array. The elements are accessed in an apparently random order. Chaos will ensue if i is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an if, while or for can include relational operators like <, <=, >, >=, == (``is equal to''), and != (``not equal to''); regular expression matches with the match operators ~ and !~; the logical operators ||, &&, and !; and of course parentheses for grouping.

The break statement causes an immediate exit from an enclosing while or for; the continue statement causes the next iteration to begin.

The statement next causes _awk_ to skip immediately to the next record and begin scanning the patterns from the top. The statement exit causes the program to behave as if the end of the input had occurred.

Comments may be placed in *awk* programs: they begin with the character # and end with the end of the line, as in

```
print x, y     # this is a comment
```

SECTION 4

DESIGN

The ZEUS system already provides several programs that operate by passing input through a selection mechanism. *Grep*, the first and simplest, merely prints all lines which match a single specified pattern. *Egrep* provides more general patterns, i.e., regular expressions in full generality; *fgrep* searches for a set of keywords with a particularly fast algorithm. *Sed* provides most of the editing facilities of the editor *ed*, applied to a stream of input. None of these programs provides numeric capabilities, logical relations, or variables.

Lex provides general regular expression recognition capabilities, and, by serving as a C program generator, is essentially open-ended in its capabilities. The use of *lex*, however, requires a knowledge of C programming, and a *lex* program must be compiled and loaded before use, which discourages its use for one-shot applications.

Awk is an attempt to fill in another part of the matrix of possibilities. It provides general regular expression capabilities and an implicit input/output loop. But it also provides convenient numeric processing, variables, more general selection, and control flow in the actions. It does not require compilation or a knowledge of C. Finally, *awk* provides a convenient way to access fields within lines; it is unique in this respect.

Awk also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases the user can simply ignore the differences.

Most of the effort in developing *awk* went into deciding what *awk* should or should not do (for instance, it doesn't do string substitution) and what the syntax should be (no explicit operator for concatenation) rather than on writing or debugging the code. The syntax is powerful but easy to use and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, *awk* usage seems to fall into two broad categories. One is what might be called ``report generation'' - processing an input to extract counts, sums, sub-

totals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is as a data transformer, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

SECTION 5

IMPLEMENTATION

The actual implementation of _awk_ uses the language development tools available on the ZEUS operating system. The grammar is specified with _yacc_; the lexical analysis is done by _lex_; the regular expression recognizers are deterministic finite automata constructed directly from the expressions. An _awk_ program is translated into a parse tree which is then directly executed by a simple interpreter.

Awk was designed for ease of use rather than processing speed; the delayed evaluation of variable types and the necessity to break input into fields makes high speed difficult to achieve in any case. Nonetheless, the program has not proven to be unworkably slow.

As might be expected, _awk_ is not as fast as the specialized tools _wc_, _sed_, or the programs in the _grep_ family, but is faster than the more general tool _lex_. The tasks are about as easy to express as _awk_ programs as programs in these other languages; tasks involving fields are considerably easier to express as _awk_ programs.

THE C PROGRAMMING LANGUAGE

PREFACE

The S8000 system uses the C programming language almost exclusively. The operating system, ZEUS, and a majority of the programs are written in C. This document supplements the information in _The C Programming Language_ by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). The reader should be familiar with the basic concepts of C before reading this document.

Despite its universality, each installation contains machine dependencies that affect the C programming language. Also, as a dynamic language, C reflects changes to handle situations not previously addressed. This document describes these machine dependencies and C language changes.

Conversion of programs to the ZEUS system is described in Section 1. Machine and object format dependencies, the setret and longret routines, and the problems encountered when passing parameters in registers are discussed.

Recent changes to the C language not documented in _The C Programming Language_ are discussed in Section 2.

TABLE OF CONTENTS

SECTION 1 CONVERSION OF PROGRAMS TO ZEUS 4

1.1 Introduction 4
1.2 Setret and Longret Routines 4
1.3 Impact of Passing Parameters in Registers 4
1.4 Object Format Dependencies 9
1.5 Byte Order Within Words 9
1.6 Machine Architecture Dependencies 11
1.7 C Compiler Features 11

SECTION 2 RECENT CHANGES TO C 13

2.1 General 13
2.2 Structure Assignment 13
2.3 Enumeration Type 13

SECTION 1

CONVERSION OF PROGRAMS TO ZEUS

1.1 Introduction

Although the standard Version 7 UNIX runs on the S8000 system and the S8000 C compiler accepts the C language, users must be aware of machine dependencies that may be present in their programs. This section describes the places for users to look for machine dependencies in their programs when trying to bring them up on the S8000 system.

1.2 Setret and Longret Routines

When using the C language routine on the S8000 system, there are problems of declaring register variables when _setjmp_ and _longjmp_ are used. Replacing _setjmp_ and _longjmp_ with _setret_ and _longret_ and removing the register attribute of variable declarations causes the program to continue to function as on PDP-11 UNIX.

The S8000 C compiler's stackframes are different from the PDP-11 UNIX. The S8000's contain only one register that is used as both the frame pointer and stack pointer. It is not possible to move back up the subroutine call chain (as the PDP-11 UNIX does) to restore the register variables.

1.3 Impact of Passing Parameters in Registers

The Z8000 processor has a larger register file than the PDP-11 processor. To use these registers efficiently, parameters are passed in registers on the S8000 instead of being passed on stack as on the PDP-11. Programs using parameters that are passed on the stack and then picked off from the stack do not work on the S8000 system. Most programs need only to be recompiled to accomodate this change. In cases when procedures handle a a variable number of parameters, however, a special process must be followed, as described in the paragraphs that follow.

Figures 1-1 and 1-2 illustrate how a machine-dependent program with a variable number of parameters can change to accomodate parameter passing in the registers. Figure 1-1 shows a program running on PDP-11 with arguments picked off from the stack. This program can have up to two pointer arguments. The same program is shown in Figure 1-2 with changes to handle parameter passing in the registers.

```
/*
**         This program allocates  space for up to two
**         string  arguments  and then copies  them in
**         the  allocated  space.  The  first argument
**         (na) is  the  number of arguments  and  the
**         second (ap) and the  third (optional) argu-
**         ments  are  the pointers to the  strings to
**         be copied.  It  returns a  pointer  to  the
**         location where the strings have been copied.
**         have been copied.
*/
char *
copy(na, ap)
char *ap;
{
     register char  *p, *np;
     char       *onp;
     register int       n;

     p = ap;
     n = 0;
     if (*p == 0)
          return 0;
     do
     {
          n++;
     } while (*p++);
     if (na > 1)
     {
          p = (&ap)[1];
          while (*p++)
               n++;
     }
     onp = np = alloc(n);
     p = ap;
     while (*np++ = *p++)
          continue;
     if (na > 1)
     {
          p = (&ap)[1];
          np--;
          while (*np++ = *p++)
               continue;
     }
     return onp;
}
```

Figure 1-1. Example of PDP-11 Program

```
char *
copy(na, apl, ap2)
char *apl, *ap2;
{
     reg char  *p, *np;
     char      *onp;
     reg int        n;

     p = apl;
     n = 0;
     if (*p == 0)
          return 0;
     do
     {
          n++;
     } while (*p++);
     if (na > 1)
     {
          p = ap2;
          while (*p++)
               n++;
     }
     onp = np = alloc(n);
     p = apl;
     while (*np++ = *p++)
          continue;
     if (na > 1)
     {
          p = ap2;
          np--;
          while (*np++ = *p++)
               continue;
     }
     return onp;
}
```

Figure 1-2. S8000 Version of Figure 1 Program

Modifying programs with a variable number of arguments of different types is difficult. Figure 1-3 shows a routine with a variable number of arguments of different types. This is a version of the C library routine printf, modified to illustrate parameter passing in registers.

```
#define R7        0 /* prcnt == 0 implies r7 already seen */
#define R5        0 /* prcnt == 0 implies r5 already seen */
#define R3        0 /* prcnt == 0 implies r3 already seen */
#define prmax  5    /* max. number of register parameters */
#define true 1
/*
**       Routine to align parameter pointer consistent with
**       the Z8000 calling conventions. It skips over
**       unused registers. This happens in C only for long
**       parameters passed in registers.
*/
zalign(prcnt, ip, stk)
int *prcnt;     /* parameter count */
int **ip;       /* pointer to low-order word of long word */
int *stk;       /* address of first parameter in the stack */
{
   int t;
            /* long cannot start in r6 or r4 */
   if (*prcnt == R7 || *prcnt == R5)
   {
     (*prcnt)++;      /* skip over the unused register */
     (*ip)++;
   }
   else if (*prcnt == R3)      /* long cannot start in r2 */
   {
     *prcnt += 2;    /* skip over r2 */
     *ip = &(*stk);  /* parameter comes from the stack */
     return;
   }
   /* exchange order of the words in a long word; they were
      inverted when they were put into local storage */
   t = **ip;
   **ip = *(*ip + 1);
   *(*ip + 1) = t;
}
/*
**  An example routine using a variable number of parameters
**  each of which can be a different size.  This is a sample
**  of a formatted I/O routine.
*/
```

```
printz(fmt,r6,r5,r4,r3,r2,stack)
register unsigned char *fmt;  /* pointer to format string */
int     r6,r5,r4,r3,r2; /* parameters passed in registers */
int     stack;          /* first parameter in the stack */
{
  int  pr6;    /* storage for parameter register 6 */
  int  pr5;    /* the order of declaration of storage for */
  int  pr4;    /* parameter registers has two effects: */
  int  pr3;    /* first, long words have their words */
  int  pr2;    /* exchanged; second, the pointer to
               /* parameter storage can be incremented */
               /* for parameters in registers and the stack */
  int  prcnt;  /* number of parameters seen */


   int  i;
   union{
        int    *ip;
        long   *lp;
     } x;
   /* save register parameters in storage */
   pr6 = r6;
   pr5 = r5;
   pr4 = r4;
   pr3 = r3;
   pr2 = r2;
   x.ip = &pr6;
   prcnt = 0;
   while (true)
   {    /* once through for each format character */
     i = *fmt++;
     switch(i)
     {
          case ' ': return;       /* end of format */

          case '%': i = *fmt++;
                  switch(i)
                  {
               case 'd': putint(*x.ip++);
                     break;

               case 'D': if (prcnt < prmax)
                           zalign(&prcnt,&x.ip,&stack);
                      putlong(*x.lp++);
                           /*second word done below*/
                      prcnt++;
                      break;

               case 'c': putchar(*x.ip++);
                      break;
```

```
                default:  putchar('%');
                          putchar(i);
                          break;
                }
                prcnt++;
                if (prcnt == prmax)
                     /* start using stack parameters */
                     x.ip = (int *)&stack;
                break;

            default: putchar(i);
                 break;
        }
    }
}
main ()
{
printz("%c0,'z');
printz("double: %D0,1L);
printz("decimal: %d0,69);
printz("%c%c%c%c%c%c%c0,'a','b','c','d','e','f','g');
printz("%D %D %D %D0,100L,123456L,1L,98765432L);
printz("%D %d %c %d0,32L,10,'x',52);
}
```

Figure 1-3. An S8000 Program with Variable Number of Arguments of Different Types

1.4 Object Format Dependencies

Programs that extract header information from the object files must be modified. Typical UNIX utilities that look at the object files (for example make and nlist) are already available on the S8000. The entire object file produced by the language processors on the S8000 conform to the S8000 object code format. Refer to a.out (5) for a complete description of the S8000 object code format.

1.5 Byte Order Within Words

Byte order on the S8000 differs from byte order on the PDP-11. On the S8000, the high-order byte of a word has an even address and the low-order byte has the next higher odd address. On the PDP-11, this is reversed. This means that the PDP-11 programs that manipulate bytes within a word or long quantities with pointers may not work correctly on the

S8000. Also, transporting files between a S8000 and a PDP-11 requires any word quantities within the file to be byte-swapped.

For example, suppose that starting at memory location 100, there is a string of eight bytes (all numbers are in hex):

    00, 01, 02, 03, 04, 05, 06, 07

On both the PDP-11 and the Z8000, these values occupy the eight consecutively addressed locations 100-107. However, consider the word value at location 102. On the Z8000, 02 is the high-order byte, so the value is 0203. On the PDP-11, 03 is the high-order value, so the value is 0302. Manipulations such as:

```
char *p;
int  i;
i = (*p++*256) + *p++;
```

produce different results on the two machines.

To illustrate the problemm of transferring files between the two machines, consider the string to have originated on the PDP-11 as a structure containing four byte values followed by two word values:

```
100: 00
101: 01
102: 02
103: 03
104: 0504
105: 0706
```

When this string is moved to a Z8000, it becomes:

```
100: 00
101: 01
102: 02
103: 03
104: 0405
105: 0607
```

So, before the data can be processed, the words at 104 and 106 must have the bytes reserved, while the bytes at 100 through 103 must not be changed.

1.6 Machine Architecture Dependencies

Another architecture dependency concerns the use of the /dev/mem device. On the PDP-11, the system data space begins at location 0 of /dev/mem. On the S8000, this system instruction space begins at 0. A program such as ps that needs to examine locations in the system data memory must use the device /dev/kmem instead of /dev/mem (mem(4)).

The -n option, which takes advantage of the PDP-11's 8K page size, is not supported. The S8000 has a 64K page size. The -i option (separate I&D) can be used instead. Both options link a program so that several copies of the same program can share the first several pages.

1.7 C Compiler Features

The ZEUS C compiler allows register variables of types short, int, pointer, long, and double. These can be unsigned where appropriate. Declarations of register float or char are ignored. In nonsegmented mode, there are seven ordinary registers and four floating (double) registers available for register variables. In segmented mode, the number of ordinary registers is reducted to six.

The sizes of the various variable types are as follows:

Type	Size (in bits)
character	8
unsigned character	8
short	16
unsigned short	16
int	16
unsigned int	16
pointer (nonsegmented)	16
pointer (segmented)	32
long	32
unsigned long	32
float	32
double	64
register double	80 (IEEE format)

Although 80 bits are used internally for register double variables, this does not mean that results will be accurate to 80 bits. For example, in the statement

```
register double    d=1.1;
```

only 64 bits fo the floating representation of 1.1 are used

to initialize d. In converting PDP-11 C programs to S8000 C programs, be aware that the PDP-11 C compiler (CC) does not do sign extension when characters are cast as unsigned.

PDP-11 C programs that contain expressions like

```
(unsigned) C
```

where C is a character, must be changed to

```
(unsigned character) C
```

to suppress sign extension on the S8000.

The legal source file names for C programs are restricted to contain only alphanumeric, period (.), and minus (-) characters. This restriction exists because the file name is used in constructing the module name for the assembler.

SECTION 2

RECENT CHANGES TO C

2.1 General

A few extensions have been made to the C language described in The C Programming Language. This section discusses these extensions.

2.2 Structure Assignment

Structures can be assigned, passed as arguments to functions, and returned by functions. The types of operands taking part must be the same.

NOTE

> There is a limitation to the C language in ZEUS implementation of functions that return structures. If an interrupt occurs during the return sequence and the same function is called again during the interrupt, the value returned from the first call can be corrupted. The problem can occur only in the presence of true interrupts, as in an operating system or a user program that makes significant use of signals. Ordinary recursive calls are safe.

2.3 Enumeration Type

There is a data type similar to the scalar types of PASCAL. To the type-specifiers in the syntax on page 193 of The C Programming Language, add

```
enum-specifier
```

with syntax

```
enum-specifier:

     enum { enum-list }
     enum identifier { enum-list }
     enum identifier
```

enum-list:

```
     enumerator
     enum-list, enumerator
```

enumerator:

```
     identifier
     identifier = constant-expression
```

The role of the identifier in the _enum-specifier_ is similar to the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret, winedark };
...
enum color *cp, col;
```

makes _color_ the enumeration tag of a type describing various colors, and then declares _cp_ as a pointer to an object of that type and _col_ as an object of that type.

The identifiers in the _enum list_ are declared as constants, and can appear wherever constants are required. If no enumerators appear with the equal sign (=), the values of the constants begin at zero and increase by one as the declaration is read from left to right. An enumerator with the equal sign gives the associated identifier the value indicated. Subsequent identifiers continue the progression from the assigned value.

Enumeration tags and constants must be distinct and, unlike structure tags and members, are drawn from the same set as ordinary identifiers.

Objects with a given enumeration are distinct from objects of all other types. In ZEUS implementation, all enumeration variables are treated as integers.

ZEUS COMMUNICATIONS PACKAGE

PREFACE

This document describes the ZEUS Communications Package, a communication path between ZEUS and Zilog development tools.

In this document, the term "development system" refers to a standard Z8(TM) or Z8000(TM) Development Module or to Z-SCAN 8000(TM). The term "remote system" refers to a System 8000(TM) executing the ZEUS Operating System. The term "local system" refers to an MCZ(TM) or a ZDS system executing the RIO Operating System.

The LOAD/SEND function in ZEUS is analogous to the MCZ/ZDS LOAD/SEND function. Refer to the _Z8000 Development Module Hardware Reference Manual_ (03-3080) for specific information.

TABLE OF CONTENTS

SECTION 1 INTRODUCTION 4

SECTION 2 FUNCTIONAL DESCRIPTION 5

2.1 Upload/Download Functional Description 5
2.2 File Transfer Functional Description 5

SECTION 3 INVOCATION AND OPERATION 7

3.1 Upload/Download Invocation and Operation .. 7
3.2 File Transfer Invocation and Operation 7

SECTION 4 TERMINATION 9

4.1 Upload/Download Termination 9
4.3 File Transfer Termination 9

SECTION 1

INTRODUCTION

The ZEUS Communications Package gives the ZEUS user a communication path between ZEUS and the development tools offered by Zilog (the Z8 and Z8000 Development Modules and Z-SCAN 8000).

The upload/download capability includes the LOAD command, which loads a ZEUS file to development tool memory, and the SEND command, which transfers the contents of development tool memory to a ZEUS file. These facilities also interface with existing PROM programming products, giving the user PROM programming capability.

The package also provides a general-purpose file transfer capability for transferring files between a local system and a remote system. This includes software that executes under both ZEUS and the RIO Operating System.

NOTE

> This software package is not designed for communication between two ZEUS systems. For this capability, use the programs uucp, uux, and uulog.

SECTION 2

FUNCTIONAL DESCRIPTION

2.1 Upload/Download Functional Description

The LOAD command downloads a Z8000 program to a development system from a ZEUS file. The binary data in the file is converted to Tektronix format and is transmitted to the development system. An acknowledgment from the development system causes the next record to be downloaded from ZEUS. If an acknowledgment is not received, the current record is retransmitted up to ten times. After continued nonacknowledgment, a record with an error message is sent, and the program aborts.

Possible error messages are:

/ABORT
/UNABLE TO OPEN FILE
/FILENAME ERROR
/INCORRECT FILE TYPE
/ERROR IN READING FILE
/CHECKSUM ERROR

The SEND command transfers the contents of development system memory to a ZEUS file. The SEND program opens the file and sends an acknowledgment to the development system to start transmission. If the file cannot be opened, an abort-acknowledgment is sent, and the program aborts. An acknowledgment is sent after each good record received. If the ASCII code double slash (//) is received from the development system, the program aborts.

Possible error messages are:

/ABORT
/OPEN FILE ERROR
/FILE WRITE ERROR
/CHECKSUM ERROR

2.2 File Transfer Functional Description

The file transfer software copies files residing on the remote system to files residing on the local system, and vice versa. On invocation of the file transfer command (Section 3), the remote system transmits a sequence of characters to the local system to initiate the file transfer. A file is transferred one record at a time, along with a

checksum to guarantee the accuracy of the data. For each successful transmission, an acknowledgment is sent, and a period (.) is displayed on the terminal to inform the user that the transfer is proceeding. If a nonacknowledgment is sent, the record is retransmitted up to ten times, after which the program proceeds to the next file. An error message is displayed for each retransmission that is necessary, unless the nonfatal error messages are suppressed in the command invocation (Section 3). A message is printed after each successful transmission that includes the file name. At the conclusion of the program, a message informs the user of the number of successful and unsuccessful transmissions. A control-x causes the current file transfer to terminate, and the program proceeds to the next file on the list. The termination message counts that file as an unsuccessful transfer (Section 4.2). Pressing the escape key (ESC) aborts the program.

Possible messages are:

Normal transmission:
<filename>
. (one . for every record for positive feedback)

Error messages:
checksum error ... retry
<filename> ... transmission aborted

ZEUS file names cannot be longer than 14 characters, but RIO file names can be as long as 32 characters. For file transfers from the local system to the remote system, only the first 14 characters of the file name are used. Path names can be specified; they apply only to the file name on the remote system. On the local system, all files to be uploaded must be in the working directory, and all downloaded files are created in the working directory (this does not apply to the MCZ/ZDS systems).

NOTE

If a duplicate file name exists on the target system, the contents of pre-existing files are automatically overwritten unless the [-q] option is specified as part of the command (Section 3). If the [-q] option is specified, the user is queried for a replacement name.

Possible message is:

replace <filename> (y/n)?

SECTION 3

INVOCATION AND OPERATION

3.1 Upload/Download Invocation and Operation

The LOAD command is given to the development system as follows:

LOAD <filename>

The development system Monitor program transmits the command line to ZEUS exactly as it is entered, and the ZEUS program (LOAD) opens the file specified by <filename>. The Monitor on a Z8000 Development Module or Z-SCAN requires that <filename> be all uppercase on the remote system. If "load prog" is entered, the remote system searches for the file PROG. The binary data in the file is transmitted to the development system. Pressing ESC aborts the LOAD command.

The SEND command is given to the development system as follows:

SEND <filename> <start address> <end address> [<entry address>]

This command transfers the contents of development system memory to a ZEUS file specified by <filename>. The development system transmits the command to ZEUS exactly as input, causing execution of the SEND program. SEND opens the file <filename> and stores in it the binary data received from the development system. Pressing ESC aborts the SEND command.

3.2 File Transfer Invocation and Operation

File transfer is accomplished in three steps. In the first step, control is transferred from the local system to the remote system by entering the following command to the local system.

remote [<rate>]

This command starts a program on the local system, which places the user in remote mode. In this mode, all characters entered from the keyboard are sent to the S8000, and all characters from the S8000 (except for character sequences that initiate file transfers and the return to local mode) are sent to the terminal screen. Therefore, the terminal is essentially operating as an S8000 terminal, and

any ZEUS command can be executed. The default communication rate is 9600 baud. Standard baud rates that can be specified for the MCZ/ZDS are 50, 75, 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19,200, and 38,400.

The second step in file transfer involves two commands: *putfile* and *getfile*, which are invoked as follows:

```
putfile [-q] [-f] [-b] [-B] <filenamel> [[-b] <filename2>...]
getfile [-q] [-f] [-b] [-B] <filenamel> [[-b] <filename2>...]
```

The command *putfile* transfers files from the remote system to the local system; *getfile* transfers files from the local system to the remote system.

The [-q] option specifies that transfer of a file to the target system where a file of the same name already exists causes a query to the user (Section 2.2). If this option is not given, the file is automatically overwritten.

The [-f] option suppresses the nonfatal error message "checksum error ... retry."

The [-b] option preceding a file name indicates a binary file and suppresses translation of ZEUS new line characters into RIO's carriage returns (and vice versa) for that file only. The type defaults to ASCII for the next file. This differs from the [-q] and [-f] options, which apply to the remainder of the line following the point at which they are invoked.

The [-B] option specifies that every file that follows is binary.

A list of files can be specified on the command line. A control-x aborts the transfer of a single file and proceeds to the next file. Pressing ESC aborts the entire transfer at any point.

The third step returns the user to the local system from the remote system. The command is:

```
local [-l]
```

The [-l] option causes a logout to be given to the remote system. It is necessary to log in after the next remote command.

SECTION 4

TERMINATION

4.1 Upload/Download Termination

After completion of the loading process, the program's entry point is displayed on the terminal, and the development system returns to Monitor mode. The LOAD program terminates and returns control to the ZEUS Operating System.

After completion of the sending process, the program's entry point is stored in the ZEUS file, and the development system returns to Monitor mode. The SEND program terminates and returns control to the ZEUS Operating System.

If there is a user or program abort during either the loading or sending process, an error message is printed (Section 2), the development system returns to Monitor mode, and the program returns control to the ZEUS Operating System.

4.2 File Transfer Termination

After completion of the file transfer, the local system returns to remote mode, enabling the user to continue to execute ZEUS commands. One of the following messages is printed on the terminal:

putfile:<nl> successful transfers <n2> unsuccessful transfers
getfile:<nl> successful transfers <n2> unsuccessful transfers

An unsuccessful file transfer does not cause the program to terminate abnormally. If the program is aborted via the escape key, it does not transfer any more files, and terminates in a normal fashion.

AN INTRODUCTION TO THE C SHELL*

* This information is based on an article orginally written by William Joy, University of California, Berkeley.

PREFACE

A shell is a command language interpreter; C shell, also known as csh is the name of an interactive command interpreter for ZEUS. Enter the command csh to call the program on the system. The primary purpose of csh is to translate command lines typed at a terminal into system actions, such as invocation of other programs. It incorporates features of other shells and a history mechanism similar to the redo of INTERLISP, all of which make csh easy to use.

This document gives instructions on the use of the csh and describes its capabilities. The last two sections describe features of the csh that are useful, but not necessary for every user. Appendix A lists characters that have special meaning for csh and ZEUS.

Appendix B is a glossary of terms and commands introduced in this document.

In addition to this document, refer to csh(1) of the ZEUS Reference Manual, which gives a full description of all features of csh.

Names of commands and words that have special meaning in csh and ZEUS are underlined. Refer to Appendix B to learn the meaning of any words that are unfamiliar.

TABLE OF CONTENTS

SECTION 1 INTERACTIVE USE 5

1.1 Commands 5
1.2 Flag Arguments 6
1.3 Output to Files 6
1.4 Metacharacters in Csh 7
1.5 Input from Files 8

1.5.1 Pipelines 8

1.6 File Names 9
1.7 Terminating Commands 12

SECTION 2 DETAILS OF CSH OPERATION 14

2.1 Csh Startup and Termination 14
2.2 Csh Variables 15
2.3 Csh's History List 16
2.4 Aliases 18
2.5 Detached Commands and Redirection .. 20
2.6 Built-In Commands 21

SECTION 3 CSH CONTROL STRUCTURES AND COMMAND SCRIPTS 25

3.1 Introduction 25
3.2 Invocation and the argv Variable ... 25
3.3 Variable Substitution 25
3.4 Expressions 27
3.5 Sample Csh Script 28
3.6 Other Control Structures 31
3.7 Applying Input to Commands 32
3.8 Catching Interrupts 32
3.9 Other Functions 33
3.10 Make 33

SECTION 4 MISCELLANEOUS SHELL MECHANISMS 34

4.1 Loops at the Terminal 34
4.2 Braces in Argument Expansion 35
4.3 Command Substitution 36

TABLE OF CONTENTS (continued)

APPENDIX A SPECIAL CHARACTERS 37

APPENDIX B GLOSSARY 38

SECTION 1

INTERACTIVE USE OF THE C SHELL

1.1 Commands

A shell in ZEUS is, primarily, a medium through which other commands are invoked. Csh has a set of built-in commands that it performs directly; however, most useful commands are external to the shell. What distinguishes csh from command interpreters of other systems is that it is a user program that acts almost exclusively as a mechanism for invoking other programs.

Commands in the ZEUS system expect a list of strings or words as arguments. For example, the command

```
mail bill
```

consists of two words. The first word, _mail_, names the command to be executed (in this case the mail program that sends messages to other users). Csh looks in a number of directories for a file with the name _mail_, which contains the mail program.

The rest of the words of the command are given to the command itself to execute. In this case, the word _bill_ is interpreted by the _mail_ program as the name of a user to whom mail is to be sent. The _mail_ command is normally used as follows:

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page five.
Does a page five exist?
     Chuck
%
```

A message is sent to _bill_ and is ended with a control-d, which sends an end-of-file message to the mail program. The mail program then transmits the message. The prompt character % is printed before and after the _mail_ command to indicate that input to csh to is needed.

After giving the % prompt, csh reads the command input from the terminal. After the command _mail bill_ is typed, csh executes the _mail_ program with argument _bill_ and waits for it to complete. The mail program reads input from the terminal until an end-of-file message notifies csh that _mail_ is

finished. Csh signals the user that it is ready to read from the terminal again by printing another % prompt.

This is the basic pattern of all interactions with ZEUS through csh. A complete command is typed at the terminal, csh executes the command, and, when execution is completed, prompts for a new command. This pattern is not affected by the time it takes to execute a command. If the editor is run for an hour, csh waits for editing to finish before prompting the user again.

1.2 Flag Arguments

Flag arguments normally begin with a dash character (-) and invoke an optional capability of the command. For example, the command

```
ls
```

produces a list of the files in the current directory. If the size option, _-s_, is added, as follows,

```
ls -s
```

_ls_ also gives the size of the file in blocks of 512 characters for each file. Refer to the _ZEUS Reference Manual_ for the available options for each command.

1.3 Output to Files

Many commands read from or write to files rather than taking input from and sending output to the terminal. These commands take special words as arguments, indicating where the output is to go. It is simpler, and usually sufficient, to connect these commands to the files to be written. This is done within csh just before the commands are executed.

The command

```
date
```

displays the current date on the terminal, which is the default standard output for the _date_ command. To save the current date in a file called now, it is possible to redirect the standard output. Csh allows the standard output of a command to be redirected through a notation using the metacharacter > and the name of the file where output is to be placed. Thus, the command

```
date > now
```

runs the <u>date</u> command with the file <u>now</u> as its standard output. This command then places the current date and time in the file <u>now</u>. It is important to realize that the <u>date</u> command is not affected by its output going to a file rather than to the terminal. Csh performs this redirection before the command begins execution.

The file <u>now</u> does not have to exist before the <u>date</u> command is executed; csh creates the file if it does not exist. If the file already exists, the previous contents are overwritten. The csh option <u>noclobber</u> (Section 2.2) prevents this from happening accidentally.

1.4 Metacharacters in Csh

Csh has a number of special characters (like >) that indicate special functions. Appendix A lists these metacharacters in functional groups. In general, most characters that are neither letters nor digits have special syntactic or semantic meaning to csh. Metacharacters normally have effect only when csh is reading input.

Metacharacters cannot be used directly as parts of words. For example, the command

```
echo *
```

does not echo the character *. It either echoes a sorted list of file names in the current directory, or prints the message <u>No match</u> if there are no files in the current directory.

The recommended mechanism for using metacharacters as arguments is to enclose them in single quotation marks ('), for example:

```
echo '*'
```

One special character, the exclamation mark (!), (used by the <u>history</u> mechanism of csh) cannot be escaped in this way. The ! and the single quote (') character can be preceded by a single backslash (\) to prevent special interpretation.

These two mechanisms suffice to place any printable character in a word that is an argument to a csh command.

1.5 Input from Files

Although it is also possible to route the standard input of a command from a file, this is not usually necessary because most commands are read from a file name given as an argument. The command

```
sort < data
```

runs the *sort* command with standard input, whereas the command normally reads from the file *data*. It is easier to enter

```
sort data
```

letting the *sort* command open the file *data* for input. If

```
sort
```

is entered, the *sort* program sorts lines from its standard input. Since the standard input is not redirected, it sorts lines as typed at the terminal until a control-d is typed to generate an end-of-file.

1.5.1 Pipelines

Csh can combine the standard output of one command with the standard input of the next. This procedure runs the commands in a sequence known as a *pipeline*. Commands separated by a vertical bar (|) are connected together by csh; the output of each is run into the input of the next. The leftmost command in a pipeline normally takes its standard input from the terminal, and the rightmost places its standard output on the terminal.

For example, the command

```
ls -s
```

produces a list of the files in the directory with the size of each in blocks of 512 characters. Combining the *ls* command with options of the *sort* command sorts the directory files by size rather than by name.

The -*n* option of *sort* specifies a numeric sort rather than an alphabetic sort. Combining this command with *ls -s* using the pipe command (|)

```
ls -s | sort -n
```

specifies that the output of the ls command, run with the option -s, is to be piped to the command sort, run with the numeric sort option. This gives a sorted list of files by size with the smallest first. Use the reverse sort (-r) option and the head command in combination with the previous command as follows:

```
ls -s | sort -n -r | head -5
```

The list of files is now sorted alphabetically, with the size of each in blocks. This is run to the standard input of the sort command, asking it to sort numerically in reverse order (largest first). This output is then run into the command head, which displays the first few lines of each file. In this example, head is asked to run the first five lines, so it gives the names and sizes of the five largest files.

1.6 File Names

Every ZEUS file has a file name up to 14 characters long. Every file is listed by name in a directory. The relationship of files to directories is expressed by path names.

ZEUS path names consist of a number of components separated by a slash (/). Each component, except the last, names a directory in which the next component resides. For example, the path name

```
/etc/motd
```

specifies a file (motd) in the directory etc, which is a subdirectory of the root directory (/). File names that do not begin with / are interpreted starting at the current working directory. This directory is, by default, the home directory. The home directory can be changed dynamically with the change directory (chdir or cd) command.

All printable characters except / can appear in file names, but characters that have special meaning should be avoided. Most file names consist of a number of alphanumeric characters and periods (.). The period character is not a shell metacharacter and is often used to separate an extension from a base name. For example,

```
prog.c prog.o prog.errs prog.output
```

are four files that share a root portion of a name. (A root portion is that part of the name that is left when a trailing period and following characters are stripped off.) The file prog.c is the source for a C program, the file prog.o

is the corresponding object file, the file *prog.errs* is the list of errors resulting from a compilation of the program, and the file *prog.output* is the output of a run of the program.

The metanotation

```
prog.*
```

can be used in a command to refer to all four of these files. This word is expanded by the shell (before the command to which it is an argument is executed) into a list of names that begin with *prog*. The asterisk (*) character matches any sequence (including the empty sequence) of characters in a file name. The names that match are sorted alphabetically into the list of command arguments. The command

```
echo prog.*
```

echoes the names

```
prog.c prog.errs prog.o prog.output
```

The names are in alphabetic order here (a different order than listed previously). The *echo* command receives four words as arguments, even though only one word is directly entered as an argument. The four words are generated by file name expansion of the metasyntax in the one input word.

Another metanotation for file name expansion is the question mark (?) character, which matches any single character in a file name. For example,

```
echo ? ?? ???
```

echoes a line of file names; first, those with one-character names, then those with two-character names, and finally, those with three-character names are echoed. The names of each type are independently sorted alphabetically.

Another mechanism consists of a sequence of characters between brackets ([ ]). This metasequence matches any single character from the enclosed set. For example,

```
prog.[co]
```

matches

```
prog.c prog.o
```

in the example above. Two characters separated by a hyphen (-) denote a range. For example,

chap.[1-5]

matches files

chap.1 chap.2 chap.3 chap.4 chap.5

if they exist. This is an abbreviated version of

chap.[12345]

and is otherwise equivalent.

If a list of arguments to a command contains file name expansion syntax, and if this syntax fails to match any existing file names, the shell considers this to be an error and prints

No match.

The period character (.) at the beginning of a file name is treated specially. The matching mechanisms *, ?, and [ ] do not match it. This prevents accidental matching of file names that have special meaning to the system (such as . and ..), and files (such as .cshrc) that are not normally visible.

Another file name expansion mechanism gives access to the path name of the home directory of other users. This notation consists of the tilde character (~) followed by another user's login name. For instance, the word ~*bill* maps to the path name /z/bill if the home directory for *bill* is the directory /z/bill. On large systems, users can have login directories scattered over many different disk volumes with different prefix directory names. This notation provides a reliable way of accessing the files of other users.

A special case of this notation consists of a ~ alone, for example ~/mbox. This notation is expanded by csh into the file *mbox* in the user's home directory. This can be very useful if the user uses *chdir* to change to another user's directory and then decides to copy a file there using *cp*. The entry

cp thatfile ~

is expanded by csh to

cp thatfile /z/bill

which the copy command interprets as a request to make a copy of *thatfile* in the directory /z/bill. Unlike the matching characters (*, ?, and []), the ~ notation does not, by itself, force named files to exist. This is useful when using the *cp* command, as in

```
cp thatfile ~/saveit
```

Braces ({ }) can be used for abbreviating a set of words that have common parts, but cannot be abbreviated by the above mechanisms because they are not files, are the names of files that do not yet exist, or are not conveniently described by the other mechanisms. This mechanism is described in Section 4.3.

1.7 Terminating Commands

It is possible to terminate programs that are running while csh is dormant without terminating csh itself. For instance, if the command

```
cat /etc/passwd
```

is entered, the system displays on the terminal a list of all users of the system. Pressing the DEL or RUB key sends an interrupt signal to the *cat* command and terminates it. Actually, pressing the key sends the interrupt signal to all programs running on the terminal, including csh. Csh normally ignores such signals, however, so that the only program affected by the interrupt is *cat*, which has no mechanism for ignoring interrupts. Upon termination of the command, csh leaves the dormant state and prompts the user with %. If the interrupt is entered again, csh simply repeats its prompt, since it catches (ignores) interrupt signals.

Many programs terminate when they receive an end-of-file message from their standard input. The *mail* program example in Section 1.1 was terminated when the user typed a control-d, which generates an end-of-file from the standard input. Csh also terminates when it receives an end-of-file. ZEUS then logs the user off the system. Since this means that typing too many control-d's can accidentally log the user off the system, csh has a mechanism for preventing this. This *ignoreeof* option is discussed in Section 2.2.

If the command has its standard input redirected from a file, it normally terminates when it reaches the end of this file. Thus, if

```
mail bill < prepared.text
```

is executed, the *mail* command terminates when it reads the end-of-file for the file *prepared.text*.

Programs that have not been fully debugged can be stopped by entering a control-\. Csh responds with a message similar to:

```
a.out: Quit -- Core dumped
```

This indicates that a file *core* has been created that contains information about the program *a.out*'s state when it encountered problems. This file can be examined by the user, or can be forwarded to the maintainer of the program describing where the *core* file is.

If background commands are running, they ignore interrupt and quit signals entered at the terminal. To stop the background commands, use the *kill* program. (See Section 2.6 for an example.)

SECTION 2

DETAILS OF CSH OPERATION

2.1 Csh Startup and Termination

When the user logs in, the system places the csh in the user's _home_ directory and begins by reading commands from the file ._cshrc_ in this directory. All user-created shells are read from this file.

After it reads commands from ._cshrc_, a _login_ shell (executed after the user logs in to the system) reads commands from a file, ._login_, also in the user's home directory. This file contains commands to be executed each time the user logs in to the ZEUS system. The following is an example of a typical ._login_ file:

```
setenv TERM adm3a
set history=20
set time=3
```

This file contains three commands executed by ZEUS each time the user logs in. The first is a _setenv_ command, which informs the system that this user usually dials in on a Lear-Siegler ADM-3A terminal.

The next two _set_ commands are interpreted directly by csh and affect the values of certain variables that modify the future behavior of csh. Setting the variable _time_ tells csh to print time statistics on commands that take more than a certain threshold of machine time (in this case three CPU seconds). Setting the variable _history_ tells csh how much history of previous command words it should save in case the user wants to repeat or rerun modified versions of previous commands. Since there is a certain overhead in this mechanism, csh does not set this variable by default; it allows users who wish to use the mechanism to set this variable themselves. The value of 20 is a reasonably large value to assign to _history_. A value of 5 or 10 is more commonly used. The use of the _history_ mechanism is described Section 2.3.

After executing commands from ._login_, csh reads commands from the user's terminal, prompting for each with %. When it receives an end-of-file from the terminal, csh prints _logout_ and executes commands from the file ._logout_ in the user's home directory. After that, csh terminates, and ZEUS logs the user off the system.

2.2 Csh Variables

Csh maintains a set of variables that have as a value an array of zero or more _strings_. Shell variables can be assigned values by the _set_ command. The most useful form of _set_ is

```
set name=value
```

Csh variables can be used to store values that are to be reintroduced into commands later through a substitution mechanism. The csh variables most commonly referenced are those referred to by csh itself. By changing the values of these variables, it is possible to directly affect the behavior of csh.

One of the most important variables is _path_, which contains a sequence of directory names where the shell searches for commands. The _set_ command shows the value of all variables currently defined in csh. The default value for _path_ is shown by _set_ to be

```
% set
argv
home     /z/bill
path     (. /bin /usr/bin)
prompt   %
shell    /bin/csh
jtatus0
%
```

This notation indicates that the variable _path_ points to the current directory (.), then /bin, and finally /usr/bin. Commands that the user can write might be in . (usually one of the user's directories). The most heavily used system commands reside in /bin and less heavily used system commands reside in /usr/bin.

A useful built-in variable is _home_, which shows the user's home directory. The variable _ignoreeof_ can be set in the ._login_ file to tell csh not to exit when it receives an end-of-file from the terminal. To log out from ZEUS with _ignoreeof_ set, type

```
logout
```

To set this variable, type

```
set ignoreeof
```

and, to unset it, type

unset ignoreeof

Both _set_ and _unset_ are built-in commands of csh.

Another built-in csh variable is _noclobber_, which prevents files from being overwritten. The metasyntax

> filename

(which directs the output of a command) overwrites and destroys the previous contents of the named file. A file that is valuable can be accidentally overwritten. To prevent csh from overwriting files in this way, enter

set noclobber

in the ._login_ file. Then, entering

date > now

causes a diagnostic if _now_ already exists. The special metasyntax >! indicates that "clobbering" the file is allowable. Entering

date >! now

makes it possible to overwrite the contents of _now_.

The variable _mail_ is also built in. To be notified of the arrival of mail while logged in, place the following command in the ._login_ file:

set mail=/z/mail/yourname

Csh checks this file every 10 minutes to see if new mail has arrived. Since this variable can delay the shell's response while it checks for mail, use it only if mail arrives frequently.

The use of csh variables to introduce text into commands, which is most useful in csh command scripts, is introduced in Section 2.4.

2.3 The C Shell's History List

Csh can maintain a history list that contains the words of previous commands. It is possible to use a metanotation to reintroduce commands, or words from commands, to form new commands, repeat previous commands, or to correct minor typing mistakes in commands.

The following transcript asks the system where *michael* is logged in.

```
% where michael
michael is on tty0   dialup      300 baud    642-7927
% write !$
write michael
Long time no see michael.
Why don't you call me at 524-4510.
EOF
%
```

The system specifies that he is on tty0. Csh is then told to invoke a write command to !$. This is a history notation that means the last word of the last command executed--in this case, michael. Csh performs this substitution, and then echoes the command as it is executed. The following interchange might take place if there is no response from *michael*.

```
% ps -t0
  PID TTY TIME COMMAND
 4808  0    0:05 -
% !!
ps -t0
  PID TTY TIME COMMAND
 5104  0    0:00 - 7
% !where
where michael
michael is not logged in
%
```

A *ps* on the teletype *michael* is logged in on is run to see if he has a shell. Repeating this command via the history substitution, !!, shows that he has logged out and that only a *getty* process is running on his terminal. Repeating the *where* command shows that he is indeed gone.

This illustrates several useful features of the history mechanism. The form !! repeats the last command execution. The form !*string* repeats the last command that began with a word, of which *string* is a prefix. Another useful command form is ↑*lhs*↑*rhs*, which performs a substitute similar to that in *ed* or *ex*. Thus, after

```
% cat ~bill/csh/sh..c
/mnt/bill/csh/sh..c: No such file or directory
% ↑..↑.
cat ~bill/csh/sh.c
#include "sh.h"

/*
```

```
 * C Shell
 *
 * Bill Joy, UC Berkeley
 * October, 1978
 */

char *pathlist[] =  { SRCHP
%
```

the substitution is used to correct a typing mistake, then rub out the command after the file is located. The substitution changes the two periods (..) to a single period (.).

The following command can then be used to put a copy of this file on the line printer:

```
% !! | lpr
cat ~bill/csh/sh.c | lpr
```

Or, immediately after the _cat_, the following can be used to print a copy on the printer using _pr_:

```
% pr !$ | lpr
pr ~bill/csh/sh.c | lpr
%
```

More advanced forms of the history mechanism are also possible. A notion of modification on substitutions makes it possible to say (after the first successful _cat_)

```
% cd !$:h
cd ~bill/csh
%
```

The trailing :h on the history substitution causes only the head portion of the path name reintroduced by the history mechanism to be substituted. This mechanism and related mechanisms are used less often than the other forms.

A complete description of history mechanism features is given in _csh_(_1_) in the _ZEUS Reference Manual_.

2.4 Aliases

The shell has an _alias_ mechanism that makes transformations on input commands by simplifying the commands typed, supplying default arguments to commands, or performing transformations on commands and their arguments. The alias facility is similar to the macro facility of many assemblers.

Some of the features obtained by "aliasing" can also be obtained using csh command files, but these take place in another instance of the shell and cannot directly affect the current shell's environment and commands.

As an example, suppose that there is a new version of the mail program on the system called _Mail_, that is to be used instead of the standard mail program (which is called _mail_). If the csh command

```
alias mail Mail
```

is placed in the user's ._login_ file, csh transforms an input line of the form

```
mail bill
```

into a call on _Mail_.

To cause the command _ls_ to show sizes of files (that is, to do _-s_), enter

```
alias ls ls -s
```

or even

```
alias dir ls -s
```

which creates a new command syntax _dir_, which does an _ls -s_. If

```
dir ~bill
```

is entered, csh translates this to

```
ls -s /z/bill
```

Thus, the _alias_ mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases that contain multiple commands or pipelines that show where the arguments to the original command are to be substituted, using the facilities of the _history_ mechanism. The definition

```
alias cd 'cd \!* ; ls'
```

does an _ls_ command after each change directory (_cd_) command. The entire alias definition is enclosed in single quote characters to prevent most substitutions from occuring and the semicolon (;) character from being recognized as a parser metacharacter. The exclamation mark (!) here is

escaped with a backslash (\) to prevent it from being interpreted when the _alias_ command is typed in. The \!* substitutes the entire argument list to the pre-aliasing _cd_ command, without giving an error if there are no arguments. The ; that separates commands indicates that one command is to be executed before the next is executed. Similarly, the definition

```
alias whois 'grep \!↑ /etc/passwd'
```

defines a command that looks up its first argument in the password file.

2.5 Detached Commands and Redirection

The ampersand (&) metacharacter can be placed after a command, or after a sequence of commands separated by ; or |. This prevents csh from waiting for the commands to terminate before prompting again. These commands are said to be _detached_ or _background_ processes. In the following example,

```
% pr ~bill/csh/sh.c | lpr &
5120
5121
%
```

Csh prints two process numbers and comes back very quickly rather than waiting for the _pr_ and _lpr_ commands to finish. The numbers 5120 and 5121 are the process numbers assigned by the system to the _pr_ and _lpr_ commands.

Running commands in the background tends to slow down the system and is not a good idea if the system is overloaded. When overloaded, the system has a slower user response when a large number of processes are run at once.

Severe complications can be expected if a command run in the background is read from the user's terminal at the same time as csh reads a command run from the terminal. To avoid this problem, the default standard input for a command run in the background is not the terminal but an empty file called /dev/null. Commands run in the background are also unaffected by interrupt and quit signals generated at the terminal. (If a background command stops suddenly when INTERRUPT or QUIT is pressed, a bug probably exists in the background program.)

If it is necessary to log off the system before the command completes, the command must be run immune to hangup signals. This is done by placing the word _nohup_ before each program in the command. For example

```
nohup man csh | nohup lpr &
```

In addition to the standard output, commands also have a diagnostic output that is normally directed to the terminal even when the standard output is directed to a file or a pipe. It is occasionally desirable to redirect the diagnostic output along with the standard output. For instance, if the output of a long running command is to be redirected into a file, and it would be helpful to have a record of any error diagnostic it produces, enter

```
command >& file
```

The >& tells csh to route both the diagnostic output and the standard output into *file*. Similarly, the following command

```
command |& lpr
```

can be used to route both standard and diagnostic output through the pipe to the line printer *lpr*. In this example, a command of the form

```
command >&! file
```

is used when *noclobber* is set and *file* already exists.

Finally, it is possible to use the form

```
command >> file
```

to place output at the end of an existing file.

If *noclobber* is set, an error results if *file* does not exist; otherwise, csh creates *file* if it does not exist. To eliminate the error condition if *file* does not exist when *noclobber* is used, enter the following:

```
command >>! file
```

2.6 Built-In Commands

The *alias* command described in Section 2.4 assigns new aliases and displays the existing aliases. With no arguments, it prints the current aliases. It can also be given an argument, such as

```
alias ls
```

to show the current alias for *ls*, for example.

The *cd* and *chdir* commands, which are equivalent, change the working directory of csh. It is useful to make a directory for each project being worked on, and to place all files related to that project in that directory. For example, the following commands can be used to enter the directory *newpaper*:

```
% pwd
/z/bill
% mkdir newpaper
% chdir newpaper
% pwd
/z/bill/newpaper
%
```

A group of related files can be placed there. The print working directory (*pwd*) command shows the name of the current directory, which is usually a subdirectory of the home directory. It is possible to return to the home login directory by entering

```
chdir
```

with no arguments.

The *echo* command prints its arguments. It is often used in csh scripts or as an interactive command to see what file name expansions yield.

The *history* command shows the contents of the history list. The numbers given with the history events can be used to reference previous events that are difficult to reference using contextual mechanisms. If a ! character is placed in the value of the shell variable *prompt*, the shell substitutes the index of the current command in the history list. This number can be used to refer to this command in a history substitution. Thus, it is possible to use the command

```
set prompt='\!%'
```

Note that the ! character has to be escaped, even here, within single quote characters.

The *logout* command can be used to terminate a login shell that has *ignoreeof* set.

The *repeat* command can be used to repeat a command several times. Thus, to make five copies of the file *one* in the file *five*, enter

```
repeat 5 cat one >> five
```

The *setenv* command can be used to set variables in the environment. For example,

```
setenv TERM adm3a
```

sets the value of the environment variable TERM to adm3a. The user program *printenv* prints out the environment, as follows:

```
% printenv
HOME /z/bill
SHELL    /bin/csh
TERM adm3a
%
```

The *source* command can be used to force the current shell to read commands from a file. For example,

```
source .cshrc
```

can be used after a change is made to the .*cshrc* file that is to take effect before the next time the user logs in.

The *time* command causes a command to be timed, no matter how much CPU time it takes. For example,

```
% time cp five five.save
0.0u 0.3s 0:01 26%
% time wc five.save
   1200    6300   37650 five.save
1.2u 0.5s 0:03 55%
%
```

indicates that the *cp* command used less that a tenth of a second of user time and only three-tenths of a second of system time in copying the file *five* to *five.save*The command word count (*wc*) on the other hand, used 1.2 seconds of user time and 0.5 seconds of system time in three seconds of elapsed time in counting the number of words, characters, and lines in *five.save*The percentage (55%) indicates that over this period of three seconds, the command *wc* used an average of 55 percent of the available CPU cycles of the machine. (This is a very high percentage and indicates that the system is lightly loaded.)

The *unalias* and *unset* commands can be used to remove aliases and variable definitions from the shell.

The *wait* command can be used after starting processes with & to quickly see if they have finished. If the shell responds immediately with another prompt, the commands have finished executing. Otherwise, it is necessary to wait for the shell

to prompt, or interrupt the shell by sending a RUB or DELETE character. If the shell is interrupted, it prints the names and numbers of the unfinished processes. An example of the response to a *wait* command follows.

```
% nroff paper | lpr &
2450
2451
% wait
  2451  lpr
  2450  nroff
wait: Interrupted.
%
```

If it is necessary to stop running a background process, the *kill* program must be used. The process number to be killed must be entered. For example, to stop *nroff* in the pipeline example, enter

```
% kill 2450
% wait
2450: nroff: Terminated.
%
```

Here the shell displayed a diagnostic indicating that the user terminated *nroff*, only after a *wait* command was done.

SECTION 3

CSH CONTROL STRUCTURES AND COMMAND SCRIPTS

3.1 Introduction

It is possible to place commands in files called shell scripts and to invoke shells to read and execute commands from these files. Those features of csh useful to the writers of such scripts are detailed in this section.

3.2 Invocation and the argv Variable

A csh command script can be interpreted by entering

```
csh script ...
```

where _script_ is the name of the file containing a group of csh commands and ... is replaced by a sequence of arguments. Csh places these arguments in the variable _argv_ and then begins to read commands from the script. These parameters are then available through the same mechanisms used to reference any other csh variables.

If the file _script_ is made executable by entering

```
chmod 755 script
```

and placing a shell comment at the beginning of the shell script, the command /bin/csh is automatically invoked to execute _script_ when

```
script
```

is entered. If the first character of the first line is not a #, csh invokes /bin/sh to interpret the command script.

3.3 Variable Substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed, a mechanism known as _variable substitution_ is applied to these words. Keyed by the dollar sign ($) character, this substitution replaces the names of variables with their values. For example,

```
echo $argv
```

when placed in a command script, causes the current value of the variable _argv_ to be echoed to the output of the shell script. It is an error for _argv_ to be _unset_ at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to 1 if _name_ is set, or to 0 if _name_ is not set. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable _name_, as in the following example:

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv
%
```

It is also possible to access the components of a variable that has several values. For example,

```
$argv[1]
```

gives the first component of _argv_ or, in the preceding example, _a_. Similarly,

```
$argv[$#argv]
```

gives _c_, and

```
$argv[1-2]
```

gives _ab_.

Other notations useful in csh scripts are

$_n_

where n is an integer as a shorthand for $argv[n], the nth parameter, and

$*

is shorthand for $argv. The form

$$

expands to the process number of the current shell. Since this process number is unique in the system, it can be used in the generation of unique temporary file names.

One minor difference between $n and $argv[n] is that the form $argv[n] yields an error if n is not in the range 1-$#argv, while $n never yields an out-of-range subscript error.

It is never an error to give a subrange of the form n-; if there are less than n components of the given variable, no words are substituted. A range of the form m-n likewise returns an empty vector without giving an error when m exceeds the number of elements of the given variable, provided the subscript n is in range.

3.4 Expressions

To construct csh scripts, it is necessary to evaluate expressions in the shell based on the values of variables. All the arithmetic operations of the language C are available in csh with the same precedence that they have in C. The operations == and != compare strings, and the operators && and || implement the boolean and/or operations.

Csh also allows file enquiries of the form

-? filename

where ? is replaced by any of a number of single characters. For instance, the expression primitive

-e filename

tells whether the file filename exists. Other primitives test for read, write, and execute access to the file, and test whether it is a directory, or whether it has nonzero length.

It is possible to test whether a command terminates normally by using a primitive of the form { command }, which returns 1 if the command exits normally with exit status zero, or 0 if the command terminates abnormally or with exit status nonzero. If more detailed information about the execution status of a command is required, it can be executed and the variable $status examined in the next command. Since $status is set by every command, it is very transient. However, it can be saved if it is more convenient to use it more than once.

For a full list of expression components available, see *csh*(*1*) in the *ZEUS Reference Manual*.

3.5 Sample Csh Script

A sample shell script that uses the expression mechanism of csh and some of its control structure follows.

```
cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

   if ($i:r.c != $i) continue
   # not a .c file so do nothing

   if (! -r ~/backup/$i:t) then
      echo $i:t not in backup... not cp\'ed
      continue
   endif

   cmp -s $i ~/backup/$i:t
   # to set $status

   if ($status != 0) then
      echo new backup of $i
      cp $i ~/backup/$i:t
   endif
end
```

This script uses the _foreach_ command, which causes csh to execute the commands between the _foreach_ and the matching _end_ for each of the values given between (and); the named variable (in this case, i) is set to successive values in the list). Within this loop, it is possible to use the command _break_ to stop execution of the loop, and _continue_ to prematurely terminate one iteration and begin the next. After the _foreach_ loop, the iteration variable retains the value at the last iteration.

The variable _noglob_ is set to prevent file name expansion of the members of _argv_. This is advisable if the arguments to a shell script are file names that have already been expanded or if the arguments can contain file name expansion metacharacters. It is also possible to quote each use of a $ variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form

```
if ( expression ) then
     command
     ...
endif
```

The placement of the keywords here is not flexible due to

the current implementation of csh.

Another form of the if statement

```
if ( expression ) command
```

can be written as follows:

```
if ( expression ) \
    command
```

Here the new line has been escaped for the sake of appearance, and the \ must immediately precede the end-of-line. The command must not involve |, &, or ; and must not be another control command.

The more general _if_ statements in the previous examples can also be used with a sequence of _else-if_ pairs followed by a single _else_ and an _endif_, as in the following:

```
if ( expression ) then
    commands
else if (expression ) then
    commands
...

else
    commands
endif
```

Another important mechanism used in csh scripts is colon (:) modifiers. The modifier :r can be used to extract a root of a file name. If the variable _i_ has the value foo.bar, the following example

```
% echo $i $i:r
foo.bar foo
%
```

shows how the :r modifier strips off the trailing _.bar_. Other modifiers take off the last component of a path name, leaving the head (:h) or all but the last component of a path name leaving the tail (:t). These modifiers are fully described in _csh_(_1_) of the _ZEUS Reference Manual_. It is also possible to use the _command substitution_ mechanism described in Section 5 to perform modifications on strings and then reenter the csh environment. Since each usage of this mechanism involves the creation of a new process, it is more expensive to use than the : modification mechanism.

It is also important to be aware that the current implementation of csh limits the number of : modifiers on a $

substitution to 1. Thus,

```
% echo $i $i:h:t
/a/b/c /a/b:t
%
```

does not do what one would expect.

3.6 Other Control Structures

Csh has control structures _while_ and _switch_, similar to those of C. These take the forms

```
while ( expression )
     commands
end
```

and

```
switch ( word )
case str1:
     commands
     breaksw
  ...

case strn:
     commands
     breaksw

default:
     commands
     breaksw

endsw
```

For details, see _csh_(_1_). C programmers should be aware that _breaksw_ is used to exit from a _switch_, while _break_ exits a _while_ or _foreach_ loop. A common mistake to make in csh scripts is to use _break_ rather than _breaksw_ in switches.

Finally, csh allows a _goto_ statement with labels similar to those in C. For example:

```
loop:
     commands
     goto loop
```

3.7 Supplying Input to Commands

Commands run from csh scripts receive, by default, the standard input of the shell that is running the script. This allows csh scripts to make full use of pipelines, but requires extra notation for commands that are to take inline data.

Thus, a metanotation for supplying inline data to commands in csh scripts is needed. For example, the following script runs the editor to delete leading blanks from the lines in each argument file.

```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/↑[ ]*//
w
q
'EOF'
end
%
```

The notation << 'EOF' means that the standard input for the _ed_ command is to come from the text in the csh script file, up to the next line consisting of 'EOF'. The fact that the EOF is enclosed in single quote (') characters prevents csh from performing variable substitution on the intervening lines. In general, if any part of the word following the << (which the csh uses to terminate the text to be given to the command) is quoted, these substitutions are performed. In this case, since the form 1,$ was used in the editor script, it is necessary to ensure that this $ is not variable substituted. It is also possible to ensure this by preceding the $ with a \. For example:

```
1,\$s/↑[ ]*//
```

However, quoting the EOF terminator is a more reliable way of achieving the same thing.

3.8 Catching Interrupts

If the csh script creates temporary files, it is helpful to catch interruptions of the csh script so that these files can be cleaned up. This can be done with

```
onintr label
```

where *label* is a label in the program. If an interrupt is received, the shell does a *goto* label. It is possible to remove the temporary files and then do an *exit* command to exit from the csh script. To exit with a nonzero status, enter

```
exit(1)
```

This exits with status 1.

3.9 Other Functions

There are other features of csh useful to writers of csh procedures. The *verbose* and *expand* variables and the related *-v* and *-x* command line options can be used to help trace the actions of csh. The *-n* option causes csh commands to be read, but not executed.

It is important to note that csh only executes scripts that begin with the character # (that is, shell scripts that begin with a comment). Similarly, the /bin/sh on the system defers to csh to interpret shell scripts that begin with #. This allows scripts for both shells to coexist without complications.

Another quotation mechanism uses double quotes ("), allowing only some of the expansion mechanisms discussed so far to occur on the quoted string, making the string into a single word as ' does.

3.10 Make

Do not attempt to use shell scripts to perform taks that can be handled by *make* (see *make* manual). The *make* program maintains a group of related files or performs sets of operations on related files. For instance, a large program consisting of one or more files can have its dependencies described in a *makefile*, which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily placed in this *makefile*. Using this format is preferable to maintaining a group of shell procedures to maintain these files.

A *makefile* can be used for applications other than programs. For example, a *makefile* can be created to define how different versions of a document are to be created and which options of *nroff* or *troff* are appropriate.

SECTION 4

MISCELLANEOUS SHELL MECHANISMS

4.1 Loops at the Terminal

The _foreach_ control structure can be used at the terminal to aid in performing a number of similar commands. For instance, suppose there were three shells in use on ZEUS, /bin/sh, /bin/nsh, and /bin/csh. To count the number of people using each shell, issue the commands

```
% grep -c nsh$ /etc/passwd
27
% grep -c csh$ /etc/passwd
34
% grep -c -v sh$ /etc/passwd
6
%
```

A simple method of requesting this information is:

```
% foreach i (`nsh$` `csh$` `-v sh$`)
? grep -c $i /etc/passwd
? end
27
34
6
%
```

The shell prompts for input with ? when reading the body of the loop.

Variables that contain lists of file names or other words are useful with loops. For example,

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The *set* command gives, as the variable *a*, a list of all the file names in the current directory as value. It is then possible to iterate these names to perform any chosen function.

The output of a command within single back quote (`) characters is converted by csh to a list of words. The quoted string can also be placed within double quote (") characters to take each nonempty line as a component of the variable, preventing the lines from being split into words at blanks and tabs. The modifier :x can be used later to expand each component of the variable into another variable, splitting it into separate words at embedded blanks and tabs.

4.2 Braces in Argument Expansion

Another form of file name expansion involves braces ({ and }), which specify that the contained strings, separated by a comma (,), are to be consecutively substituted into the containing characters, and the results are to be expanded left to right. For example,

```
A{str1,str2,...strn}B
```

expands to

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other file name expansions, and can be nested. The results of each expanded string are sorted separately, left to right. The resulting file names are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments that are not file names, but that have common parts.

For example,

```
mkdir ~/{hdrs,retrofit,csh}
```

can be used to make subdirectories *hdrs*, *retrofit*, and *csh* in the user's home directory. This mechanism is useful when the common prefix is longer than in the above example, such as in the following example:

```
chown bin /usr/{bin/{ex,edit},lib/{ex1.1strings,how_ex}}
```

4.3 Command Substitution

A command enclosed in single back quote characters is replaced, just before file names are expanded, by the output from that command. Thus, it is possible to enter

```
set pwd=`pwd`
```

to save the current directory in the variable _pwd_, or to enter

```
ex `grep -l TRACE *.c`
```

to run the editor _ex_, suppling as arguments those file names ending in .c that have the string _TRACE_ in them. (Command expansion also occurs in input redirected with << and within double quotations. Refer to _csh_(_1_) for more details.)

APPENDIX A

SPECIAL CHARACTERS

The following lists the special characters of csh and the ZEUS system. A number of these characters also have special meaning shown in expressions. See csh(1) in the ZEUS Reference Manual for a complete list of characters.

`|` separates commands in a pipeline; the output of one command in a pipeline is the input to the succeeding command

`;` separates commands to be executed sequentially

`&` follows commands to be executed without waiting for completion

`( )` brackets expressions and variable values; any of the preceding commands can be placed inside brackets to form a command that in turn can be part of a larger string

`&&` indicates a pipeline in which the second command is executed only if the first command succeeds

`||` indicates a pipeline in which the second command is executed only if the first command fails

`<` indicates redirected input

`>` indicates redirected output

`<<` reads shell input up to string matching the following argument

`>>` writes output at end of argument file

`'` prevents metameaning of a group of characters

`"` similar to ', but allows variable and command expansion

`\` prevents special meaning of following single character

`\newline` expands to an embedded new line if within a quoted string; expands to a blank, otherwise

#	begins a shell comment
/	separates components of a file's path name
?	expansion character matching any single character
*	expansion character matching any sequence of characters
[ ]	expansion sequence matching any single character from a set
~	used at the beginning of a file name to indicate home directories
{ }	used to specify groups of arguments with common parts
$	indicates variable substitution
!	indicates history substitution
:	precedes substitution modifiers
↑	used in special forms of history substitution
`	indicates command substitution
-	prefixes option (flag) arguments to commands

APPENDIX B

GLOSSARY

The most important terms introduced in this document are listed in this Appendix. References of the form (2.5) indicate that more information can be found in Section 2.5 of this document. References of the form _pr_(_1_) indicate that the command _pr_ is in Section 1 of the _ZEUS Reference Manual_. To get an on-line copy of the manual page, enter

```
man 1 pr
```

. The user's current directory has the name . as well as the name printed by the command _pwd_. The current directory (.) is usually the first component of the search path contained in the variable _path_. Thus, commands that are in . are found first (2.2). The period character is also used to separate components of file names (1.6). The character . at the beginning of a component of a path name is treated specially and is not matched by the file name expansion metacharacters ?, *, and [ ] pairs (1.6).

.. Each directory has a file .. in it, which is a reference to its parent directory. After changing directories with _chdir_, for example,

```
chdir paper
```

it is possible to return to the parent directory by entering

```
chdir ..
```

The current directory is printed by _pwd_ (2.6).

alias An _alias_ specifies a shorter or different name for a ZEUS command, or a transformation on a command to be performed in the shell. The shell command _alias_ establishes aliases and can print their current values. The command _unalias_ is used to remove aliases (2.6).

argument Commands in ZEUS receive a list of argument words. Thus, the command

echo a b c

consists of a command name *echo* and three argument words a, b, and c (1.1).

argv — The list of arguments to a command written in a shell script or shell procedure is stored in a variable called *argv* within the shell. This name is taken from the conventional name in the C programming language (3.4).

background — A *background* command is a command that runs while the shell executes other commands. (2.5).

bin — A directory containing binaries of programs and shell scripts to be executed is typically called a *bin* directory. The standard system *bin* directories are /bin, which contains the most heavily used commands, and /usr/bin, which contains most of the other user programs. Binaries can be placed in any directory. The name of the directories should be a component of the variable *path* if the binaries are to be executed often.

break — *Break* is a built-in command used to exit from loops within the control structure of the shell (3.6).

builtin — A command executed directly by the shell is called a *builtin* command. Most commands in ZEUS are not built into the shell, but exist as files in *bin* directories. These commands are accessible because the directories in which they reside are named in the *path* variable.

case — A *case* command is used as a label in a *switch* statement in the shell's control structure, similar to that of the language C *csh*(*1*) (3.7).

cat — The *cat* program catenates a list of specified files on the standard output. It is usually used to look at the contents of a single file on the terminal (1.7, 2.3).

cd — The *cd* command changes the working directory. With no arguments, *cd* changes the user's working directory to be the user's *home*

directory (2.3, 2.6).

chdir
: The _chdir_ command is a synonym for _cd_, which is usually used because it is easier to type.

chsh
: The _chsh_ command is used to change the shell that is used on ZEUS. By default, the user uses csh, which resides in /bin/csh.

cmp
: _cmp_ is a program that compares files. It is usually used on binary files, or to see if two files are identical (3.5). For comparing text files, use the program _diff_, described in _diff_(_1_).

command
: A function performed by the system, either by the shell or by a program residing in a file in the ZEUS system, is called a _command_ (2.1).

command substitution
: The replacement of a command enclosed in single back quote (`) characters by the text output by that command is referred to as _command substitution_ (3.7, 4.2).

component
: A part of a _path name_ between slash (/) characters is called a _component_ of that path name. A _variable_ that has multiple strings as its value is said to have several _components_; each string is a _component_ of the variable.

continue
: A built-in command that causes execution of the enclosing _foreach_ or _while_ loop to cycle prematurely. Similar to the _continue_ command in the C programming language (3.5).

core dump
: When a program terminates abnormally, the system places an image of its current state in a file named _core_. The _core dump_ can be examined with the system debuggers _adb_(_1_) and _zdb_(_1_) to determine what went wrong with the program (1.7). If, for a system program, the shell produces a message of the form:

commandname: Segmentation violation--Core dumped

(where "Segmentation violation" is only one of several possible messages), report this with the _str_ command _str_(_1_).

cp — The copy (*cp*) program copies the contents of one file into another file (1.6, 2.6).

.cshrc — The file .*cshrc* in the *home* directory is read by each shell as it begins execution. It is usually used to change the setting of the variable *path* and to set *alias* parameters that are to take effect globally (2.1).

date — The *date* command prints the current date and time (1.3).

debugging — Debugging is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables that can be used to aid in shell debugging.

default — The label _default: is used within shell *switch* statements to label the code to be executed if none of the *case* labels matches the value switched (3.6, 3.8).

DELETE — The *DELETE* or *RUBOUT* key on the terminal is used to generate a ZEUS interrupt signal that stops the execution of most programs (2.6).

detached — A command that runs while the shell is executing other commands is referred to as *detached* (2.5).

diagnostic — An error message produced by a program is often referred to as a *diagnostic*. Most error messages are not written to the standard output, since that is often directed away from the terminal (1.3, 1.5). Instead, error messsages are written to the *diagnostic output*, which usually appears on the terminal (2.5).

directory — A structure that contains files is called a *directory*. The directory in which the user first logs in is the *home* directory (1.6).

echo — The *echo* command prints arguments to the command in effect (1.6, 3.5, 3.9).

else — The *else* command is part of the "if-then-else-endif" control command construct (3.5).

EOF — An *end-of-file* is generated whenever a command reads to the end of a file that it has been given as input. It can also be

generated at the terminal with a control-d. Commands receiving input from a _pipe_ receive an EOF when the command sending them input completes. Most commands terminate when they receive an EOF. The shell has an option to ignore EOF from a terminal input, which makes it possible to avoid logging out accidentally by typing too many control-d's (1.1, 1.8, 3.7).

escape — A backward slash (\) character used to prevent the special meaning of a metacharacter is said to _escape_ the character from its special meaning. Thus,

```
echo \*
```

echoes the character *, while

```
echo *
```

echoes the names of the file in the current directory. In this example, \ escapes * (1.5).

/etc/passwd — This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by : characters (2.3). This file can be examined by entering

```
cat /etc/passwd
```

The command _grep_ is often used to search for information in the file. See _passwd_(_5_) and _grep_(_1_) for more details.

exit — The _exit_ command, which is built into the shell, is used to force termination of a shell script (3.8).

exit status — A command that uncovers a problem can reflect this problem back to the command that invoked it by returning a nonzero number as its _exit status_ (a status of zero being considered normal termination). The _exit_ command can be used to force a shell command script to give a nonzero _exit status_ (3.4).

expansion — Replacing shell input strings that contain metacharacters with other strings is referred to as the process of _expansion_. For example,

replacing the word * with a sorted list of files in the current directory is a file name expansion. Replacing the characters !! with the text of the last command is a history expansion. Expansions are also referred to as _substitutions_ (1.6, 3.3, 4.2).

expressions
: Expressions are used in csh to control the conditional structures used in writing shell scripts and in calculating values for these scripts. The operators available in csh expressions are those of the C language (3.4).

extension
: File names often consist of a _root_ name and an _extension_, separated by the period character (.). By convention, groups of related files often share the same root name. Extensions are added to differentiate among files within the group. Thus, if prog.c is a C program, the object file for this program would be stored in prog.o. Similarly, a paper written with the -ms nroff macro package might be stored in paper.ms, while a formatted version of this paper might be kept in paper.out and a list of spelling errors in paper.errs (1.6).

file name
: Each file in ZEUS has a name consisting of up to 14 characters, not including the slash character (/), which is used in path name building. Most file names do not begin with the period character. They contain only letters and digits, with perhaps a period separating the root portion of the file name from an extension (1.6).

file name expansion
: File name expansion uses the metacharacters *, ?, and [and] to provide a convenient mechanism for naming files. Using file name expansion makes it easy to name all the files in the current directory, or all files that have a common root name. Other file name expansion mechanisms use the metacharacter ~ and allow files in other users' directories to be named easily (1.6, 4.2).

flag
: Many ZEUS commands accept arguments that are not the names of files or other users, but are used to modify the action of the commands. These are referred to as _flag_ options

and, by convention, consist of one or more letters preceded by the hyphen (-) character (1.2). For example, the *ls* list file command has an option *-s* to list the sizes of files. This is specified

```
ls -s
```

foreach — The *foreach* command is used in csh scripts and at the terminal to specify repetition of a sequence of commands while the value of a given csh variable falls within a specified range (3.5, 4.1).

getty — The *getty* program determines the speed at which the terminal is to run when the user first logs in. It displays the initial system banner and *login*.

goto — The csh command *goto* is used in csh scripts to transfer control to a given label (3.6).

grep — The *grep* command searches through a list of argument files for a specified string. For example,

```
grep bill /etc/passwd
```

prints each line in the file /*etc*/*passwd* that contains the string *bill*. Actually, *grep* scans for regular expressions in the sense of the editors *ed*(*1*) and *ex*(*1*). *grep* stands for "globally find regular expression and print."

hangup — When a user hangs up a phone line, a *hangup* signal is sent to all running processes on the user's terminal, causing them to terminate execution prematurely. To allow commands to continue running after logging off a dialup, use the command *nohup* (2.5).

head — The *head* command prints the first few lines of one or more files. Run the *head* program with a group of file names as arguments to get a general idea of the contents of the files (1.5, 2.3).

history — The *history* mechanism of csh allows previous commands to be repeated. Csh has a *history list* where these commands are kept, and a *history* variable that controls how large this list is (1.7, 2.5).

home directory Each user has a home directory, that is given in the password file /*etc*/*passwd*. The user is placed in the home directory when first logging in. The *cd* or *chdir* command with no arguments returns the user to this directory. The name of this directory is recorded in the shell variable *home*.

if The *if* command is a conditional command used in csh command scripts to determine what course of action to take next (3.5).

ignoreeof Normally, the user's shell exits, printing logout if the user types a control-d at a % prompt. This is the usual way to log off the system. The user can *set* the *ignoreeof* variable in the .*login* file, and then use the command *logout* to log out. This is useful to avoid accidentally logging off by typing too many control-d characters. (2.2, 2.6).

input Information taken from the terminal or from files is called *input*. Commands normally read input from their *standard input* which is, by default, the terminal. The metacharacter followed by a file name can be used to cause input to be read from a file. Many commands also read from a file specified as an argument. Commands placed in pipelines are read from the output of the previous command in the pipeline. The leftmost command in a pipeline reads from the terminal if its input is not redirected and if a file name is not given to use as standard input. Special mechanisms exist for supplying input to commands in csh scripts (1.1, 1.5, 3.7).

interrupt An *interrupt* is a signal that causes most programs to stop execution. It is generated by pressing the RUB or DEL key. Certain programs such as csh and the editors handle an interrupt in special ways, usually by stopping what they are doing and prompting for another command. While csh is executing another command and waiting for it to finish, csh does not respond to interrupts. (1.7, 2.6, 3.8).

kill The *kill* program terminates processes that are run with the & option (2.6).

.login The file .*login* in the user's *home* directory

is read by csh each time the user logs in to ZEUS; the commands there are executed (2.1).

logout
: The _logout_ command causes a login shell to exit. Normally, a login shell exits when control-d is pressed, generating an EOF. If _ignoreeof_ has been set in the ._login_ file, control-d does not work, and it is necessary to use the command _logout_ to log off the ZEUS system (2.2).

.logout
: When a user logs off of ZEUS, the shell prints _logout_ and executes commands from the file ._logout_ in the user's _home_ directory.

lpr
: The command _lpr_ is the line printer command. The standard input of _lpr_ is spooled and printed on the ZEUS line printer. It is possible to give _lpr_ a list of file names as arguments to be printed. It is common to use _lpr_ as the last component of a _pipeline_ (2.3).

ls
: The list file (_ls_) command is one of the most commonly used ZEUS commands. With no argument file names, it displays the names of the files in the current directory. It has a number of useful _flag_ arguments. It can also be given the names of directories as arguments, in which case it lists the names of the files in these directories (1.2).

mail
: The _mail_ program is used to send and receive messages from other ZEUS users (1.1, 2.2).

make
: The _make_ command is used to maintain one or more related files and to organize functions to be performed on these files. Its primary use is maintaining a single program consisting of several source files. In many ways, _make_ is easier to use, and more helpful, than shell command scripts (3.10).

makefile
: The file containing the commands for _make_ is called _makefile_ (3.18).

metacharacter
: Many characters that are neither letters nor digits have special meaning, either to the shell or to ZEUS. These characters are called _metacharacters_. It is necessary to enclose these characters in quotes if they are used in arguments to commands and no

special meaning is required. An example of a metacharacter is the character >, which is used to indicate placement of output into a file. For the purposes of the *history* mechanism, most unquoted metacharacters form separate words (1.4). Appendix A of this document lists the metacharacters.

mkdir — The *mkdir* command is used to create a new directory (2.6).

modifier — A modifier is a part of a command line that changes the way the original command is interpreted. Substitutions, with the history mechanism (keyed by the character !), or of variables using the metacharacter $, are often subjected to modifications, which are indicated by placing the character : after the substitution and following this with the modifier itself (3.5).

noclobber — The csh variable *noclobber* can be set in the file .*login* to prevent accidental destruction of files by the > output redirection metasyntax of the shell (2.2, 2.5).

nohup — The shell *nohup* command is used to run background commands to completion even if the user logs off before these commands complete (2.5).

nroff — The standard text formatter on ZEUS is the program *nroff*. Using *nroff* and one of the available *macro* packages for it, it is possible to have documents automatically formatted and to prepare them for phototypesetting using the typesetter program *troff* (3.10).

onintr — The *onintr* command is built into chs and is used to control the action of a shell command script when an interrupt signal is received (3.8).

output — Many commands in ZEUS produce data that is called *output*. This output is usually placed on what is known as the *standard output*, which is normally connected to the user's terminal. The shell has a syntax using the metacharacter > for redirecting the standard output of a command to a file (1.3). Using the *pipe* mechanism and the metacharacter |, it is also possible for the standard output

of one command to become the standard input of another command (1.5). Some commands do not direct their output to the standard output. The line printer command (*lpr*), for example, diverts its output to the line printer (2.3). The *write* command places its output on another user's terminal (2.3). Commands also have a *diagnostic output* where they write their error messages. Normally, these go to the terminal even if the standard output has been sent to a file or another command. However, it is possible to direct error diagnostics along with standard output using a special metanotation (2.5).

path

The csh variable *path* gives the names of the directories in which it searches for the commands it is given. It always checks first to see if the named command is built into the shell. If it is, it does not need to search for the command, as it can perform it internally. If the command is not built in, csh searches for a file with the name given in each of the directories in the *path* variable, left to right. Since the normal definition of the *path* variable is

```
path (. /bin /usr/bin)
```

Csh normally looks in the current directory, and then in the standard system directories, /bin and /usr/bin, for the named command (2.2). If the command cannot be found, csh prints an error diagnostic. Scripts of C shell commands are executed using another shell to interpret them if they have execute bits set. This is normally true because a command of the form

```
chmod 755 script
```

is executed to turn on these execute bits (3.2).

path name

A list of names, separated by slash (/) characters forms a *path name*. Each *component* between successive / characters names a directory in which the next component file resides. Path names that begin with the character / are interpreted relative to the *root* directory in the file system. Other path names are interpreted relative to the

current directory as reported by *pwd*. The last component of a path name can name a directory; however, it usually names a file.

pipeline — A group of commands that are connected together with the standard output of each connected to the standard input of the next is called a *pipeline*. The *pipe* mechanism used to connect these commands is indicated by the vertical bar (|) metacharacter (1.5, 2.3).

pr — The *pr* command prepares listings of the contents of files with headers that give the name of the file and the date and time at which the file was last modified (2.3).

printenv — The *printenv* command is used on ZEUS systems to print the current setting of variables in the environment.

process — An instance of a running program is called a *process* (2.6). The numbers used by *kill* and printed by *wait* are unique numbers generated for these processes by ZEUS. They are useful in *kill* commands, which can be used to stop background processes (2.6).

program — A *program* (usually synonymous with *command*) is a binary file or csh command script that performs a useful function.

prompt — Many programs print a *prompt* on the terminal when they expect input. For example, the editor *ex*(*1*) prints a colon (:) when it expects input. The shell prompts for input with a percent sign (%), and occasionally with a question mark (?), when reading commands from the terminal (1.1). The csh variable *prompt* can be set to a different value to change the shell's main *prompt*. This is primarily used when debugging the shell (2.6).

ps — The *ps* command shows the processes a user is currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (2.3, 2.6). Login shells (such as the csh obtained

when logging in) are shown as -.

pwd — The *pwd* command prints the full path name of the current working directory.

quit — The *quit* signal, generated by a control-\, terminates programs that are behaving abnormally. It normally produces a *core image* file (1.7).

quotation — The process that prevents metacharacters from being interpreted with special meaning, usually by using the single quote (') character in pairs or by using the backslash (\) character, is referred to as *quotation* (1.4).

redirection — The routing of input or output from or to a file is known as *redirection* of input or output (1.3).

repeat — The *repeat* command iterates another command a specified number of times (2.6).

RUB — The RUB or DEL key generates an interrupt signal that is used to stop programs or to cause them to return and prompt for more input (2.6).

script — Sequences of csh commands placed in a file are called shell command scripts. It is often possible to perform simple tasks using these scripts without writing a program by using the shell to selectively run other programs (3.2).

set — The built-in *set* command assigns new values to shell variables and displays the values of the current variables. Many csh variables have special meaning to csh itself (2.1).

setenv — On ZEUS systems, variables in the environment *environ*(*5*) can be changed by using the *setenv* built-in command (2.6). The *printenv* command can be used to print the value of the variables in the environment.

shell — A shell is a command language interpreter. It is possible for users to write and run their own shells, as shells are no different from any other programs in terms of system response. This document deals with the details of one particular shell, called *csh*.

shell script — See _script_ (3.2).

sort — The _sort_ program sorts a sequence of lines in ways that can be controlled by argument flags (1.5).

source — The _source_ command causes csh to read commands from a specified file. It is useful for reading files such as ._cshrc_ after changing them (2.6).

special character
: See *metacharacters* and Appendix A of this document.

standard
: The *standard input* and *standard output* of commands are often referred to. See *input* and *output* (1.3, 3.7).

status
: A command normally returns a *status* when it finishes. By convention, a *status* of zero indicates that the command succeeded. Commands can return nonzero status to indicate that some abnormal event has occurred. The csh variable *status* is set to the status returned by the last command. It is most useful in shell commmand scripts (3.4, 3.5).

substitution
: Csh implements several *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Examples of this are *history* substitution keyed by the metacharacter !, and *variable* substitution indicated by $. Substitutions are also referred to as *expansions* (3.3).

switch
: The *switch* command of csh allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the *switch* statement in the C language (3.6).

termination
: When a command being executed finishes, it is said to *terminate*. Commands normally terminate when they read an EOF from their standard input. It is also possible to terminate commands by sending them an interrupt or quit signal (1.7). The *kill* program terminates commands specified by their process numbers (2.6).

then
: The *then* command is part of csh's if-then-else-endif control construct used in command scripts (3.5)

time
: The *time* command measures the amount of CPU and real time consumed by a specified command (2.1, 2.6).

troff
: The *troff* program is used to typeset documents. See also *nroff* (3.10).

unalias
: The *unalias* command removes aliases (2.6).

unset — The *unset* command removes the definitions of csh variables (2.2, 2.6).

variable expansion — See *variables* and *expansion* (2.2, 3.3).

variables — Variables in csh hold one or more strings as value. The most common use of variables is in controlling the behavior of the shell. See *path*, *noclobber*, and *ignoreeof* for examples. Variables such as *argv* are also used in writing csh command scripts (2.2).

verbose — The *verbose* csh variable causes commands to be echoed after they are history expanded. This is often useful in debugging csh scripts. The *verbose* variable is set by the shell's command line option (3.9).

wait — The built-in command *wait* causes csh to pause, and not prompt, until all commands run in the background have terminated (2.6).

where — The *where* command shows where the users named as arguments are logged in to the system (2.3).

while — The *while* built-in control construct is used in csh command scripts (3.6).

word — A group of characters that forms an argument to a command is called a *word*. Many characters that are neither letters, digits, -, ., or / form words by themselves, even if they are not surrounded by blanks. Any sequence of characters can be made into a word by surrounding it with single quote (') characters, except for the single quote character itself and !, which require special treatment (1.1, 1.5).

working directory — Any directory a user is currently working in is called a *working directory*. This directory name is printed by the *pwd* command, and the files listed by *ls* are the ones in this directory. The user can change working directories using the *chdir* or *cd* command.

write — The *writer* command is used to communicate with other users who are logged in to ZEUS (2.3).

ZEUS

ZEUS is the operating system on which csh runs. ZEUS provides facilities that allow csh to invoke other programs, such as editors and text formatters.

THE ZEUS LINE-ORIENTED TEXT EDITOR, *ed**

* This information is based on articles originally written by Brian W. Kernighan, Bell Laboratories.

PREFACE

Although most text manipulation on the ZEUS Operating System is done with the screen-oriented editor, _vi_, some special circumstances warrent the use of the line editor, _ed_. This document is a tutorial guide to help beginners get started with _ed_ and to introduce experienced users to its more complex options.

Sections 1-12 are oriented mostly for beginners. These sections cover basic commands or basic uses of more complex commands. When a subsection of a command is for experienced users, it is labeled as such. Beginners should be aware that more information is presented in these subsections than they need for basic tasks and that concepts are used in these explanations that have not yet been introduced in the regular text. Sections 13-23 offer experienced users more complex commands and describe ways that commands act on each other. Basic commands are summarized in the Appendix.

The recommended way for both beginners and experienced users to learn _ed_ is to read this document, simultaneously using _ed_ to follow the examples, then to read the description in Section 1 of the _ZEUS Reference Manual_. Experiment with _ed_. The only sure way of seeing how a command works is to try it. The exercises cover material not completely discussed in the text. A _learn_(1) script, %_learn editor_, is also available for _ed_.

The end-of-line character varies between terminals. This character is the RETURN key on most terminals, and is referred to in this text as RETURN.

This document is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that _ed_ offers. Also, there is not enough space to explain basic ZEUS procedures; read _ZEUS for Beginners_ to learn how to log in to ZEUS and what a file is.

TABLE OF CONTENTS

SECTION 1 GETTING STARTED 6

SECTION 2 CREATING TEXT 7

SECTION 3 WRITING TEXT AS A FILE 8

SECTION 4 LEAVING THE EDITOR 9

SECTION 5 READING TEXT FROM A FILE WITH "e" 10

5.1 Basic Uses 10
5.2 Advanced Uses 11

SECTION 6 READING TEXT FROM A FILE WITH "r" 12

SECTION 7 PRINTING THE CONTENTS OF BUFFER 13

7.1 Print Command 13
7.2 Specific Lines 13
7.3 Current Line 14
7.4 Advanced Commands 16

SECTION 8 DELETING LINES 17

SECTION 9 MODIFYING TEXT 19

9.1 Substitute Command 19
9.2 Basic Modification 19
9.3 Advanced Modification 21

SECTION 10 CONTEXT SEARCHING 23

SECTION 11 CHANGING AND INSERTING TEXT 26

SECTION 12 MOVING TEXT 29

TABLE OF CONTENTS (continued)

SECTION 13 USING SPECIAL CHARACTERS 30

13.1 General 30
13.2 Period 30
13.3 Backslash 31
13.4 Dollar Sign 33
13.5 Circumflex 34
13.6 Asterisk 34
13.7 Brackets 36
13.8 Ampersand 37

SECTION 14 USING GLOBAL COMMANDS 39

14.1 Global g 39
14.2 Global v 39
14.3 Advanced Global Commands 39
14.4 Advanced Multiline Global Commands 41

SECTION 15 SUBSTITUTING NEW LINES 42

SECTION 16 MANIPULATING LINES 43

16.1 Join Lines 43
16.2 Rearrange Lines 43

SECTION 17 MANIPULATING ADDRESSES 45

17.1 Line Addressing 45
17.2 Address Arithmetic 45

SECTION 18 DOING REPEATED SEARCHES 47

SECTION 19 USING DEFAULT LINE REFERENCES 48

SECTION 20 USING THE SEMICOLON 51

SECTION 21 INTERRUPTING THE EDITOR 53

TABLE OF CONTENTS (continued)

SECTION 22 MANIPULATING FILES 54

22.1 General 54
22.2 Change a File Name 54
22.3 Copy a File 54
22.4 Remove a File 55
22.5 Put Two or More Files Together 55
22.6 Add Text to the End of a File 55
22.7 Insert One File into Another 56
22.8 Write Part of a File 56
22.9 Move Lines 57
22.10 Mark a Line 58
22.11 Copy Lines 58
22.12 Temporary Escape 59

SECTION 23 SUPPORTING TOOLS 60

23.1 General 60
23.2 Grep 60
23.3 Editing Scripts 61
23.4 Sed 61

APPENDIX A SUMMARY OF COMMANDS AND LINE NUMBERS 62

SECTION 1

GETTING STARTED

Ed is a line-oriented text editor--an interactive program for creating and modifying text on a line-by-line basis, using directions typed at a terminal. The text is often a document like this one, a program, or data for a program.

In *ed* terminology, the text being worked on is said to be "kept in a buffer." Think of the buffer as a work space, or as the information to be edited.

Tell *ed* what to do to the text by typing instructions called "commands." Most commands consist of a single letter that must be typed in lowercase. Type each command on a separate line. *Ed* makes no response to most commands, it simply carries them out. Enter a RETURN after every *ed* command line.

The prompt character, either a $ or a %, appears after logging into the system. Invoke *ed* by typing

    ed   (followed by a RETURN)

after the prompt. *Ed* is now waiting for commands.

SECTION 2

CREATING TEXT

When *ed* starts, it is like a blank piece of paper--there is no text or information present. Text must be supplied by typing it into *ed*, or by reading it into *ed* from a file.

The first command is *append*, written as the letter

```
a
```

by itself. It means "append (add) text lines to the buffer, as they are typed in." Appending is like writing fresh material on a piece of paper.

To enter lines of text into the buffer, type an *a* (followed by a RETURN), followed by the lines of text, like this:

```
a
Now is the time
for all good men
to come to the aid of their party.
.
```

The only way to stop appending is to type a line that contains only a period. If *ed* is not responding, it is probably because the . was omitted.

After the append command, the buffer contains the three lines

```
Now is the time
for all good men
to come to the aid of their party.
```

The *a* and . are not there because they are not text.

To add more text, issue another *a* command and continue typing.

An error in the commands typed to *ed* results in the response

```
?
```

This is a cue to look for an error.

SECTION 3

WRITING TEXT AS A FILE

To save text for later use, write the contents of the buffer into a file. Use the _write_ command

```
w
```

followed by the file name to be written on. This copies the buffer's contents into the specified file and destroys any previous information in the file. To save the text in a file named junk, for example, type

```
w junk
```

Leave a space between _w_ and the file name. _Ed_ responds by printing the number of characters it wrote out. In this case, _ed_ responds with

```
68
```

Blanks and the return character at the end of each line are included in the character count.

Writing a file makes a copy of the text. The contents of the buffer are not disturbed, so lines can be added to it. This is an important point. _Ed_ always works on the buffer copy of a file, not the file itself. No change in the contents of a file takes place until _ed_ receives a _w_ command. Writing out the text to a file from time to time as it is being created is a good idea. If the system crashes, only the text in the buffer is lost, but any text written in a file is safe.

SECTION 4

LEAVING THE EDITOR

To terminate a session with *ed*, save the text by writing it into a file, using the *w* command. Then type the command

```
q
```

which stands for *quit*. The shell responds with the prompt character $ or %. At this point, the buffer with all its text is no longer present. To protect the buffer from an accidental erasure, *ed* displays ? if it receives a quit command that was not preceded by a *w* command. At that point, either write the file or type another *q* to get out of *ed*.

Exercise

Enter *ed* and create some text using

```
a
... text ...
.
```

Write it out using *w*. Then leave *ed* with the *q* command, and print the file to see that everything worked. To print a file, type

```
pr filename
```

or

```
cat filename
```

in response to the prompt character. Try both.

SECTION 5

READING TEXT FROM A FILE WITH "e"

5.1 Basic Uses

The most common way to get text into the buffer is to read it from a file in the file system. This is done to edit text saved with the _w_ command in a previous session. The _edit_ command _e_ fetches the entire contents of a file into the buffer.

If the three lines "Now is the time ..." have been saved with a _w_ command, the _ed_ command

```
e junk
```

fetches the entire contents of the file _junk_ into the buffer, and responds

```
68
```

which is the number of characters in _junk_. Remember that if anything was already in the buffer, it is deleted first.

Using the _e_ command to read a file into the buffer eliminates the need to use a file name after a subsequent _w_ command; _ed_ retains the last file name used in an _e_ command, and _w_ writes on this file. Thus, a good way to operate is with the following set of commands:

```
ed
e file
[editing session]
w
q
```

Simply enter _w_ from time to time; the file name used at the beginning is updated with _w_.

To find out what file name _ed_ is working on, type the file command _f_. In this example, an

```
f
```

prompts _ed_ to reply

```
junk
```

5.2 Advanced Uses

The command

```
e  newfile
```

says "edit a new file called _newfile_ without leaving the editor." The _e_ command clears the buffer and reads in _newfile_. It is the same as the _q_ command followed by a reentry of _ed_ with a new file name, except that if _ed_ retained a pattern, then a command like // still works.

Entering _ed_ with the command

```
ed  file
```

has _ed_ read _file_ into the buffer and hold the name of the file. Any subsequent _e_, _r_, or _w_ commands that do not contain a file name refer to this file. Thus, the commands

```
ed  file1
 ... (editing) ...
w    (writes back in file1)
e  file2  (edit new file, without leaving editor)
 ... (editing on file2) ...
w    (writes back on file2)
```

do a series of edits on various files without leaving _ed_; it is not necessary to type the name of any file more than once.

To change the name of the hold file, use _f_ as follows:

```
ed  precious
f  junk
 ... (editing) ...
w
```

This reads the file _precious_ into the buffer, then changes the name of the hold file _junk_. The _w_ command applies the editing changes to the _junk_ file, leaving the _precious_ file untouched.

SECTION 6

READING TEXT FROM A FILE WITH "r"

To read a file into the buffer without destroying anything that is already there, use the *read* command *r*. The command

```
r junk
```

reads the file *junk* into the buffer by adding it to the end of whatever is already in the buffer. Doing a read after an edit, that is, entering

```
e junk
r junk
```

puts a duplicate copy of the text after the current copy. The buffer now contains the following six lines:

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

The *r* command displays the number of characters read in after the reading operation is complete.

Exercise

Experiment with the *e* command. Try reading and printing various files. *Ed* may respond with ?*name*, where *name* is the name of a file. This means that the file does not exist, typically because the file name is spelled wrong, or reading the file is not allowed. Try alternately reading and appending to see that they work similarly. Verify that

```
ed filename
```

is equivalent to

```
ed
e filename
```

SECTION 7

PRINTING THE CONTENTS OF THE BUFFER

7.1 Print Command

Use the print command

    p

to display the entire or partial contents of the buffer at the terminal.

7.2 Specific Lines

Specify the lines where printing is to begin and end, separated by a comma, and followed by the letter _p_. Thus, to print the first two lines of the buffer (that is, lines 1 through 2), enter

    1,2p   (starting line=1, ending line=2 p)

_Ed_ responds with

    Now is the time
    for all good men

To print all the lines in the buffer, _ed_ provides a short-hand symbol for "line number of the last line in the buffer"--the dollar sign ($). Use the command:

    1,$p

to print all the lines in the buffer, line 1 to last line. To stop the printing before it is finished, push the DEL (delete) key. _Ed_ responds with

    ?

and waits for the next command.

To print the last line of the buffer, it would be possible to use

    $,$p

However, _ed_ lets this be abbreviated to

    $p

Any single line can be printed by typing the line number followed by a _p_. Thus,

```
1p
```

produces the response

```
Now is the time
```

which is the first line of the buffer.

It is possible to abbreviate even further by entering the line number without the letter _p_. So

```
$
```

causes _ed_ to print the last line of the buffer.

The $ can be used in combinations such as

```
$-1,$p
```

which prints the last two lines of the buffer.

Exercise

Create some text using the _a_ command and experiment with the _p_ command. Verify that line 0 or a line beyond the end of the buffer cannot be printed and that attempts to print a buffer in reverse order by typing

```
3,1p
```

also fail.

7.3 Current Line

Suppose the buffer contains the six lines as above, that the command

```
1,3p
```

was issued, and that _ed_ has printed the three lines. Typing

```
p    (no line numbers)
```

causes _ed_ to print

```
to come to the aid of their party.
```

which is the third line of the buffer. It is also the last or most recent line that had actions performed on it. This p command can be repeated without line numbers, and ed continues to print line 3.

Ed maintains a record of the last line that had actions performed on it so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol dot (.).

Dot is a line number in the same way that $ is. It means "the current line" or "the line that most recently had action on it," and can be used in several ways. One possibility is to type

```
.,$p
```

This prints all the lines from and including the current line through the end line of the buffer. In this example, these are lines 3 through 6.

Some commands change the value of dot, and others do not. The p command sets dot to the number of the last line printed; the last command sets dot to six.

Dot is most useful in combinations such as:

```
.+1   (or .+1p)
```

This means "print the next line" and is a handy way to step slowly through a buffer.

The command

```
.-1    (or .-1p )
```

means "print the line before the current line." This allows the line number to go backwards. Another useful command is

```
.-3,.-1p
```

which prints the previous three lines.

Remember that all these commands change the value of dot. To find out what dot is at any time, type

```
.=
```

Ed responds by printing the value of dot.

To summarize, p can be preceded by zero, one, or two line numbers. If there is no line number given, ed prints the

current line; that is, the line that dot refers to. If there is one line number given with or without the letter p, it prints that line and sets dot there. If there are two line numbers, it prints all the lines in that range and sets dot to the last line printed. If two line numbers are specified, the first cannot be bigger than the second (Exercise 2).

Typing a single return prints the next line and is equivalent to .+1p. Typing a - is equivalent to .-1p.

7.4 Advanced Commands

For the experienced user, the list command (l) gives slightly more information than p. In particular, l makes characters visible that are normally invisible, such as tabs and backspaces. With l, each tab appears as > and each backspace appears as <. This command makes it much easier to correct typing mistakes that insert extra spaces adjacent to tabs, or insert a backspace followed by a space.

The l command also provides for displaying long lines on short terminals. Any line that exceeds 72 characters is displayed on multiple lines, and each folded line, except the last, is terminated by a backslash.

Occasionally, the l command prints a string of numbers preceded by a backslash, such as \07 or \16. These combinations make visible characters that normally do not print, such as form feed. Each such combination is a single character value of the nonprinting character in octal. Delete these characters unless they produce the desired result on the specific device used for ed output.

SECTION 8

DELETING LINES

Suppose the buffer contains two copies of _junk_ as in Section 6. To get rid of the three extra lines in the buffer, use the _delete_ command

    d

The lines to be deleted are specified for _d_ exactly as they are for _p_:

starting line, _ending line_ d

Thus the command

    4,$d

deletes line 4 through the end. There are now three lines left, which can be checked by entering

    1,$p

The $ now is line 3. Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to $.

Exercise

Experiment with _a_, _e_, _r_, _w_, _p_, and _d_. Be sure to understand how dot, $, and line numbers are used.

Next, try using line numbers with _a_, _r_, and _w_ as well. Verify that:

- _a_ appends lines _after_ the line number specified rather than after dot
- _r_ reads a file in _after_ the line number specified and not the end of the buffer
- _w_ writes out exactly the lines specified, not the whole buffer

These variations are sometimes handy. For instance, a file can be inserted at the beginning of a buffer by entering

    0r filename

Lines can be inserted at the beginning of the buffer by entering

```
0a
... text ...
.
```

SECTION 9

MODIFYING TEXT

9.1 Substitute Command

One of the most important commands is the substitute command

    s

which changes individual words or letters within a line or group of lines. It is used, for example, for correcting spelling mistakes and typing errors. This command has the most complexity of any *ed* command and can provide the greatest use.

9.2 Basic Modification

Suppose that line 1 reads

    Now is th time

The *e* has been left off *the*. Use *s* to fix this as follows:

    1s/th/the/

This says: "in line 1, substitute for the characters *th* the characters *the*." To verify that it works, type

    p

and get

    Now is the time

Dot must have been set to the line where the substitution took place, since the *p* command printed that line. Dot is always set this way with the *s* command.

The general way to use the substitute command is

starting-line, *ending-line* s/*change this*/*to this*/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in all the lines between starting-line and ending-line. Only the first occurrence on each line is changed, however. To change every occurrence, see Exercise 5. The rules for line numbers are the same as those for *p*, except that dot is

set to the last line changed. If no substitution took place, however, dot is not changed. This causes ? to appear as a warning.

Thus, enter

```
1,$s/speling/spelling/
```

to correct the first spelling mistake on each line in the text.

If no line numbers are given, the _s_ command assumes "make the substitution on line dot," so it makes changes only on the current line. This leads to the very common sequence

```
s/something/something else/p
```

which makes some correction on the current line, and then prints it.

It is also possible to type

```
s/something//
```

to change the first string of characters to _nothing_, that is, remove them. This is useful for deleting extra words in a line or for removing extra letters from words. For instance, in the line

```
Nowxx is the time
```

type

```
s/xx//p
```

to get

```
Now is the time
```

In _ed_, two adjacent slashes (//) mean no characters, not a blank.

Exercise

Experiment with the substitute command. Verify that the substitute command changes only the first occurrence of the first string. For example, enter:

```
a
the other side of the coin
.
s/the/on the/p
```

to get

```
on the other side of the coin
```

To change all occurrences, add a _g_ (for "global") to the _s_ command, like this:

```
s/ ... / ... /gp
```

Try other characters instead of slashes to delimit the two sets of characters in the _s_ command. Any character except blanks or tabs will work.

The following characters have special meanings:

```
^    .    $    [ ]    *    \    &
```

Read Section 13 for an explanation of their use.

9.3 Advanced Modificaton

Either form of the _s_ command can be followed by _p_ or _l_ to print or list the contents of the line. The commands

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

are all legal, and mean slightly different things. Also, _ed_ does not recognize _pg_ as being equivalent to _gp_.

Any _s_ command can be preceded by one or two line numbers to specify that the substitution is to take place on a group of lines. Thus, the command

```
1,$s/mispell/misspell/
```

changes the first occurrence of _mispell_ to _misspell_ on every line of the file, but the command

```
1,$s/mispell/misspell/g
```

changes every occurrence in every line.

Adding a p or l to the end of any of these substitute commands prints only the last line that was changed.

The undo command (u) "undoes" the last substitution: the last line that was substituted can be restored to its previous state by typing the command

    u

SECTION 10

CONTEXT SEARCHING

Suppose the original three lines of text are in the buffer:

```
Now is the time
for all good men
to come to the aid of their party.
```

To find the line that contains _their_, use context searching. This specifies a line, regardless of what its number is, by specifying some of its contents.

Say "search for a line that contains this particular string of characters" by typing

/_string of characters_/

For example, the _ed_ command

```
/their/
```

is a context search to find the next occurrence of the characters between slashes (their). It also sets dot to that line and prints the line for verification:

```
to come to the aid of their party.
```

"Next occurrence" means that _ed_ starts looking for the string at line .+1, searches to the end of the buffer, then continues at line 1 and searches to line dot. That is, the search "wraps around" from $ to 1. It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters cannot be found in any line, _ed_ types the error message

```
?
```

To search for the desired line and substitute with one command, enter

```
/their/s/their/the/p
```

which yields

```
to come to the aid of the party.
```

There are three parts to that command: context search for the desired line, make the substitution, and print the line.

Context searches are interchangeable with line numbers and can be used by themselves to find and print a desired line, or as line numbers for some other command, like _s_. They were used both ways in the previous examples.

With the buffer lines

```
Now is the time
for all good men
to come to the aid of their party.
```

the _ed_ line numbers

```
/Now/+1
/good/
/party/-1
```

are all context search expressions, and all refer to the same line (line 2). To make a change in line 2, enter

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

The choice is dictated by convenience. To print all three lines, enter

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

or by any number of similar combinations. The first of these is better if the number of lines involved is unknown.

_Ed_ also provides a shorthand for repeating a context search for the same string. For example, the _ed_ line number

```
/string/
```

finds the next occurrence of _string_. If this is not the desired line, the search must be repeated. This can be done by typing

```
//
```

This shorthand stands for the most recently used context search expression. It can also be used as the first string of the substitute command, as in

/string1/s//string2/

which finds the next occurrence of _string1_ and replaces it with _string2_.

Exercise

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Use context searches as line numbers for the substitute, print, and delete commands. Context searches are used less frequently with _r_, _w_, and _a_, but try them.

Try context searching using ?_text_? instead of /_text_/. This scans lines in the buffer in reverse order (end to beginning). This is useful when a desired string of characters is passed while going forward.

Again, the following characters have special meaning:

^ . $ [ ] * \ &

Read Section 13 for an explanation of their use.

SECTION 11

CHANGING AND INSERTING TEXT

This section discusses the *change* command and the *insert* command. Both of these commands operate on a group of one or more lines.

The *change* command is written as

```
c
```

and replaces a number of lines with different lines that are typed in at the terminal. For example, to change lines .+1 through $ to something else, type

```
.+1,$c
... type the lines of text here ...
.
```

The lines typed between the *c* command and the . take the place of the original lines between start line and end line. This is useful for replacing a line or several lines that have errors in them. It is possible to replace a single line with several lines.

If only one line is specified in the *c* command, just that line is replaced. The dot ends the input and works like the dot in the append command; it must appear by itself on a new line. If no line number is given, line dot is replaced and the value of dot is set to the last line typed in.

Insert (*i*) is similar to append. For instance

```
/string/i
... type the lines to be inserted here ...
.
```

inserts the given text before the next line that contains the string. The text between *i* and dot is inserted before the specified line. If no line number is specified, dot is used and dot is set to the last line inserted.

<u>Exercise</u>

The change command is rather like the combination delete followed by insert. Experiment to verify that

```
start, end d
i
... text ...
.
```

is like

```
start, end c
... text ...
.
```

These are not precisely the same if line $ gets deleted. Check this. What is dot?

Experiment with _a_ and _i_ to see that they are similar, but not the same. For instance,

```
line-number a
... text ...
.
```

appends after the given line, while

```
line-number i
... text ...
.
```

inserts before it. If no line number is given, _i_ inserts before line dot, but _a_ appends after line dot.

SECTION 12

MOVING TEXT

The *move* command (*m*) moves a group of lines from one place to another in the buffer. To put the first three lines of the buffer at the end, enter:

```
1,3m$
```

The general format is

start line, end line m *after this line*

where *after this line* specifies where to put the text.

The lines to be moved can also be specified by context searches. To reverse the two paragraphs

```
First paragraph
...
end of first paragraph.
Second paragraph
...
end of second paragraph.
```

type:

```
/Second/,/end of second/m/First/-1
```

The -1: moves the text before the line specified. Dot is set to the last line moved.

SECTION 13

USING SPECIAL CHARACTERS

13.1 General

The following characters have special meaning to ed when used in context searches and in the substitute command:

```
^    $    .    []    *    \    &
```

13.2 Period

On the left side of a substitute command or in a search with /.../, the period (.) stands for any single character. Thus, the search

```
/x.y/
```

finds any line where x and y occur and are separated by a single character, as in

```
x+y
x-y
x y
x.y
```

This is useful in conjunction with the repetition character (*). Thus, a* is a shorthand for any number of a's, and .* matches any number of any characters. The expression

```
s/.*/stuff/
```

changes an entire line and

```
s/.*,//
```

deletes all characters in the line up to and including the last comma (.* finds the longest possible match).

Since the period matches a single character, there is a way to deal with previously invisible characters printed by l.

Suppose there is a line that, when printed with the l command, appears as

```
....   th\07is   ....
```

The character string \07 really represents a single character (Section 7.4), so typing

```
s/th.is/this/
```

matches the character set between the h and the i, whatever it is. Since the period matches any single character, the command

```
s/./,/
```

converts the first character on a line into a comma.

The period has several meanings, depending on its context. The command

```
.s/././
```

shows all three.

The first period is the number of the line being edited, also called line dot. The second period is a special character that matches any single character on that line. The third period is the only one that is a literal period. On the right side of a substitution, a period is not special. Applying this command to the line

```
Now is the time.
```

results with

```
.ow is the time.
```

13.3 Backslash

The backslash (\) turns off any special meaning that the next character might have. In particular, \. converts . from a "match anything" into a period, so it can be used to replace the period in

```
Now is the time.
```

with a question mark like this:

```
s/\./?/
```

The pair of characters \. is interpreted by ed as a single period.

The backslash can also search for lines that contain a special character. To look for a line that contains

```
.PP
```

the search

```
/.PP/
```

is not adequate, because it finds a line

```
THE APPLICATION OF ...
```

since the . matches the letter A. However, the command

```
/\.PP/
```

finds only lines that contain .PP.

The backslash can also turn off special meanings for characters other than period. For example, to find a line that contains a backslash, precede one backslash with another as in

```
/\\/
```

Similarly, search for a forward slash (/) with

```
/\//
```

The backslash turns off the meaning of the immediately following /, so that it does not terminate the /.../ construction prematurely.

Any character can be used instead of slash to delimit the elements of an s command, but slashes must be used for context searching. For instance, in a line that contains many slashes, such as

```
//exec //sys.fort.go // etc...
```

a colon can be used as the delimiter. To delete all the slashes, type

```
s:/::g
```

Exercise

Find two substitute commands to convert the line

```
\x\.\y
```

into the line

```
\x\y
```

Here are several solutions to verify.

```
s/\\\.//
s/x../x/
s/..y/y/
```

13.4 Dollar Sign

Dollar sign ($) stands for the end of the line. To add the word _time_ to the end of the line

```
Now is the
```

use the dollar sign

```
s/$/ time/
```

to get

```
Now is the time
```

A space must appear before _time_ in the substitute command, or the result is

```
Now is thetime
```

To convert the line

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

into

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

the command needed is

```
s/,$/./
```

The $ sign here provides context to make specific which comma is meant. Without it, the _s_ command operates on the first comma, to produce

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

As another example, to convert

```
Now is the time.
```

into

```
Now is the time?
```

use

```
s/.$/?/
```

The dollar sign has multiple meanings depending on context. In the line

```
$s/$/$/
```

the first dollar sign refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign to be added to that line.

13.5 Circumflex

The circumflex (^) stands for the beginning of the line. To look for a line that begins with _the_, use

```
/^the/
```

to narrow the context and arrive at the desired word more easily.

The other use of ^ inserts text at the beginning of a line. The command

```
s/^/ /
```

places a space at the beginning of the current line.

Special characters can be combined. To search for a line that contains only the characters

```
.PP
```

use the command

```
/^\.PP$/
```

13.6 Asterisk

A character followed by an asterisk (*) stands for a variable number of consecutive occurrences of that character. A line can look like this:

```
text x                y text
```

where text stands for a lot of text and there is an undetermined number of spaces between the x and the y.

To replace all the spaces at once, use

```
s/x *y/x y/
```

Thus x *y means "an x, as many spaces as there are then a y."

The asterisk can be used with any character, not just space. If the original example were

```
text  x--------y  text
```

then all - signs can be replaced by a single space with the command

```
s/x-*y/x y/
```

To change a line entered as

```
text  x.......y  text
```

turn off the special meaning of dot (a match of any single character) with a backslash, as in

```
s/x\.*y/x y/
```

The because \.* means "as many periods as possible."

There are times when the pattern .* is exactly what is needed. For example, to change

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

into

```
Now is the time.
```

use .* to remove everything after the for with the command

```
s/ for.*/./
```

Zero is a legitimate number of possible occurrences. For example, for a line

```
text  xy  text  x             y  text
```

the command

```
s/x *y/x y/
```

was entered. The first xy matches this pattern, since it consists of an x, zero spaces, and a y. The result is that the substitute acts on the first xy, and does not touch the later one, which actually contains some intervening spaces.

The way around this is to specify a pattern like

```
/x  *y/
```

which describes an x, a space, then as many more spaces as possible, that is, one or more spaces, then a y.

The command to convert an x into y

```
s/x*/y/g
```

when applied to the line

```
abcdef
```

produces

```
yaybycydyeyfy
```

This is because zero is a legal number of matches. There are no x's at the beginning of the line, and no-x gets converted to a y. There are no x's between a and b, so the non-x (zero characters) is converted into y. This process continues down the string. To solve the problem, write

```
s/xx*/y/g
```

where xx* is one or more x's.

13.7 Brackets

The brackets ([]) match any element of the character class within them.

To delete any numbers that appear at the beginning of all lines of a file, use the construction

```
[0123456789]*
```

This matches zero or more digits. Thus, the command

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class. The only special characters inside the brackets are ^ in the initial position and - between characters; even the backslash does not have a special meaning.

To search for special characters, for example, use

```
/[.\$^[]/
```

Within [...], the [is not special. To get a] into a character class, make it the first character.

To abbreviate the digits, use [0-9]. Similarly, [a-z] stands for the lowercase letters, and [A-Z] for uppercase letters.

Specify a class that means "none of the following characters" by beginning the class with a circumflex. For example,

```
[^0-9]
```

stands for any character except a digit. To find the first line that does not begin with a tab or space, search with command

```
/^[^(space)(tab)]/
```

Within a character class, the ^ has a special meaning only if it occurs at the beginning. As an exercise, verify that

```
/^[^^]/
```

finds a line that does not begin with a circumflex.

13.8 Ampersand

The ampersand (&) is used to save typing. Suppose the line

```
Now is the time
```

must be changed to

```
Now is the best time
```

The command

```
s/the/the best/
```

can be used, but it is redundant to repeat the <u>the</u>. The ampersand eliminates that repetition. On the right side of

a substitute, the ampersand means "whatever was just matched," so the command

```
s/the/& best/
```

& stands for the. For example, to parenthesize a line, regardless of its length, use

```
s/.*/(&)/
```

The ampersand can occur more than once on the right side:

```
s/the/& best and & worst/
```

makes the original line into

```
Now is the best and the worst time
```

and

```
s/.*/&? &!!/
```

converts the original line into

```
Now is the time? Now is the time!!
```

To get a literal ampersand, use the backslash to turn off the special meaning. The command

```
s/ampersand/\&/
```

converts the word into the symbol. Ampersand has its special meaning only on the right side of a substitute command, not on the left side.

SECTION 14

USING GLOBAL COMMANDS

14.1 Global g

Global commands operate on the entire buffer instead of an individual line.

The global command (g) executes one or more ed commands on all lines in the buffer that match some specified string. For example

```
g/peling/p
```

prints all lines that contain peling. More usefully,

```
g/peling/s//pelling/gp
```

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

```
1,$s/peling/pelling/gp
```

which prints only the last line substituted. Another difference is that the g command does not give a ? if it does not find peling, but the s command does.

Use these examples to see the difference between the global command g and the g following a substitute command. These g's occur at different places in the command line and have different meanings.

14.2 Global v

The v command is the same as g, except that the commands are executed on every line that does not match the string following v. For example:

```
v/ /d
```

deletes every line that does not contain a blank.

14.3 Advanced Global Commands

The global commands g and v perform one or more editing commands on all lines that either contain (with g) or do not contain (with v) a specified pattern.

The pattern that goes between the slashes can be anything used in a line search or in a substitute command; the same rules and limitations apply.

The command

```
g/^\./p
```

prints all the formatting commands in a file because these lines begin with a dot. (Section 13 describes use of backslash to escape dot.)

The command that follows _g_ or _v_ can be anything. So

```
g/^\./d
```

deletes all lines that begin with . and

```
g/^$/d
```

deletes all empty lines.

Probably the most useful command that can follow a global is the substitute command to change and print each affected line for verification. For example, to change the word _zeus_ to _ZEUS_ everywhere and verify that it worked, enter

```
g/zeus/s//ZEUS/gp
```

The // in the substitute command means "the previous pattern," in this case, _zeus_. The _p_ command is done on every line that matches the pattern, not just those on which a substitution took place.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a _g_ or _v_ to use addresses or set dot. The command

```
g/^\.PP/+
```

prints the line that follows each .PP command. Remember that + means "one line past dot." The command

```
g/topic/?^\.SH?1
```

searches for each line that contains _topic_, scans backwards until it finds a line that begins .SH (a section heading) and prints the line that follows that, thus showing the section headings under which _topic_ is mentioned.

Finally,

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines between lines beginning with .EQ and .EN formatting commands.

The *g* and *v* commands can also be preceded by line numbers to search only those in the range specified.

14.4 Advanced Multiline Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is often cumbersome. As an example, suppose the task is to change *x* to *y* and *a* to *b* on all lines that contain *thing*. Then the commands

```
g/thing/s/x/y/\
s/a/b/
```

are sufficient. The backslash (\) signals the *g* command that the set of commands continues on the next line and terminates on the first line that does not end with \. A substitute command cannot be used to insert a new line within a *g* command.

To match the last pattern that was actually executed, use:

```
g/x/s/x/y/\
s/a/b/
```

To execute *a*, *c*, and *i* commands under a global command, add a backslash at the end of each line except the last. Thus, to add a *.nf* and *.sp* command before each .EQ line, type

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a . to terminate the *i* command unless there are further commands under the global.

SECTION 15

SUBSTITUTING NEW LINES

_Ed_ provides a facility for splitting a single line into two or more shorter lines by substituting a new line. As the simplest example, suppose a line is unmanageably long. If it looks like

    text xy _text_

it can be broken between the x and the y like this:

```
s/xy/x\
y/
```

This is actually a single command, although it is typed on two lines. The \ at the end of a line makes the following new line there no longer special.

Make a single line into several lines with this same mechanism. The word _very_ in a long line can be underlined by splitting _very_ onto a separate line and preceding it with the roff formatting command _.ul._

    text a very big _text_

The command

```
s/ very /\
.ul\
very\
/
```

converts the line into four shorter lines, preceding the word _very_ by the line _.ul,_ and eliminating the spaces around the _very_.

When a new line is substituted, dot points at the last line created.

SECTION 16

MANIPULATING LINES

16.1 Join Lines

Lines can be joined together with the *j* command. If dot is set to the first of the lines

```
Now is
 the time
```

The *j* command joins them. A blank has been added at the beginning of the second line because the command itself does not cause blanks to be added.

By itself, a *j* command joins line dot to line dot+1. Any contiguous set of lines can also be joined by specifying the starting and ending line numbers. For example,

```
1,$jp
```

joins all the lines into one line and prints it.

16.2 Rearrange Lines

Lines can be rearranged by tagging the pieces of the pattern by enclosing them between \(and \) and then rearranging the pieces. On the left side of a substitution, whatever matched that part is remembered and available for use on the right side. On the right side, the symbol \1 refers to whatever matched the first pair, \2 to the second pair, and so on.

For example, to convert a file of lines that consist of names in the form

```
Smith, A. B.
Jones, C.
```

to a file in the form

```
A. B. Smith
C. Jones
```

use the command

```
1,$s/^\([^,]*\), *\(.*\)/\2 \1/
```

The first \(...\) matches the last name (any string up to the comma) and is referred to on the right side with \1. The second \(...\) is whatever follows the comma and any spaces, and is referred to as \2.

When this type of editing is performed, use the global commands g or v followed by p to print each substitution as it is made.

SECTION 17

MANIPULATING ADDRESSES

17.1 Line Addressing

Line addressing is the method used to specify what lines are to be affected by editing commands. Constructions like

```
1,$s/x/y/
```

start on line 1 and specify a change on all lines.

17.2 Address Arithmetic

Line numbers such as . and $ can be combined with + and - in a process called address arithmetic. For example,

```
$-1
```

is a command to print the next-to-last line of the current file (that is, one line before line $). To see how much was entered in a previous editing session, use

```
$-5,$p
```

to print the last six lines.

The command

```
.-3,.+3p
```

prints from three lines before the current line to three lines after. The + can be omitted, so the command

```
.-3,.3p
```

is identical in meaning.

The - and + can be used as line numbers by themselves. The - by itself is a command to move up one line in the file. Several minus signs can be strung together to move back that many lines. For example,

```
---
```

moves up three lines, as does -3. Thus

```
-3,+3p
```

is also identical to the previous examples.

Since - is shorter than -1, constructions such as

```
-,.s/bad/good/
```

are useful. This changes _bad_ to _good_ on the previous line and on the current line.

The + and - can be used in combination with searches using /.../ and ?...?, and with $. The search

```
/thing/--
```

finds the line containing _thing_, and positions dot two lines before it.

SECTION 18

DOING REPEATED SEARCHES

The construction

```
//
```

is a shorthand for "the previous thing that was searched for," whatever it was. This can be repeated as many times as necessary. The search can also go backwards. The command

```
??
```

searches for the same thing, but in the reverse direction.

The // can also be used as the left side of a substitute command to mean the most recent pattern. The command

```
/horrible thing/
s//good/p
```

finds the line containing _horrible thing_, prints the line, changes _horrible thing_ to _good_, and prints the changed line.

To go backwards and change a line, enter

```
??s//good/
```

The & can be used on the right side of a substitute to stand for the character that was matched. The command

```
//s//& &/p
```

finds the next occurrence of whatever was searched for last, replaces it with two copies of itself, then prints the line.

SECTION 19

USING DEFAULT LINE REFERENCES

One of the most effective ways to speed up editing is always knowing what lines will be affected by a command and the value of dot when a command finishes.

If a search command

/thing/

is issued, dot points at the next line that contains _thing_. No address is required with commands

_s_ to make a substitution on that line

_p_ to print it

_l_ to list it

_d_ to delete it

_a_ to append text after it

_c_ to change it

_i_ to insert text before it

If no match occurs, the position of dot is unchanged. This is also true if dot is at the only _thing_ when the command is issued. The same rules hold for searches that use ?...?; the only difference is the direction of the search.

The delete command _d_ leaves dot pointing at the line that followed the last deleted line. If line $ gets deleted, however, dot points at the new last line.

The line-changing commands _a_, _c_, and _i_ all affect the current line. If no line number is given with them, _a_ appends text after the current line, _c_ changes the current line, and _i_ inserts text before the current line.

Commands _a_, _c_, and _i_ move dot to the last line entered. For example, the commands

```
a
 ... text ...
 ... botch ...          (minor error)
.
s/botch/correct/        (fix line)
a
 ... more text ...
```

can be given without specifying any line number for the substitute command or for the second append command. Alternatively, use

```
a
 ... text ...
 ... horrible botch ...        (major error)
.
c                              (replace entire line)
 ... fixed line ...
```

The _r_ command reads a file into the text being edited, either at the end if no address is given, or after the specified line if there is an address. In either case, dot points at the last line read. Remember that _0r_ reads a file in at the beginning of the text.

The _w_ command writes the entire file. If the command is preceded by one line number, that line is written. If it is preceded by two line numbers, that range of lines is written. The _w_ command does not change dot; the current line remains the same, regardless of what lines are written. This is true even if there is a command such as

```
/^\.AB/,/^\.AE/w abstract
```

involving a context search.

The _s_ command positions dot on the last line that changed. If there were no changes, then dot is unchanged.

With the text

```
x1
x2
x3
```

the command

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. With the three lines

```
x1
y2
y3
```

the same command changes and prints only the first line and positions dot there.

SECTION 20

USING THE SEMICOLON

In *ed*, the semicolon (;) can be used like comma, except that a semicolon forces dot to be set where the line numbers are being evaluated. In effect, the semicolon moves dot.

Searches with /.../ and ?...? start at the current line and move forward or backward until they either find the pattern or return to the current line. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
.
bc
.
.
```

Starting at line 1, the command

```
/a/,/b/p
```

would be expected to print all the lines from the *ab* to the *bc*. Instead, both searchs start from the same point and they both find the line that contains *ab*. The result is to print a single line. Worse, if there had been a line with a *b* in it before the *ab* line, the print command would be in error, since the second line number would be less than the first; it is illegal to try to print lines in reverse order.

The comma separator for line numbers does not set dot as each address is processed. Instead, each search starts from the same place. Thus, in this example, the command

```
/a/;/b/p
```

prints the range of lines from *ab* to *bc*. After the *a* is found, dot is set to that line, then *b* is searched for, starting beyond that line.

To find the second occurrence of *thing*, enter

```
/thing/;//
```

This finds the first occurrence of _thing_, sets dot to that line, then finds the second and prints only that.

Closely related is searching for the second previous occurrence of something, as in

```
?something?;??
```

As an exercise, try printing the third or fourth occurrence in either direction.

To find the first occurrence of something in a file, starting at an arbitrary place within the file, use

```
0;/thing/
```

This starts the search at line 1.

SECTION 21

INTERRUPTING THE EDITOR

Pressing the INTERRUPT, DELETE, RUBOUT, or BREAK key while *ed* is doing a command restores the state in effect before the command began. An interrupt during reading or writing a file, making substitutions, or deleting lines stops the command in an unpredictable state and does not always change dot.

Printing does not change dot until the printing is done. Thus, if the DELETE key is pressed while a file is being printed, dot is still where it was when the *p* command was started.

SECTION 22

MANIPULATING FILES

22.1 General

In addition to editor commands, other commands exist to manipulate files. Manipulating files includes changing the name of a file, making a copy of a file somewhere else, moving a few lines from one place to another in a file, inserting one file in the middle of another, splitting a file into pieces, and splicing two or more files together.

22.2 Change the Name of a File

To change a file name, use _mv_.

    mv _oldname_ _newname_

This program moves the file from the old name to the new name. For example, to change a file named _memo_ into one called _paper_, enter

    mv memo paper

NOTE

> If there is already a file with the new name, its present contents are overwritten by the information from the old file. Also, a file cannot be moved to itself. So
>
>   mv x x
>
> is illegal.

22.3 Copy a File

Copy a file with the _cp_ command. The format of _cp_ is

    cp _original_ _copy_

to copy _original_ into _copy_. To save a file called _good_ choose a name (here _savegood_) then type

    cp good savegood

This copies _good_ onto _savegood_, so that there are two identical copies of the file _good_. If _savegood_ previously contained something, it is overwritten.

To restore the original state of _good_, enter

```
mv  savegood  good
```

which erases _savegood_, or

```
cp  savegood  good
```

to retain a safe copy.

22.4 Remove a File

To remove a file forever, use the _rm_ command. The entry

```
rm  savegood
```

permanently erases the file called _savegood_.

22.5 Put Two or More Files Together

Collecting two or more files into one is performed with _cat_ (short for concatenate).

To combine the files _file1_ and _file2_ into a single file called _bigfile_, enter

```
cat  file1  file2  >bigfile
```

The > before _bigfile_ means to take the output of the _cat_ command and put it into _bigfile_. As with _cp_ and _mv_, anything that was already in bigfile is destroyed.

More than two files can be combined. The command

```
cat  file1  file2  file3  ...  >bigfile
```

collects many files.

22.6 Adding Text to the End of a File

To add one file to the end of another, use the >> construction. This is identical to >, except that instead of overwriting the old file, it simply adds text at the end. Thus, enter

```
cat  good1  >>good
```

to add good1 to the end of good. If good did not previously exist, this makes a copy of good1 called good.

22.7 Insert One File into Another

Suppose that a file called memo needs the file called table to be inserted just after the reference to Table 1. That is, in memo somewhere is a line that says

```
Table 1 shows that ...
```

and the data contained in table goes there.

Edit memo, find Table 1, and add the file table by entering

```
ed  memo
/Table 1/
Table 1 shows that ... [response from ed]
.r  table
```

The critical line is the last one; the r command reads the file table and inserts it immediately after the referenced line.

22.8 Write Part of a File

It is possible to split into a separate file the table from the previous example. In the file being edited, there are the lines

```
.TS
...[lots of stuff]
.TE
```

To isolate the table in a separate file called table, first find the start of the table (the .TS line), then write out the table

```
/^\.TS/
.TS  [ed prints the line it found]
.,/^\.TE/w table
```

All these steps can be consolidated with

```
/^\.TS/;/^\.TE/w table
```

The w command can write out a group of lines instead of the whole file. In fact, a single line can be written by giving

one line number instead of two. For example, if there is a complicated line that is going to be needed later, save it to avoid retyping it. Enter

```
a
...lots of stuff...
...complicated line...
.
.w  temp
a
...more stuff...
.
.r temp
a
...more stuff...
.
```

22.9 Move Lines

To move a paragraph from its present position in a paper to the end, use the editor _move_ command (_m_).

The _m_ command takes up to two line numbers in front that tell what lines are to be affected. It is also followed by a line number that tells where the lines are to go. Thus

```
line1, line2 m line3
```

says to move all the lines between _line1_ and _line2_ after _line3_.

If dot is at the first line of the paragraph beginning with .PP, type

```
.,/^\.PP/-m$
```

The order of two adjacent lines can be reversed by positioning the first one after the second. If dot is at the first line, the command

```
m+
```

moves line dot to a position one line after the first line. If dot is at the second line, the command

```
m--
```

interchanges the two lines.

The _m_ command is more succinct and direct than writing, deleting, and rereading. The main difficulty with the _m_

command is that if patterns are used to specify both the lines being moved and the target, they must be specified properly. Doing the job a step at a time makes it easier to verify at each step that the desired result is accomplished. Issue a _w_ command before doing any complicated commands. If there is an error, it is easy to back up.

22.10 Mark a Line

_Ed_ provides a facility for marking a line with a particular name to later reference it by name, regardless of its line number. This can be handy for moving lines and for keeping track of them as they move. The _mark_ command is _k_. The command

```
kx
```

assigns the name _x_ to the current line, where _x_ is any single lowercase letter. (To mark a line for which the line number is known, precede the k with the line number.) Refer to the marked line with the address

```
'x
```

For example, to move a block of text, find the first line of the block to be moved, and mark it with _ka_. Then find the last line and mark it with _kb_. Now position dot where the text is to go and enter

```
'a,'bm.
```

Only one line can have a particular mark name associated with it at any given time.

22.11 Copy Lines

_Ed_ provides another command, called _t_ (for transfer) for making a copy of a group of one or more lines. This is often easier than writing and reading.

The _t_ command is identical to the _m_ command, except that instead of moving lines it duplicates them at the place named. Thus

```
1,$t$
```

duplicates the entire file that is being edited.

A more common use for _t_ is for creating a series of lines that differ only slightly. For example, type

```
a
..........  x  .........  (long line)
.
t.          (make a copy)
s/x/y/      (change it a bit)
t.          (make third copy)
s/y/z/      (change it a bit)
```

and so on.

22.12 Temporary Escape

The escape command (!) provides a way to temporarily leave the editor for a ZEUS command and immediately return to the editor.

Entering

```
!any ZEUS command
```

suspends the current editing state and executes the command asked for. When the command finishes, _ed_ prints another prompt and editing can be resumed. Any ZEUS command, including another _ed_, can be entered following the escape.

SECTION 23

SUPPORTING TOOLS

23.1 General

There are several tools and techniques based on the editor. In this section are some introductory examples of these tools.

23.2 Grep

To find all occurrences of some word or pattern in a set of files, use the program _grep_. The search patterns described in the document are often called "regular expressions," and "grep" stands for

```
g/re/p  (get / regular expression / print)
```

That describes exactly what _grep_ does--it prints every line in a set of files that contains a particular pattern. Thus

```
grep  'thing'  file1  file2  file3  ...
```

finds _thing_ wherever it occurs in any of the files listed. _Grep_ also indicates the file in which the line was found for any further file manipulation.

The pattern represented by _thing_ can be any pattern that can be used in _ed_. Always enclose the pattern in single quotes if it contains any nonalphabetic characters. These characters carry special meaning in the ZEUS command interpreter (Section 15).

There is also a way to find lines that do not contain a pattern. The command

```
grep  -v  'thing'  file1  file2  ...
```

finds all lines that do not contain _thing_. The _-v_ must occur in the position shown. Given _grep_ and _grep -v_, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain _x_ but not _y_, use

```
grep  x  file...  |  grep  -v  y
```

The notation | is a pipe command, which causes the output of the first command to be used as input to the second command.

23.3 Editing Scripts

To execute a complicated set of editing operations on a set of files, make up a script, that is, a file that contains the operations to perform. Then apply this script to each file.

For example, to change every _Zeus_ to _ZEUS_ and every _bad_ to _good_ in a large number of files, put into the file script the lines

```
g/Zeus/s//ZEUS/g
g/bad/s//good/g
w
q
```

Now enter

```
ed file1 <script
ed file2 <script
...
```

This causes _ed_ to take its commands from the prepared script.

25.4 Sed

Sed (stream editor) processes unlimited amounts of input. _Sed_ copies its input to its output, applying one or more editing commands to each line of input.

As an example, to change _Zeus_ to _ZEUS_ as in the previous example without rewriting the files, use the command

```
sed 's/Zeus/ZEUS/g' file1 file2 ...
```

This applies the command s/Zeus/ZEUS/g to all lines from the files specified and copies all lines to the output. The advantage of using _sed_ is that it handles input too large for _ed_. All the output can be collected in one place, and either saved in a file or piped into another program.

If the editing transformation is so complicated that more than one editing command is needed, commands can be supplied from a file with a slightly more complex syntax. To take commands from a file, for example, use

```
sed -f cmdfile input-files...
```

APPENDIX A

SUMMARY OF COMMANDS AND LINE NUMBERS

The general form of ed commands is the command name, perhaps preceded by one or two line numbers, and, in the case of e, r, and w, followed by a file name. Only one command is allowed per line, but a p command can follow commands other than e, r, w, and q.

a: Append (add) lines to the buffer at line dot unless a different line is specified. Appending continues until dot is typed on a new line. Dot is set to the last line appended.

c: Change the specified lines to the new text that follows. The new lines are terminated by a dot, as with a. If no lines are specified, line dot is changed. Dot is set to the last line changed.

d: Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless $ is deleted, in which case dot is set to $.

e: Edit new file. Previous contents of the buffer are deleted.

f: Print current filename. If a name follows f, the current name is set to it.

g: The command

```
g/---/commands
```

executes the commands on those lines that contain ---, which can be any context search expression.

i: Insert lines before specified line or dot until a dot is typed on a new line. Dot is set to the last line inserted.

m: Move lines specified to a position after the line specified after m. Dot is set to the last line moved.

p: Print specified lines. If none are specified, print line dot. A single line number is equivalent to line number p. A single return prints .+1 (the next line).

q: Quit ed. Deletes all text in buffer if it is given twice in a row without first giving a w command.

r: Read a file into the end of the buffer unless a different location is specified. Dot is set to last line read.

s: The command

```
s/string1/string2/
```

substitutes the characters string2 for string1 in the specified lines. If no lines are specified, it makes the substitution in line dot. Dot is set to the last line in which a substitution took place (if no substitution took place, dot is not changed). s changes only the first occurrence of string1 on a line. To change all occurences, type a g after the final slash.

v: The command

```
v/---/commands
```

executes commands on those lines that do not contain ---, which can be any context search expression.

w: Write out buffer onto a file. Dot is not changed.

.=: Print value of dot. The = by itself prints the value of $.

!: The line

```
!command-line
```

causes command-line to be executed as a ZEUS command.

/-----/: Context search. Search for next line that contains this string of characters and print it. Dot is set to the line where string was found. Search starts at .+1, wraps around from $ to 1, and continues to dot, if necessary.

?-----?: Context search in reverse direction. Start search at .-1, scan to 1, and wrap around to $.

FILE SYSTEM CHECK PROGRAM (FSCK) REFERENCE MANUAL

PREFACE

This document describes how the file system check program (fsck) maintains file system integrity. More broadly, this document presents the normal updating of the file system, discusses the possible causes of file system corruption, and describes the corrective actions used by fsck. Both internal functions of the fsck program and the interaction between the program and the operator appear in this document.

Section 1 gives a brief introduction to fsck, and Section 2 discusses normal updating of the file system. File system corruption is described in Section 3. Section 4 presents the set of corrective actions used by fsck. Error conditions and operator actions are explained in the Appendix.

In this document, the sentence structure "fsck can ..." often appears; for example, "Fsck can clear the inodes." This is a shorthand notation for the process of fsck prompting the operator, the operator responding to continue fsck, and fsck actually performing the action, in this case, clearing the inodes (setting its contents to zero).

Additional information on fsck appears in the _ZEUS Software Reference Manual_ (part number 03-3195) under _fsck_(1).

TABLE OF CONTENTS

SECTION 1 INTRODUCTION 5

SECTION 2 UPDATE OF THE FILE SYSTEM 6

2.1 General 6
2.2 Super-Block 6
2.3 Inodes 6
2.4 Indirect Blocks 6
2.5 Data Blocks 7
2.6 Free-List Blocks 7

SECTION 3 CORRUPTION OF THE FILE SYSTEM 8

3.1 General 8
3.2 Improper System Shutdown and Startup 8
3.3 Hardware Failure 8

SECTION 4 DETECTION AND CORRECTION OF CORRUPTION ... 9

4.1 General 9
4.2 Super-Block 9

4.2.1 File-System Size and Inode-List Size 9
4.2.2 Free-Block List 9
4.2.3 Free-Block Count 10
4.2.4 Free-Inode Count 10

4.3 Inodes 10

4.3.1 Format and Type 10
4.3.2 Link Count 11
4.3.3 Duplicate Blocks 11
4.3.4 Bad Blocks 12
4.3.5 Size Checks 12

4.4 Indirect Blocks 12
4.5 Data Blocks 13
4.6 Free-List Blocks 13

TABLE OF CONTENTS (continued)

APPENDIX A FSCK ERROR CONDITIONS 15

A.1 Conventions 15
A.2 Initialization 15
A.3 Phase 1: Check Blocks and Sizes ... 18
A.4 Phase 1B: Rescan for More Duplicates 21
A.5 Phase 2: Check Path Names 22
A.6 Phase 3: Check Connectivity 24
A.7 Phase 4: Check Reference Counts ... 25
A.8 Phase 5: Check Free List 28
A.9 Phase 6: Salvage Free List 30
A.10 Cleanup 31

SECTION 1

INTRODUCTION

When the ZEUS Operating System is brought up, the file system check program (fsck) must be run. Fsck is an interactive file system program that uses the redundant structural information in the ZEUS file system to perform consistency checks. Fsck detects file inconsistencies and reports them to an operator who elects to fix or ignore them. This precautionary measure helps to ensure a reliable environment for file storage on disk.

Every file activity (creation, modification, or deletion) updates at least one of the five data blocks that ZEUS uses to monitor files. Fsck checks for matches in the contents of redundant fields among these blocks, and for matches between information in the blocks and the files themselves. When any error is found, fsck reports it to an operator. Most errors allow for operator intervention to continue running fsck or to terminate it. Serious errors, such as illegal options, cause fsck to terminate.

SECTION 2

UPDATE OF THE FILE SYSTEM

2.1 General

Every time a file is created, modified, or removed, the ZEUS Operating System performs a series of file system updates on the super-block, inodes, indirect blocks, data blocks (directories and files), and free-list blocks. Update requests are honored in a specific order to yield a consistent file system. Knowing this order makes it easier to understand what happens when a problem occurs, and to repair a corrupted file system.

2.2 Super-Block

The super-block contains information about the size of the file system, the size of the inode list, part of the free-block list, the count of free blocks, the count of free inodes, and part of the free-inode list.

The root file system is always mounted, and the super-block of a mounted file system is written to the file system whenever the file system is unmounted or a sync command is issued.

2.3 Inodes

An inode contains information about the type of inode (directory, data, or special), the number of directory entries linked to the inode, the list of blocks claimed by the inode, and the size of the inode.

An inode is written to the file system on closure of the file associated with the inode, and when a sync command is issued.

2.4 Indirect Blocks

There are three types of indirect blocks: single-indirect, double-indirect, and triple-indirect. A single-indirect block contains a list of some of the block numbers claimed by an inode. Each of the 128 entries in an indirect block is a data-block number. A double-indirect block contains a list of single-indirect block numbers, and a triple-indirect block contains a list of double-indirect block numbers.

Indirect blocks are written to the file system when the operating system modifies them and queues them for writing. Actual I/O is deferred until ZEUS needs the buffer or a sync command is issued.

2.5 Data Blocks

A data block contains file information or directory entries. Each directory entry consists of a file name and an inode number.

Data blocks are written to the file system when the operating system modifies them and queues them for writing. Actual I/O is deferred until ZEUS needs the buffer or a sync command is issued.

2.6 Free-List Blocks

The free-list blocks list all blocks that are not allocated to the super-block (only the first free-list block), inodes, indirect blocks, or data blocks. Each free-list block contains a count of the entries in this free-list block, a pointer to the next free-list block, and a partial list of free blocks in the file system.

Free-list blocks are written to the file system when the operating system modifies them and queues them for writing. Actual I/O is deferred until ZEUS needs the buffer or a sync command is issued.

SECTION 3

CORRUPTION OF THE FILE SYSTEM

3.1 General

The most common reasons for corruption of a file system are improper shutdown and hardware failure.

3.2 Improper System Shutdown and Startup

Improper shutdown procedures include forgetting to sync the system prior to halting the CPU, physically write-protecting a mounted file system, and taking a mounted file system off-line.

Improper startup procedures include not checking a file system for inconsistencies and not repairing inconsistencies.

3.3 Hardware Failure

Any piece of hardware can fail at any time. Failures range from a bad block of a disk pack to a nonfunctional disk controller.

SECTION 4

DETECTION AND CORRECTION OF CORRUPTION

4.1 General

A quiescent file system (one that is unmounted and not being written on) can be checked for structural integrity by performing consistency checks of the redundant data that is part of the file system. A quiescent state is important during the file system check because of the multipass nature of the fsck program. Fsck discovers each file inconsistency, reports it to the operator, and allows for interactive corrective action.

This section discusses how to discover inconsistencies and take corrective actions for super-blocks, inodes, indirect blocks, data blocks containing directory entries, and free-list blocks.

4.2 Super-Block

The super-block is most prone to corruption because every change to the file system's block or inodes modifies the super-block. Corruption most frequently occurs when the computer is halted and the last command involving the output of the file system was not a sync command.

Check the super-block for inconsistencies involving file-system size, inode-list size, free-block list, free-block count, and the free-inode count.

4.2.1 File-System Size and Inode-List Size

These sizes are critical because all other checks of the file system depend on them, and because fsck can only check for them being within reasonable bounds. The file system size must be larger than both the number of blocks used by the super-block and the number of blocks used by the list of inodes. The number of inodes must be less than 65,535.

4.2.2 Free-Block List

The free-block list starts in the super-block and continues through the free-list blocks of the file system. Each free-list block is checked for a list count out of range, for block numbers out of range, and for blocks already

allocated within the file system. A check is made to see that all the blocks in the file system were found. If anything is wrong with the free-block list, fsck can rebuild it, excluding all blocks in the list of allocated blocks.

Fsck checks the list count for the first free-block for a value of less than zero or greater than 50. It also checks each block number for a value of less than the first data block in the file system. Then it compares each block number to a list of already allocated blocks. If the free-list block pointer is nonzero, the next free-list block is read in, and the process is repeated.

When all the blocks have been accounted for, a check is made to see if the number of blocks used by the free-block list plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

4.2.3 Free-Block Count

The super-block contains a count of the total number of free blocks within the file system. Fsck compares this count to the number of blocks it found free within the file system and, if they do not agree, replaces the count in the super-block with the actual free-block count.

4.2.4 Free-Inode Count

The super-block contains a count of the total number of free inodes within the file system. Fsck compares this count to the number of inodes it found free within the file system and, if they do not agree, replaces the count in the super-block with the actual free-inode count.

4.3 Inodes

A large quantity of active inodes increases the likelihood of corruption. Fsck sequentially checks the list of inodes for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

4.3.1 Format and Type

Each inode contains a mode word that describes the type and state of the inode. Valid inode types are regular, directory, special block, and special character. Valid inode states are unallocated, allocated, and neither allocated nor unallocated (incorrectly formatted as a result of bad data

being written into the inode list through hardware failure). Fsck can clear the inode.

4.3.2 Link Count

A count of the total number of directory entries linked to the inode is contained in each inode. Fsck verifies this count by traversing down the total directory structure starting from the root directory and calculating an actual link count for each inode.

If the stored link count is nonzero and the actual link count is zero, no directory entry appears for the inode. Fsck can link the disconnected file to the lost+found directory.

If the stored and actual link counts are nonzero and unequal, a directory entry may have been added or removed without the inode being updated. Fsck can replace the stored link count with the actual link count.

4.3.3 Duplicate Blocks

Each inode contains a list, or pointers to lists, (indirect blocks) of all the blocks claimed by the inode. Fsck compares each block number claimed by an inode to a list of already allocated blocks. If there are any inconsistencies, fsck can clear both inodes.

If a block number is already claimed by another inode, the block number is added to a list of duplicate blocks. Otherwise, the list of allocated blocks is updated to include the block number.

If there are any duplicate blocks, fsck makes a partial pass of the inode list to find the duplicate block. Fsck needs to examine the files associated with these inodes to determine which inode is corrupted and should be cleared (most frequently, this is the inode with the earlier modify time). This error condition occurs when using a file system with blocks claimed by both the free-block list and by other parts of the file system.

A large number of duplicate blocks in an inode is often due to an indirect block not being written to the file system.

4.3.4 Bad Blocks

Each inode contains a list, or pointer to lists, of all the blocks claimed by the inode. Fsck checks each block number claimed by an inode for a value within the range bounded by the first data block (minimum) and the last block (maximum) in the file system. A block number outside the range is called a bad block number.

A large number of bad blocks in an inode can be due to an indirect block not being written to the file system.

4.3.5 Size Checks

Each inode contains a 32-bit (four-byte) size field that contains the number of characters in the file associated with the inode. Fsck checks this field for inconsistencies such as directory sizes that are not a multiple of 16 characters, and for the number of blocks actually used not matching the number indicated by the inode size.

A directory inode in the ZEUS file system has the directory bit set on in the inode mode word. The directory size must be a multiple of 16 because a directory entry contains 16 bytes of information, two bytes for the inode number and 14 bytes for the file or directory name. Fsck warns of directory misalignment, but cannot gather sufficient information to correct the problem.

Fsck calculates the number of blocks that there should be in an inode by dividing the number of characters in an inode by the number of characters per block (512), rounding up, and adding one block for each indirect block associated with the inode. If this computed number does not match the actual number of blocks, fsck warns of a possible file-size error, but does not correct it because ZEUS does not insert blocks into files that are created in random order.

4.4 Indirect Blocks

Since indirect blocks are owned by an inode, inconsistencies in the indirect blocks affect the inode. Fsck checks that blocks are not already claimed by another inode, and that block numbers are not outside the range of the file system. The procedures discussed in Sections 4.3.3 and 4.3.4 are iteratively applied to each level of indirect blocks.

4.5 Data Blocks

The two types of data blocks are plain data blocks and directory blocks. Plain data blocks contain the information stored in a file and are not checked by fsck.

Directory data blocks contain directory entries and are checked for inconsistencies involving directory inode numbers pointing to unallocated inodes, directory inode numbers greater than the number of inodes in the file system, incorrect directory inode numbers for . (current directory) and .. (parent directory), and directories that are disconnected from the file system.

If a directory entry inode number points to an unallocated inode, fsck can remove that directory entry. This condition usually occurs when the data block containing the directory entries are modified and written to the file system, and the inode is not yet written.

If a directory entry inode number is pointing beyond the end of the inode list, fsck can remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for . must be the first entry in the directory data block. Its value must equal the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory). If the directory inode numbers are incorrect, fsck can replace them with the correct values.

Fsck checks the general connectivity of the file system. If directories are not linked into the file system, fsck links the directory back into the file system in the lost+found directory. This condition can be caused by inodes being written to the file system without the corresponding directory data blocks being written.

4.6 Free-List Blocks

Free-list blocks are owned by the super-block, and inconsistencies in free-list blocks directly affect the super-block.

Fsck can check for a list count outside of range, block numbers outside of range, and blocks already associated with the file system.

Section 4.2.2 contains a discussion of detection and correction of the inconsistencies associated with free-list blocks.

APPENDIX A

FSCK ERROR CONDITIONS

A.1 Conventions

Fsck is a multipass file system check program with each file system pass invoking a different phase of the fsck program. After the initial setup, fsck performs successive phases on each file system, checks blocks and sizes, path names, connectivity, reference counts, and the free-block list (which might be rebuilt), and performs some cleanup.

When an inconsistency is detected, fsck reports it to the operator. If a response is required, fsck prints a prompt message and waits for a response. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the phase of the fsck program in which they occur. The error conditions that occur in more than one phase are discussed in the Initialization Section.

A.2 Initialization

Before a file system check can be performed, certain tables must be set up and certain files must be opened. This section lists error conditions resulting from initializing tables and opening files; specifically, it lists error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file.

C OPTION ?

C is not a legal option of fsck; legal options are -y, -n, -s, -S, and -t. Fsck terminates on this error condition (*fsck*(*1*)).

BAD -t OPTION

The -t option is not followed by a file name. Fsck terminates on this error condition (*fsck*(*1*)).

INVALID -s ARGUMENT, DEFAULTS ASSUMED

The -s option is not suffixed by 3, 4, blocks-per-cylinder, or blocks-to-skip. Fsck assumes a default value of 400 blocks-per-cylinder and nine blocks-to-skip (fsck(1)).

INCOMPATIBLE OPTIONS: -n and -s

It is not possible to salvage the free-block list without modifying the file system. Fsck terminates on this error condition (fsck(1)).

CAN'T GET MEMORY

Fsck's request for memory for its virtual memory tables failed. This should never happen. Fsck terminates on this error condition. See an experienced fsck user.

CAN'T OPEN CHECKLIST FILE: F

The default file system checklist file F (usually /etc/checklist) cannot be opened for reading. Fsck terminates on this error condition. Check access modes of F.

CAN'T STAT ROOT

Fsck's request for statistics about the root directory / failed. This should never happen. Fsck terminates on this error condition. See an experienced fsck user.

CAN'T STAT F

Fsck's request for statistics about the file system F failed. Fsck ignores this file system and continues checking the next file system given. Check access modes of F.

F IS NOT A BLOCK OR CHARACTER DEVICE

Fsck has a wrong regular file name that it ignores, and it continues checking the next file system given. Check file type of F.

CAN'T OPEN F

The file system F cannot be opened for reading. Fsck ignores this file system and continues checking the next file system given. Check access modes of F.

SIZE CHECK: fsize X isize Y

More blocks are used for the inode list Y than there are blocks in the file system X, or there are more than 65,535 inodes in the file system. Fsck ignores this file system and continues checking the next file system given (Section 4.2.1).

CAN'T CREATE F

Fsck's request to create a scratch file F failed. Fsck ignores this file system and continues checking the next file system given. Check access modes of F.

CANNOT SEEK: BLK B (CONTINUE?)

Fsck's request for moving to a specified block number B in the file system failed. This should never happen. See an experienced fsck user.

Possible responses to the CONTINUE? prompt are:

YES Attempt to continue to run the file system check. Often, however, the problem persists since this error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If the block is part of the virtual memory buffer cache, fsck terminates with the message FATAL I/O ERROR.

NO Terminate the program.

CANNOT READ: BLK B (CONTINUE?)

Fsck's request for reading a specified block number B in the file system failed. This should never happen. See an experienced fsck user.

Possible responses to the CONTINUE? prompt are:

YES Attempt to continue to run the file system check. Often, however, the problem persists since this error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If the block is part of the virtual memory buffer cache, fsck terminates with the message FATAL I/O ERROR.

NO Terminate the program.

CANNOT WRITE: BLK B (CONTINUE?)

Fsck's request for writing a specified block number B in the file system failed. The disk is write-protected. See an experienced fsck user.

Possible responses to the CONTINUE? prompt are:

YES Attempt to continue to run the file system check. Often, however, the problem persists since this error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If the block is part of the virtual memory buffer cache, fsck terminates with the message FATAL I/O ERROR.

NO Terminate the program.

A.3 Phase 1: Check Blocks and Sizes

This phase concerns itself with the inode list. This section lists error conditions resulting from checking inode types, setting up the zero-link-count table, examining inode block numbers for bad or duplicate blocks, checking inode size, and checking inode format.

UNKNOWN FILE TYPE I=I (CLEAR?)

The mode word of the inode I indicates that the inode is not a special character inode, regular inode, or directory inode (Section 4.3.1).

Possible responses to the CLEAR? prompt are:

YES Continue with the program. This error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If another allocated inode with a zero link count is found, this error condition is repeated.

NO Terminate the program.

LINK COUNT TABLE OVERFLOW (CONTINUE?)

An internal table for fsck containing allocated inodes with a link count of zero has no more room. Recompile fsck with a larger value of MAXLNCNT.

Possible responses to the CONTINUE? prompt are:

YES Continue with the program. This error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If another allocated inode with a zero link count is found, this error condition is repeated.

NO Terminate the program.

B BAD I=I

Inode I contains block number B with a number lower than the number of the first data block in the file system, or greater than the number of the last block in the file system. This error condition invokes the EXCESSIVE BAD BLKS error condition in Phase 1 if inode I has too many block numbers outside the file system range. This error condition always invokes the BAD/DUP error condition in Phase 2 and Phase 4. See Section 4.3.4.

EXCESSIVE BAD BLKS I=I (CONTINUE?)

There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system, or greater than the number of last block in the file system associated with inode I (Section 4.3.4).

Possible responses to the CONTINUE? prompt are:

YES Ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system.

NO Terminate the program.

B DUP I=I

Inode I contains block number B which is already claimed by another inode. This error condition invokes the EXCESSIVE DUP BLKS error condition in Phase 1 if inode I has too many block numbers claimed by other inodes. This error condition always invokes Phase 1B and the BAD/DUP error condition in Phase 2 and Phase 4 (Section 4.3.3).

EXCESSIVE DUP BLKS I=I (CONTINUE?)

There is more than a tolerable number (usually 10) of blocks claimed by other inodes (Section 4.4.3).

Possible responses to the CONTINUE? prompt are:

YES Ignore the rest of the blocks in this inode and continue checking with the next inode in this file system. This error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system.

NO Terminate the program.

DUP TABLE OVERFLOW (CONTINUE?)

An internal table in fsck containing duplicate block numbers has no more room. Recompile fsck with a larger value of DUPTBLSIZE.

Possible responses to the CONTINUE? prompt are:

YES Continue with the program. This error condition does not allow a complete check of the file system. A second run of fsck must be made to recheck this file system. If another duplicate block is found, this error condition repeats.

NO Terminate the program.

POSSIBLE FILE SIZE ERROR I=I

The inode I size does not match the actual number of blocks used by the inode. This is only a warning (Section 4.3.5).

DIRECTORY MISALIGNED I=I

The size of a directory inode is not a multiple of the size of a directory entry (usually 16). This is only a warning (Section 4.3.5).

PARTIALLY ALLOCATED INODE I=I (CLEAR?)

Inode I is neither allocated nor unallocated (Section 4.3.1).

Possible responses to the CLEAR? prompt are:

YES Deallocate inode I by zeroing its contents.

NO Ignore this error condition.

A.4 Phase 1B: Rescan for More Duplicates

When a duplicate block is found in the file system, the system is rescanned to find the inode that previously claimed that block. This section lists the error condition when the duplicate block is found.

B DUP I=I

Inode I contains block number B, which is already claimed by another inode. This error condition always invokes the BAD/DUP error condition in Phase 2. Inodes that have over-

lapping blocks can be determined by examining this error condition and the DUP error condition in Phase 1 (Section 4.3.3).

A.5 Phase 2: Check Path Names

This phase removes directory entries pointing to inodes with error conditions from Phase 1 and Phase 1B. This section lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes.

ROOT INODE UNALLOCATED. TERMINATING.

The root inode (usually inode number 2) has no allocate mode bits. This should never happen. The program terminates (Section 4.3.1).

ROOT INODE NOT DIRECTORY (FIX?)

The root inode (usually inode number 2) is not a directory inode (Section 4.3.1).

Possible responses to the FIX? prompt are:

YES Make the root inode's type a directory. If the root inode's data blocks are not directory blocks, a very large number of error conditions are produced.

NO Terminate the program.

DUPS/BAD IN ROOT INODE (CONTINUE?)

Phase 1 or Phase 1B found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system (Sections 4.3.3 and 4.3.4).

Possible responses to the CONTINUE? prompt are:

YES Ignore the DUPS/BAD error condition in the root inode and attempt to continue to run the file system check. If the root inode is not correct, this results in a large number of other error conditions.

NO Terminate the program.

I OUT OF RANGE I=I NAME=F (REMOVE?)

A directory entry F has an inode number I which is greater than the end of the inode list (Section 4.5).

Possible responses to the REMOVE? prompt are:

YES The directory entry F is removed.

NO Ignore this error condition.

UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T NAME=F (REMOVE?)

A directory entry F has an inode I without allocate mode bits. The owner O, mode M, size S, modify time T, and file name F are printed (Section 4.5).

Possible responses to the REMOVE? prompt are:

YES The directory entry F is removed.

NO Ignore this error condition.

DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE?)

Phase 1 or Phase 1B found duplicate blocks or bad blocks associated with directory entry F, directory inode I. The owner O, mode M, size S, modify time T, and directory name F are printed (Sections 4.3.3 and 4.3.4).

Possible responses to the REMOVE? prompt are:

YES The directory entry F is removed.

NO Ignore this error condition.

DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F (REMOVE?)

Phase 1 or Phase 1B have found duplicate blocks or bad blocks associated with directory entry F, inode I. The owner O, mode M, size S, modify time T, and file name F are printed (Sections 4.3.3 and 4.3.4).

Possible responses to the REMOVE? prompt are:

YES The directory entry F is removed.

NO Ignore this error condition.

A.6 Phase 3: Check Connectivity

This phase checks the directory connectivity seen in Phase 2. This section lists error conditions resulting from unreferenced directories, and missing or full _lost_+_found_ directories.

UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT?)

The directory inode I was not connected to a directory entry when the file system was checked. The owner O, mode M, size S, and modify time T of directory inode I are printed (Sections 4.5 and 4.3.2).

Possible responses to the RECONNECT? prompt are:

YES Reconnect directory inode I to the file system in the directory for lost files (usually _lost_+_found_). This invokes the _lost_+_found_ error condition in Phase 3 if there are problems connecting directory inode I to _lost_+_found_. This also invokes thee CONNECTED error condition in Phase 3 if the link was successful.

NO Ignore this error condition. This always invokes the UNREF error condition in Phase 4.

SORRY. NO _lost_+_found_ DIRECTORY

There is no _lost_+_found_ directory in the root directory of the file system; fsck ignores the request to link a directory in _lost_+_found_. This always invokes the UNREF error condition in Phase 4. Check access modes of _lost_+_found_ (_fsck_(_1_)).

SORRY. NO SPACE IN _lost_+_found_ _DIRECTORY_

There is no space to add another entry to the _lost_+_found_ directory in the root directory of the file system; fsck ignores the request to link a directory in _lost_+_found_. This always invokes the UNREF error condition in Phase 4. Remove unnecessary entries in _lost_+_found_ or make _lost_+_found_ larger (_fsck_(_1_)).

DIR I=I1 CONNECTED. PARENT WAS I=I2

This is an advisory message indicating that a directory inode I1 is successfully connected to the _lost_+_found_ directory. The parent inode I2 of the directory inode I1 is replaced by the inode number of the _lost_+_found_ directory (Sections 4.5 and 4.3.2).

A.7 Phase 4: Check Reference Counts

This phase checks the link count information seen in Phase 2 and Phase 3. This section lists error conditions resulting from unreferenced files, missing or full _lost_+_found_ directory, incorrect link counts for files, directories, or special files, unreferenced files and directories, bad and duplicate blocks in files and directories, and incorrect total free-inode counts.

UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT?)

Inode I was not connected to a directory entry when the file system was checked. The owner O, mode M, size S, and modify time T of inode I are printed (Section 4.3.2).

Possible responses to the RECONNECT? prompt are:

YES Reconnect inode I to the file system in the directory for lost files (usually _lost_+_found_). This invokes the _lost_+_found_ error condition in Phase 4 if there are problems connecting inode I to _lost_+_found_.

NO Ignore this error condition. This always invokes the CLEAR error condition in Phase 4.

SORRY. NO _lost_+_found_ DIRECTORY

There is no _lost_+_found_ directory in the root directory of the file system; fsck ignores the request to link a file in _lost_+_found_. This always invokes the CLEAR error condition in Phase 4. Check access modes of _lost_+_found_.

SORRY. NO SPACE IN _lost_+_found_ DIRECTORY

There is no space to add another entry to the _lost_+_found_ directory in the root directory of the file system; fsck

ignores the request to link a file in *lost+found*. This always invokes the clear error condition in Phase 4. Check size and contents of *lost+found*.

(CLEAR?)

The inode mentioned in the immediately previous error condition cannot be reconnected (Section 4.3.2).

Possible responses to the CLEAR? prompt are:

YES Deallocate the inode mentioned in the immediately previous error condition by setting its contents to zero.

NO Ignore this error condition.

LINK COUNT FILE I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST?)

The link count for the file inode I is X but should be Y. The owner O, mode M, size S, and modify time T are printed (Section 4.3.2).

Possible responses to the ADJUST? prompt are:

YES Replace the link count of file inode I with Y.

NO Ignore this error condition.

LINK COUNT DIR I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST?)

The link count for directory inode I is X but should be Y. The owner O, mode M, size S, and modify time T of directory inode I are printed (Section 4.3.2).

Possible responses to the ADJUST? prompt are:

YES Replace the link count of inode I with Y.

NO Ignore this error condition.

LINK COUNT F I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST?)

The link count for F inode I is X but should be Y. The name F, owner O, mode M, size S, and modify time T are printed (Section 4.3.2).

Possible responses to the ADJUST? prompt are:

YES Replace the link count of inode I with Y.

NO Ignore this error condition.

UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR?)

File inode I was not connected to a directory entry when the file system was checked. The owner O, mode M, size S, and modify time T of inode I are printed (Sections 4.3.2 and 4.5).

Possible responses to the CLEAR? prompt are:

YES Deallocate inode I by zeroing its contents.

NO Ignore this error condition.

UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR?)

Directory inode I was not connected to a directory entry when the file system was checked. The owner O, mode M, size S, and modify time T of inode I are printed (Section 4.3.2 and 4.5).

Possible responses to the CLEAR? prompt are:

YES Deallocate inode I by zeroing its contents.

NO Ignore this error condition.

BAD/DUP FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR?)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with file inode. The owner O, mode M, size S, and modify time T of inode I are printed (Sections 4.3.3 and 4.3.4).

Possible responses to the CLEAR? prompt are:

YES Deallocate inode I by setting its contents to zero.

NO Ignore this error condition.

BAD/DUP DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR?)

Phase 1 or Phase 1B have found duplicate blocks or bad blocks associated with directory inode I. The owner O, mode M, size S, and modify time T of inode I are printed (Sections 4.3.3 and 4.3.4).

Possible responses to the CLEAR? prompt are:

YES Deallocate inode I by setting its contents to zero.

NO Ignore this error condition.

FREE INODE COUNT WRONG IN SUPERBLK (FIX?)

The actual count of the free inodes does not match the count in the super-block of the file system (Section 4.2.4).

Possible responses to the FIX? prompt are:

YES Replace the count in the super-block by the actual count.

NO Ignore this error condition.

A.8 Phase 5: Check Free List

This section lists error conditions resulting from bad blocks in the free-block list, bad free-blocks count, duplicate blocks in the free-block list, unused blocks from the file system not in the free-block list, and an incorrect total free-block count.

EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE?)

The free-block list contains more than a tolerable number (usually 10) of blocks with a value of less than the first data block in the file system or greater than the last block in the file system (Sections 4.2.2 and 4.3.4).

Possible responses to the CONTINUE? prompt are:

YES Ignore the rest of the free-block list and continue the execution of fsck. This error condition always invokes the BAD BLKS IN FREE LIST error condition in Phase 5.

NO Terminate the program.

EXCESSIVE DUP BLKS IN FREE LIST (CONTINUE?)

The free-block list contains more than a tolerable number (usually 10) of blocks claimed by inodes or earlier parts of the free-block list (Section 4.2.2 and 4.3.3).

Possible responses to the CONTINUE? prompt are:

YES Ignore the rest of the free-block list and continue the execution of fsck. This error condition always invokes the DUP BLKS IN FREE LIST error condition in Phase 5.

NO Terminate the program.

BAD FREEBLK COUNT

The count of free blocks in a free-list block is greater than 50 or less than zero. This error condition always invokes the BAD FREE LIST condition in Phase 5 (Section 4.2.2).

X BAD BLKS IN FREE LIST

X blocks in the free-block list have a block number lower than the first data block in the file system or greater than the last block in the file system. This error condition always invokes the BAD FREE LIST condition in Phase 5 (Sections 4.2.2 and 4.3.4).

X DUP BLKS IN FREE LIST

X blocks claimed by inodes or earlier parts of the free-list block were found in the free-block list. This error condition always invokes the BAD FREE LIST condition in Phase 5 (Sections 4.2.2 and 4.3.3).

X BLK(S) MISSING

X blocks unused by the file system were not found in the free-block list. This error condition always invokes the BAD FREE LIST condition in Phase 5 (Section 4.2.2).

FREE BLK COUNT WRONG IN SUPERBLOCK (FIX?)

The actual count of free blocks does not match the count in the super-block of the file system (Section 4.2.3).

Possible responses to the FIX? prompt are:

YES Replace the count in the super-block with the actual count.

NO Ignore this error condition.

BAD FREE LIST (SALVAGE?)

Phase 5 has found bad blocks in the free-block list, duplicate blocks in the free-block list, or blocks missing from the file system (Sections 4.2.2, 4.3.3, and 4.3.4).

Possible responses to the SALVAGE? prompt are:

YES Replace the actual free-block list with a new free-block list. The new free-block list will be ordered to reduce time spent by the disk waiting for the disk to rotate into position.

NO Ignore this error condition.

A.9 Phase 6: Salvage Free List

This phase checks the free block list reconstruction. This section lists error conditions resulting from the blocks-to-skip and blocks-per-cylinder values.

DEFAULT FREE-BLOCK LIST SPACING ASSUMED

This is an advisory message indicating the blocks-to-skip is greater than the blocks-per-cylinder, the blocks-to-skip is less than one, the blocks-per-cylinder is less than one, or the blocks-per-cylinder is greater than 500. The default values of nine blocks-to-skip and 400 blocks-per-cylinder are used (_fsck_(_1_)).

A.10 Cleanup

Once a file system has been checked, cleanup functions are performed. This section lists advisory messages about the file system and modify status of the file system.

X FILES Y BLOCKS Z FREE

This is an advisory message indicating that the file system checked contained X files using Y blocks, leaving Z blocks free in the file system.

*****BOOT ZEUS (NO SYNC!)*****

This is an advisory message indicating that a mounted file system or the root file system has been modified by fsck. If ZEUS is not rebooted immediately, the work done by fsck may be undone by the in-core copies of tables ZEUS keeps.

*****FILE SYSTEM WAS MODIFIED*****

This is an advisory message indicating that the current file system was modified by fsck. If this file system is mounted or is the current root file system, fsck must be halted and ZEUS rebooted. If ZEUS is not rebooted immediately, the work done by fsck may be undone by the in-core copies of tables ZEUS keeps.

INDEX OF MESSAGES

(Alphabetically within each section)

INTIALIZATION

BAD -t OPTION 15
C OPTION? 15
CANNOT READ: BLK B (CONTINUE?) 17
CANNOT SEEK: BLK B (CONTINUE?) 17
CANNOT WRITE: BLK B (CONTINUE?) 18
CAN'T CREATE F 17
CAN'T GET MEMORY 16
CAN'T OPEN CHECKLIST FILE: F 16
CAN'T OPEN F 17
CAN'T STAT F 16
CAN'T STAT ROOT 16
F IS NOT A BLOCK OR CHARACTER DEVICE 16
INCOMPATIBLE OPTIONS: -n and -s 16
INVALID -s ARGUMENT, DEFAULTS ASSUMED 16
SIZE CHECK: FSIZE X ISIZE Y 17

PHASE 1: CHECK BLOCKS AND SIZES

B BAD I=I 19
B DUP I=I 19
DIRECTORY MISALIGNED I=I 20
DUP TABLE OVERFLOW (CONTINUE?) 20
EXCESSIVE BAD BLKS I=I (CONTINUE?) 19
EXCESSIVE DUP BLKS I=I (CONTINUE?) 21
LINK COUNT TABLE OVERFLOW (CONTINUE?) 21
PARTIALLY ALLOCATED INODE I=I (CLEAR?) 21
POSSIBLE FILE SIZE ERROR I=I 20
UNKNOWN FILE TYPE I=I (CLEAR?) 18

PHASE 1B: RESCAN FOR MORE DUPS

B DUP I=I 21

PHASE 2: CHECK PATH-NAMES

DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F
(REMOVE?) 23
DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F
(REMOVE?) 23
DUPS/BAD IN RROOT INODE (CONTINUE?) 22
I OUT OF RANG I=I NAME=F (REMOVE?) 23
ROOT INODE NOT DIRECTORY (FIX?) 22
ROOT INODE UNALLOCATED TERMINATING 22

INDEX OF MESSAGES (continued)

UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T
NAME=F (REMOVE?) 23

PHASE 3: CHECK CONNECTIVITY

DIR I-I1 CONNECTED PARENT WAS I=I2 25
SORRY. NO SPACE IN lost+found DIRECTORY 24
SORRY. NO lost+found DIRECTORY 24
UNREF DIRE I=I OWNER=O MODE=MM SIZE=S MTIME=T
(RECONNECT?) 24

PHASE 4: CHECK REFERENCE COUNTS

BAD/DUP DIR I=I OWNER=O MODE=M SIZE=S MTIME=T
(CLEAR?) 28
BAD/DUP FILE I=I OWNER=O MODE=M SIZE=S MTIME=T
(CLEAR?) 27
(CLEAR?) 26
FREE INODE COUNT WRONG IN SUPERBLK (FIX?) 28
LINK COUNT DIR I=I OWNER=O MODE=M SIZE=S MTIME=T
COUNT=X SHOULD BE Y (ADJUST?) 26
LINK COUNT FILE I=I OWNER=O MODE=M SIZE=S MTIME=T
COUNT=X SHOULD BE Y (ADJUST?) 26
LINK COUNT F I=I OWNER=O MODE=M SIZE=S MTIME=T
COUNT=X SHOULD BE Y (ADJUST?) 27
SORRY. NO SPACE IN lost+found DIRECTORY 25
SORRY. NO lost+found DIRECTORY 25
UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T
(CLEAR?) 27
UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T
(CLEAR?) 27
UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T
(RECONNECT?) 25

PHASE 5: CHECK FREE LIST

BAD FREE LIST (SALVAGE?) 30
BAD FREEBLK COUNT 29
EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE?) 28
EXCESSIVE DUP BLKS IN FREE LIST (CONTINUE?) 29
FREE BLK COUNT WRONG IN SUPERBLOCK (FIX?) 30

INDEX OF MESSAGES (continued)

X BAD BLKS IN FREE LIST 29
X BLK(S) MISSING 30
X DUP BLKS IN FREE LIST 29

PHASE 6: SALVAGE FREE LIST

DEFAULT FREE-BLOCK LIST SPACING ASSUMED 30

CLEANUP

*****BOOT ZEUS (NO SYNC!)***** 31
*****FILE SYSTEM WAS MODIFIED***** 31
X FILES Y BLOCKS Z FREE 31

LEARN

COMPUTER-AIDED INSTRUCTION ON ZEUS

* This information is based on an article originally written by Brian W. Kernighan and Michael E. Lesk, Bell Laboratories.

PREFACE

This document describes the LEARN program and its seven Computer-Aided Instruction (CAI) scripts that provide lessons on the ZEUS Operating System. Since LEARN is a self-explanatory program, this document gives the theoretical background instead of detailed instructions on how to use it. The purpose of this document is to guide people preparing programs similar to LEARN, not to assist them in learning basic computer skills.

Section 1 contains a general introduction to LEARN and Section 2 states educational assumptions and design. The topic of each script appears in Section 3, and Section 4 describes how the LEARN program interprets the scripts. Conclusions about the LEARN experience are in Section 5.

TABLE OF CONTENTS

SECTION 1 INTRODUCTION 4

SECTION 2 EDUCATIONAL ASSUMPTIONS AND DESIGN 5

2.1 Theoretical Assumptions 5
2.2 Types of Lessons 6
2.3 Sample Lesson Display 6
2.4 Track Levels 7

SECTION 3 SCRIPTS 9

3.1 General Information 9
3.2 First-Time User Script 9
3.3 Basic File Handling Script 9
3.4 Context Editor Script 10
3.5 Advanced File Handling Script 10
3.6 Eqn Language Script 10
3.7 -ms Script 10
3.8 C Language Script 10

SECTION 4 THE SCRIPT INTERPRETER 11

4.1 General Information 11
4.2 File Structure 11
4.3 Requirements 12
4.4 Sequence of Events 12
4.5 Interpreted Script 13

SECTION 5 CONCLUSIONS 18

SECTION 1

INTRODUCTION

The system that teaches computer skills has two main parts: a driver called LEARN that interprets the scripts, and the scripts themselves. At present, there are seven Computer Aided Instruction (CAI) scripts:

1. first-time user introduction
2. basic file handling commands
3. ZEUS text editor (ed)
4. advanced file handling commands
5. eqn language for mathematical typing
6. the -ms macro package for document formatting
7. C programming language

The advantages of CAI scripts include the following:

- students are forced to perform the exercises
- students receive immediate feedback and confirmation of progress
- students progress at their own rate
- no schedule requirements are imposed
- lessons can be individually improved
- the computer is accessible to the student at the student's convenience.
- usage of high technology motivates students and maintains management interest

Since there is no one the student can question, CAI is comparable to a textbook, lecture series, or taped course rather than to a seminar. CAI has been used for many years in a variety of educational areas. Using the computer as a self-teaching device offers unique advantages; the skills developed to go through the script are exactly those needed to operate the computer; therefore, there is no wasted effort.

SECTION 2

EDUCATIONAL ASSUMPTIONS AND DESIGN

2.1 Theoretical Assumptions

The best way to teach people how to do something is to have them do it. Scripts should not contain long explanations, but instead should frequently ask the student to do a task. Teaching is always by example; the typical lesson shows a small example of some technique and then asks the student to either repeat that example or produce a variation of it. All lessons are intended to be easy enough so that most students get most questions right, reinforcing the desired behavior.

After each correct response, the computer congratulates the student and indicates the lesson number that has just been completed, permitting the student to restart the script after that lesson. If the answer is wrong, the student is offered a chance to repeat the lesson.

It is assumed that there is no foolproof way to determine if the student truly "understands" what he or she is doing; the LEARN scripts measure performance, not comprehension.

The computer provides an immediate check of the correctness of what the student does. Unlike many CAI scripts, these scripts provide few facilities for dealing with wrong answers. In practice, if most of the answers are not right, the script is a failure. The solution to the problem of excessive student error is to provide a new, easier script. Anticipating possible wrong answers is an endless job; it is easier and better to provide a simpler script.

LEARN also provides a mechanical check on performance. If a student is unable to complete one lesson, that should not prevent access to the rest. The current version of LEARN allows the student to skip a lesson that he or she cannot pass. For example, a "no" answer to the "Do you want to try again?" question in Section 2-3 causes the program to go on to the next lesson.

There are valid objections to these assumptions, since some students object to not understanding what they are doing. Since writing a CAI script is more tedious than writing ordinary manuals, there are always alternatives to the scripts as a way of learning.

2.2 Types of Lessons

Most lessons are one of three types. The simplest lesson asks for a yes or no answer to a question. The student is given a chance to experiment before replying, and the lesson checks for the correct reply. Problems of this form are used sparingly.

The second type asks for a word or number as an answer. For example, a lesson on files might say

> How many files are there in the current directory?
> Type "answer N," where N is the number of files.

The student is expected to enter (perhaps after experimenting) a response similar to

```
answer 17
```

The idea of a substitutable argument (replacing N by 17) is difficult for nonprogrammer students, so the first few such lessons require special attention.

The third type of lesson is open-ended. A task is set for the student, appropriate parts of the input or output are monitored, and the student types:

```
ready
```

when the task is done.

2.3 Sample Lesson Display

The following sample is from the script that teaches file handling. It incorporates the open-ended and the word or number answer types of lessons. Most LEARN lessons are of this form. Student responses are shown in italics. The "$" is the system prompt.

```
A file can be printed on your terminal
by using the "cat" command. Just say
"cat file" where "file" is the file name.
For example, there is a file named
"food" in this directory. List it
by saying "cat food"; then type "ready".
$ cat food
  this is the file
  named food.
$ ready
```

```
Good.  Lesson 3.3a (1)

Of course, you can print any file with "cat".
In particular, it is common to first use
"ls" to find the name of a file and then "cat"
to print it.  Note the difference between
"ls", which tells you the name of the file,
and "cat", which tells you the contents.
One file in the current directory is named for
a President.  Print the file, then type "ready".
$ cat President
cat: can't open President
$ ready

Sorry, that's not right.  Do you want to try again? yes
Try the problem again.
$ ls
.ocopy
X1
roosevelt
$ cat roosevelt
  this file is named roosevelt
  and contains three lines of
  text.
$ ready

Good.  Lesson 3.3b (0)

The "cat" command can also print several files
at once.  In fact, it is named "cat" as an abbreviation
for "concatenate"....
```

2.4 Track Levels

In the files and editor scripts there are three tracks, differing in degrees of difficulty. The fastest script (sequence of lessons), roughly the bulk and speed of a typical tutorial manual, should be adequate for review and for well-prepared students. The next track, intended for most users, is about twice as long. The third and slowest track, which is often three or four times the length of the fast track, is intended for the most basic instruction. For example, the fast track presents an idea and asks for a variation on the example shown. The normal track first asks the student to repeat the example that was shown before attempting a variation. The lesson in Section 2.3 is from the third track.

The LEARN driver combines lessons in different ways to produce scripts in each track. For example, the fast track is produced by skipping lessons from the slower track. The

driver can also switch tracks, depending on the number of correct answers the student has given for the last few lessons.

SECTION 3

SCRIPTS

3.1 General Information

The present scripts follow a three-track theory. Care must be taken in lesson construction to see that every necessary fact is presented in every possible path throughout the scripts. In addition, it is desirable that every lesson have alternate successors to deal with student errors.

There are some preliminary skills that the student must know before any scripts can be tried. In particular, the student must know how to connect a ZEUS system, set the terminal properly, log in, and execute simple commands (for example LEARN itself). In addition, the character erase and line kill conventions (control-h and control-x) should be known. The student will need assistance for a few minutes to gain familiarity with these skills.

In existing scripts, the first few lessons are devoted to checking prerequisites. For example, before the student is allowed to proceed through the editor script, the script verifies that the student understands files and is able to type. Anyone proceeding through the scripts should get correct answers; otherwise, the system will be unsatisfactory both because the wrong habits are being learned and because the scripts make little effort to deal with wrong answers. Therefore, unprepared students should not be encouraged to continue with scripts.

3.2 First-Time User Script

The first-time user script covers a few important features of the system in very brief lessons. Here, I/O redirection, pipes, make files, the C compiler, and the ZEUS text editor (_ed_) are introduced.

3.3 Basic File Handling Script

It is assumed that the user of this script has basic knowledge of Script 1; it teaches the student about the _ls_, _cat_, _mv_, _rm_, _cp_ and _diff_ commands. It also deals with the abbreviation characters *, ?, and [ ] in file names. It does not cover pipes or I/O redirection, nor does it present the many options of the _ls_ command.

3.4 Context Editor Script

This script trains students in the use of the ZEUS context editor, *ed*, a sophisticated editor using regular expressions for searching. All editor features except encryption, mark names, and ; in addressing are covered.

3.5 Advanced File Handling Script

The advanced file handling script, assuming the basic file handling script as a prerequisite, deals with *ls* options, I/O diversion, pipes, and supporting programs like *pr*, *wc*, *tail*, *spell*, and *grep*.

3.6 *Eqn* Language Script

This script covers the *eqn* language for typing mathematics and must be run on a terminal capable of printing mathematical symbols (for instance the DASI 300 and similar Diablo-based terminals). Most advanced lessons provide additional practice for students who are having trouble in the basic track.

3.7 -*ms* Script

The -*ms* script for formatting macros is a short, one-track script. However, the linear style of a single LEARN script is inappropriate for the macros, since the macro package is composed of many independent features, and few users need all of them.

3.8 C Language Script

The script on the language C has been partially converted to follow the order of presentation in *The C Programming Language*. The C script was never intended to teach C; rather it is a series of exercises for which the computer provides checking and a suggested solution.

SECTION 4

THE SCRIPT INTERPRETER

4.1 General Information

The LEARN program interprets scripts. It provides facilities to capture student responses and their effects, and simplifies the job of passing control to and recovering control from the student. This section describes the operation and use of the driver program, and indicates what is required to produce a new script. Readers interested only in the existing scripts should skip this section.

4.2 File Structure

The file structure used by LEARN is shown below. There is one parent directory named _lib_ containing the script data. Within this directory are subdirectories, one for each subject where a course is available, one for logging (named log), and one where user subdirectories are created (named play). The subject directory contains master copies of all lessons, plus any supporting material for that subject. In a given subdirectory, each lesson is a single text file. Lessons are usually named systematically; the file that contains lesson _n_ is called _Ln_.

```
lib

     play
          student1
               files for student1...
          student2
               files for student2...

     files
          L0.1a     lessons for files course
          L0.1b
          ...

     editor
          ...

     (other courses)

     log
```

Directory Structure for LEARN

When LEARN is executed, it makes a private directory for the user to work in, within the LEARN portion of the file system. A fresh copy of all the files used in each lesson is usually made by the lesson script each time a student starts a lesson. The student directory is deleted after each session; any permanent records must be kept elsewhere.

4.3 Requirements

Each lesson must contain the following basic items:

- the text of the lesson
- the set-up commands to be executed before the user gets control
- the data, if any, that the user is supposed to edit, transform, or otherwise process
- the evaluating commands to be executed after the user has finished the lesson, which decide whether the answer is right
- a list of possible successor lessons

LEARN minimizes the work of bookkeeping and installation, so that most of the effort involved in script production is in planning lessons, writing tutorial paragraphs, and coding tests of student performance.

4.4 Sequence of Events

LEARN first creates the working directory. Then, for each lesson, LEARN reads the text for the lesson and processes it a line at a time. The lines in the text are commands to the text interpreter to print something, to create a files, or to test something, text to be printed or put in a file, and other lines that are sent to the shell to be executed. One line in each lesson turns control over to the user, who can run any ZEUS command. The user mode terminates when the user types yes, ok, no, ready, or answer. At this point, the user's work is tested; if the lesson is passed, a new lesson is selected; if not, the old one is repeated.

4.5 Interpreted Script

To illustrate the flow of LEARN, the sample script from Section 2.3 is interpreted here.

Lines that begin with # are commands to the learn script interpreter. For example,

    #print

causes printing of any text that follows, up to the next line that begins with a sharp. The command

    #print file

prints the contents of _file_; it is the same as _cat file_. Both forms of #_print_ have the added property that if a lesson is failed, the #_print_ is not executed the second time; this avoids annoying the student by repeating the preamble to a lesson. The command

    #create file name

creates a file of the specified name and copies any subsequent text up to a # in the file. This creates and initializes working files and reference data for the lessons. The command

    #user

gives control to the student; each line typed is passed to the shell for execution. The #_user_ mode is terminated when the student types one of the special keywords _yes_, _ok_, _no_, _ready_, or _answer_. At that time, the driver resumes interpretation of the script.

The _yes_ and _no_ responses return control to the script, where the answer can be evaluated with #_match_. Since _ok_ is an alias for _yes_, the script writer can also use #_match_ _ok_ when an indication to proceed with the course is the only response needed.

The _ready_ response returns control to the script but cannot be evaluated with #_match_. Instead, the user's previous respnses are evaluated in some way. The _answer_ response prepares for evaluation of the answer given. For instance, if the correct answer is _3_, and the user responds with _answer_ _3_ then #_match3_ is used in the script to process that response. Anything the student types between the commands

```
#copyin
#uncopyin
```

is copied onto a file called *.copy*. This allows for interrogation of the student's responses upon regaining control. Between the commands

```
#copyout
#uncopyout
```

any material typed by the student for any program is copied to the file *.ocopy*. This allows interrogation of the effect of what the student typed.

Normally the student's input and the script commands are fed to the ZEUS command interpreter (the shell) one line at a time. A sequence of editor commands does not work, since the input to the editor must be handed to the editor, not to the shell. Accordingly, the material between the commands #*pipe* and #*unpipe* is fed continuously through a pipe so that such sequences work. If *copyout* is also desired, the copyout brackets must include the pipe brackets.

There are several commands for setting status after the student has attempted the lesson.

```
#cmp file1 file2
```

is an in-line implementation of *cmp* that compares two files for identity. Following the command

```
#match stuff
```

the last line of the student's input is compared to *stuff*, and the success or fail status is set according to this comparison. Extraneous things like the word *answer* are stripped before the comparison is made. There can be several #*match* lines; this provides a convenient mechanism for handling multiple "right" answers. Any text up to a # on subsequent lines after a successful #*match* is printed, as shown next.

```
#print
What command will move the current line
to the end of the file? Type
"answer COMMAND", where COMMAND is the command.
#copyin
#user
#uncopyin
#match m$
#match .m$
"m$" is easier.
```

```
#log
#next
63.ld 10
```

```
#bad stuff
```

This is similar to #*match*, except that it corresponds to specific failure answers; this produces hints for particular wrong answers that have been anticipated by the script writer. The commands

```
#succeed
#fail
```

print a message upon success or failure (as determined by some other mechanism).

When the student types one of the "commands" *yes*, *ok*, *no*, *ready*, or *answer*, the driver terminates the #*user* command, and evaluation of the student's work can begin. This can be done either by the built-in commands, such as #*match* and #*cmp*, or by status returned by normal ZEUS commands, typically *grep* and *test*. The last command should return status true (0) if the task is done successfully and false (nonzero) otherwise; this status return tells the driver whether or not the student has successfully passed the lesson.

Performance can be logged:

```
#log file
```

writes the date, lesson, user name and speed rating, and a success/failure indication on *file*. The command

```
#log
```

by itself writes the logging information in the logging directory within the LEARN hierarchy, and is the normal form. The commands

```
# cleanup
# nocleanup
```

are for directing the LEARN driver to clean up or ignore the temporary files created in a lesson. By default, learn cleans out the temporary files after each lesson. Specifically, all files that begin with a lowercase letter and are not ".c" files are deleted before the next lesson. The #nocleanup directive enables following lessons to depend on files already created or changed by the user, and #cleanup

restores the default action at any time. The command

    #next

is followed by a few lines, each with a successor lesson name and an optional speed rating on it. A typical set reads

```
25.1a   10
25.2a   5
25.3a   2
```

indicating that unit 25.1a is a suitable follow-on lesson for students with a speed rating of 10 units, 25.2a for student with a speed rating of 5 units, and 25.3a for students with a speed rating of 2 units. Speed ratings are maintained for each session per student; the rating is increased by one each time the student gets a lesson right and decreased by four each time the student gets a lesson wrong. Thus, the driver maintains a level at which the users get 80% right answers. The maximum rating is limited to 10, and the minimum is zero. The initial rating is zero unless the student specifies a different rating when starting a session.

If the student passes a lesson, a new lesson is selected, and the process repeats. If the student fails, a false status is returned, and the program reverts to the previous lesson and tries another alternative. If it cannot find another alternative, it skips forward a lesson.

If the student is unable to answer one of the exercises correctly, the driver searches for a previous lesson with a set of alternatives as successors (following the #<u>next</u> line). The program selects an alternative different from the one tried in the previous lesson.

Sophisticated scripts can be written to evaluate the student's speed of response, estimate the subjective merits of the answer, or to provide detailed analysis of wrong answers.

The driver program depends heavily on features of ZEUS that are not available on many other operating systems. Although some parts of LEARN might be transferable to other systems, some generality will be lost.

SECTION 5

CONCLUSIONS

The following are observations about nonprogrammers using LEARN.

A novice must have assistance with the mechanics of communicating with the computer to get through the first or second lesson. Once the first few lessons are passed, people can proceed on their own. Most students enjoy the system, and motivation matters a great deal.

The terminology used in the first few lessons is obscure to those inexperienced with computers. It would help if there were a low-level reference card to supplement the existing manual and reference card. The concept of "substitutable argument" is hard to grasp and requires help.

It takes an hour or two for a novice to get through the script on file handling. The total time for a novice to create new files and manipulate old ones is a few days, with perhaps half of each day spent on the machine.

The normal way of proceeding has been to have students in the same room with someone who knows ZEUS and the scripts. Thus, the student is not brought to a halt by difficult questions. The burden on the counselor is much lower than that on a teacher of a course. The students should be encouraged to proceed with instruction immediately prior to their actual use of the computer. They should exercise the scripts on the same computer and the same kind of terminal that they will later use for their real work, and their first few jobs with the computer should be relatively easy ones. Also, both training and initial work should take place on days when the ZEUS hardware and software are working reliably. Students are frustrated by machine downtime; when nothing is happening, it takes some sophistication and experience to distinguish among an infinite loop, a slow but functioning program, a program waiting for the user, or a broken machine.

One disadvantage of training with LEARN is that students come to depend completely on the CAI system and do not try to read manuals or use other learning aids. This is unfortunate, not only because of the increased demands for completeness and accuracy of the scripts but because the scripts do not cover all of the ZEUS system. New users should have manuals (appropriate for their level) and read them; the scripts ought to be altered to recommend suitable documents and to urge students to read them.

From the student's viewpoint, the most serious difficulty is that there are lessons that simply cannot be passed. Sometimes this is due to poor explanations, but just as often it is some error in the lesson itself, a wrong setup, a missing file, an invalid test for correctness, or some system facility that does not work on the local system as on the development system. It takes knowledge and a certain healthy arrogance on the part of the users to recognize that the fault is not theirs. Permitting the student to continue with the next lesson regardless does alleviate this, and the logging facilities make it easy to watch for lessons that no one can pass.

The biggest problem with some scripts, notably eqn, is that they are very slow. Another potential problem is that it is possible to break LEARN by pushing interrupt at the wrong time, by removing critical files, or any number of similar slips. The defenses against such problems have steadily been improved to the point where most students should not notice difficulties.

One area is more fundamental: LEARN currently does not allow ZEUS global commands to be executed. The most obvious is cd, which changes to another directory. The prospect of a student who is learning about directories moving to some random directory and removing files has prevented lessons on cd.

LEX

A LEXICAL ANALYZER GENERATOR *

USER GUIDE

* This information is based on an article originally written by M. E. Lesk and E. Schmidt, Bell Laboratories.

PREFACE

This document is a reference manual for Lex, a lexical analyzer generator that accepts string matching specifications and produces a program in a general-purpose language. The reader is assumed to have some experience with Lex before using this document.

Sections 1-6 give an introduction to Lex and describe its internal rules. Hints for compiling Lex appear in Section 7. Section 8 describes the interface between Lex and Yacc (yet another compiler-compiler). Examples of Lex are shown in Section 9, and Section 10 gives ways to define different Lex environments. Sections 11-13 summarize the Lex character set, source format, and cautions.

TABLE OF CONTENTS

SECTION 1 INTRODUCTION 5

SECTION 2 LEX SOURCE 7

SECTION 3 LEX REGULAR EXPRESSIONS 8

3.1 Introduction 8
3.2 Operators 8
3.3 Character Classes 9
3.4 Arbitrary Character 10
3.5 Optional Expressions 10
3.6 Repeated Expressions 10
3.7 Alternation and Grouping 11
3.8 Context Recognition 11
3.9 Repetitions and Definitions 12
3.10 Segmented Separator 12

SECTION 4 LEX ACTIONS 13

4.1 Introduction 13
4.2 Regular Routines 13
4.3 Input/Output Routines 16
4.4 Library Routines 16

SECTION 5 AMBIGUOUS SOURCE RULES 18

SECTION 6 LEX SOURCE DEFINITIONS 21

SECTION 7 COMPILING LEX 23

SECTION 8 LEX AND YACC 24

SECTION 9 EXAMPLES 25

9.1 Copy with Simple Arithmetic Changes ... 25
9.2 Statistical Accumlations 25

TABLE OF CONTENTS (continued)

SECTION 10 LEFT CONTEXT SENSITIVITY 27

SECTION 11 CHARACTER SET 30

SECTION 12 SUMMARY OF SOURCE FORMAT 31

SECTION 13 CAUTIONS 33

SECTION 1

INTRODUCTION

Lex is a program generator for lexical processing of character input streams. It accepts user-supplied specifications for character string matching and produces a program in a general-purpose language (yylex). This program recognizes regular expressions in an input stream and performs the specified actions for each expression as it is detected. This entire process is shown as follows:

```
Source ->     Lex       -> yylex


Input ->      yylex     -> Output
```

Lex is not a complete language, but rather a generator representing a new language feature that can be added to different programming languages, called host languages. Just as general-purpose languages produce code to run on different computer hardware, Lex writes code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application can be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C.

Code needed for task completion, except expression-matching, is supplied by the user. This can include code written by other generators. A high-level language is provided to write the string expressions to be matched, while the user's freedom to write actions is unimpaired. This allows the use of several string manipulation languages.

For example, to delete from the input all blanks or tabs at the ends of lines, all that is required is:

```
%%
[ \t]+$  ;
```

This program contains a %% delimiter to mark the beginning of the rules and one rule that matches one or more instances of the characters blank or tab (written \t for visibility) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one

or more ..."; the $ indicates "end of line." No action is specified, so the program generated by Lex (yylex) ignores these characters. Everything else is copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$    ;
[ \t]+     printf(" ");
```

This source scans for both rules at once and executes the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule matches all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Additional programs can be added easily to programs written by Lex. Lex can also be used with a parser generator such as Yacc to perform the lexical analysis phase. When used as a preprocessor for a later parser generator, Lex partitions the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown below

```
          lexical           grammar
           rules             rules
           (Lex)             (Yacc)
Input ->    yylex      ->     yyparse  ->  Parsed input
```

Yacc users realize that the name yylex is what Yacc expects its lexical analyzer to be named, so the use of this name by Lex simplifies interfacing.

The time a Lex program takes to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules that include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the program generated by Lex.

Lex is not limited to source that can be interpreted on the basis of one-character look-ahead. For example, if there are two rules, one looking for ab and another for abcdefg, and the input stream is abcdefh, Lex recognizes ab and leaves the input pointer just before cd. Such backup is more costly than the processing of simpler languages.

SECTION 2

LEX SOURCE

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions and the user subroutines are often omitted.

The *rules* represent the user's control decisions. They are in the form of a table, in which the left column contains *regular expressions* (Section 3) and the right column contains *actions*--program fragments to be executed when the expressions are recognized. The second %% is optional, but the first is required to mark the beginning of the rules.

To change a number of words from British spelling to American spelling, start with Lex rules such as:

```
colour          printf("color");
mechanise       printf("mechanize");
petrol          printf("gas");
```

These rules are not quite enough, since the word *petroleum* would become *gaseum*; a way of dealing with this will be described in Sections 4 and 5.

An individual rule such as

```
integer   printf("found keyword INT");
```

is used to look for the string *integer* in the input stream; it prints the message "found keyword INT" whenever it appears. In this example, the host procedural language is C and the C library function *printf* prints the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces.

SECTION 3

LEX REGULAR EXPRESSIONS

3.1 Introduction

A regular expression specifies a set of strings to be matched. It contains text characters that match the corresponding characters in the strings being compared and operator characters that specify repetitions, choices, and other features.

The letters of the alphabet and the digits are always text characters; thus, the regular expression

```
integer
```

matches the string integer wherever it appears, and the expression

```
a57D
```

looks for the string a57D.

3.2 Operators

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

When operators are used as text characters, an escape must be used. The quotation mark operator (") indicates that any characters contained between a pair of quotes should be treated as text characters. Thus,

```
xyz"++"
```

matches the string xyz++ when it appears. A part of a string can be quoted.

Ordinary text characters can be included within quotes. For example, the expression

```
"xyz++"
```

is the same as the one above. The practice of quoting every nonalphanumeric character being used as a text character eliminates the need to remember the list of current operator characters.

An operator character can also be turned into a text character by preceding it with \, as in the command

```
xyz\+\+
```

which is another (less readable) equivalent of the above expressions.

Another use of the quoting mechanism is to insert a blank into an expression. Normally, blanks or tabs end a rule. Any blank character not contained within brackets ([]) must be quoted.

Several normal C escapes with \ are recognized: \n is new line, \t is tab, and \b is backspace. To enter \ itself, use \\. Since a new line is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Characters other than blank, tab, new line, and the operator characters are always text characters.

3.3 Character Classes

Classes of characters can be specified using the operator pair []. The construction [*abc*] matches a single character, which can be *a*, *b*, or *c*. When enclosed in brackets, most characters lose any special meaning (they are not treated as operators). The only exceptions are \, -, and ^.

The - character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underline. Ranges can be given in either order. Using - between any pair of characters that are not both uppercase letters, both lowercase letters, or both digits causes a warning message. If a minus sign is included in a character class, it should be first or last; thus,

```
[-+0-9]
```

matches all the digits and the two signs.

The ^ operator matches the complement of the subsequent character string. Thus,

```
[^abc]
```

matches all characters except a, b, or c, including all special or control characters. The expression

```
[^a-zA-Z]
```

matches any character that is not a letter.

The ^ operator must immediately follow the left bracket.

The \ character provides the usual escapes within character class brackets.

3.4 Arbitrary Character

To match almost any character, use the operator character

```
.
```

which is the class of all characters except new line. Escaping into octal is possible, although nonportable, with the command

```
[\40-\176]
```

which matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

3.5 Optional Expressions

The operator ? indicates an optional element of an expression. Thus,

```
ab?c
```

matches either ac or abc.

3.6 Repeated Expressions

Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive a characters, including zero; while

```
a+
```

is one or more instances of a. For example,

```
[a-z]+
```

is all strings of lowercase letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

3.7 Alternation and Grouping

The operator | indicates alternation:

```
(ab|cd)
```

matches either ab or cd. Parentheses are used for grouping, although they are not necessary on the outside level. For example,

```
ab|cd
```

is sufficient for the previous command.

Parentheses more commonly occur in more complex expressions, such as:

```
(ab|cd+)?(ef)*
```

which matches such strings as abefef, efefef, cdef, or cddd, but not abc, abcd, or abcdef.

3.8 Context Recognition

Lex recognizes a small amount of surrounding context. The / operator indicates trailing context. The expression

```
ab/cd
```

matches the string ab, but only if followed by cd. Thus,

```
ab$
```

is the same as

```
ab/\n
```

The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression is only matched at the beginning of a line (after a new line

character, or at the beginning of the input stream). This can never conflict with the other meaning of ^ (complementation of character classes) since that only applies within the [] operators. If the last character is $, the expression is only matched at the end of a line (when immediately followed by a new line). If a rule is to be executed only when the Lex interpreter is in start condition _x_, the rule is prefixed by

<x>

using the angle bracket operator characters. If "being at the beginning of a line" is considered to be start condition _ONE_, then the ^ operator is equivalent to

<ONE>

Start conditions are explained more fully in Section 10.

3.9 Repetitions and Definitions

The operator pair {} specifies either repetitions (if it encloses numbers) or definition expansion (if it encloses a name). For example, the command

{digit}

looks for a predefined string named _digit_ and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules.

In contrast,

a{1,5}

looks for one to five occurrrences of _a_.

3.10 Segment Separator

The initial % is the separator for Lex segments.

SECTION 4

LEX ACTIONS

4.1 Introduction

When an expression is matched, Lex executes the corresponding action. This section describes some features of Lex that aid in writing actions. There is a default action, which consists of copying the input to the output, that is performed on all strings not otherwise matched. Thus, to absorb the entire input without producing any output, rules must be provided to match everything. When Lex is used with Yacc, this is the normal situation. Actions are used instead of copying the input to the output. A character combination that is omitted from the rules but appears as input is likely to be printed on the output, calling attention to the gap in the rules.

4.2 Regular Routines

Specifying a C null statement (;) as an action causes the input to be ignored. A frequently used rule is

```
[ \t\n]     ;
```

which causes the three spacing characters (blank, tab, and new line) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "         |
"\t"        |
"\n"        ;
```

with the same result. The quotes around \n and \t are not required.

In more complex actions, it is often necessary to know the actual text that matches some expression like [a-z]+. Lex leaves this text in an external character array named yytext. To print the name found, use a rule like:

```
[a-z]+    printf("%s", yytext);
```

This prints the string in *yytext*. The C function *printf* accepts a format argument and data to be printed. In this case, the format is "print string," % indicates data conversion, *s* indicates string type, and the characters in *yytext* are the data. This rule simply places the matched string on the output.

This action is so common that it can be written as ECHO. The expression

```
[a-z]+    ECHO;
```

is the same as the previous example. Such rules are often required to avoid matching some other rule that is not desired. For example, if there is a rule that matches *read*, it normally matches the instances of *read* contained in *bread* or *readjust*. To avoid this, a rule of the form [*a*-*z*]+ is needed. See examples in this section for variations of this situation.

Sometimes it is more convenient to know the end of what has been found; therefore, Lex also provides a count (*yyleng*) of the number of characters matched. To count both the number of words and the number of characters in words in the input, enter

```
[a-zA-Z]+ {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1]
```

Occasionally, a Lex action determines that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore*() can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. (Normally, the next input string overwrites the current entry in *yytext*.) Second, *yyless* (*n*) can be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of look-ahead offered by the / operator, but in a different form.

For example, consider a language that defines a string as a set of characters between quotation marks ("), and provides that to include a " in a string, it must be preceded by a \. The regular expression that matches this requirement is

somewhat confusing, so it might be preferable to write

```
\"[^"]*    {
        if (yytext[yyleng-1] == '\\')
            yymore();
        else
            ... normal user processing
        }
```

which, a upon finding a string such as "abc\"def", will first match the five characters, "abc\. Then the call to yymore() causes the next part of the string, "def", to be tacked on the end. The final quote terminating the string is picked up in the code labeled "normal processing."

The function yyless() reprocesses text in various circumstances. Consider the C problem of distinguishing the ambiguity of "=-a"; to treat this as "=- a" but print a message, it is possible to use a rule like:

```
=-[a-zA-Z]     {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-1);
        ... action for =- ...
        }
```

This prints a message, returns the letter after the operator to the input stream, and treats the operator as "=-". Alternatively, to treat this as "= -a", just return the minus sign as well as the letter to the input. The following command performs the other interpretation:

```
=-[a-zA-Z]     {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-2);
        ... action for = ...
        }
```

The expressions for the two cases are more easily be written as

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second. No backup is then required in the rule action.

It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of =-3, however, makes

```
=-/[^ \t\n]
```

a better rule.

4.3 Input/Output Routines

Lex also permits access to the Input/Output routines it uses. They are:

- input(), which returns the next input character
- output(c), which writes the character c on the output
- unput(c), which pushes the character c back onto the input stream to be read later by input()

By default, these routines are provided as macro definitions, but it is possible to override them and supply original versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They can be redefined to cause input or output to be transmitted to or from places, including other programs or internal memory. The character set that is used must be consistent in all routines. This means that a value of zero returned by input must mean end-of-file, and the relationship between unput and input must be retained, or the Lex look-ahead will not work.

Lex looks ahead with every rule ending in +, *, ?, or $, or containing /. Look-ahead is also necessary to match an expression that is a prefix of another expression. In other instances, Lex does not look ahead.

4.4 Library Routines

Lex library routine yywrap() is called whenever lex reaches an end-of-file. The user may wish to redefine this function. If yywrap returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, it is necessary to provide a yywrap that arranges for new input and returns 0. This instructs Lex to continue processing. The default yywrap always returns 1.

This routine is convenient for printing tables and summaries at the end of programs. It is not possible to write a normal rule that recognizes end-of-file; the only access to this condition is through *yywrap*. Unless an original version of *input*() is supplied, a file containing nulls cannot be handled, because a value of 0 returned by *input* is taken to be end-of-file.

SECTION 5

AMBIGUOUS SOURCE RULES

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1. The longest match is preferred.

2. Among rules that match the same number of characters, the rule given first is preferred.

For example, given the following rules

```
integer   keyword action ...;
[a-z]+    identifier action ...;
```

if the input is <u>integers</u>, it is taken as an identifier, because [<u>a</u>-<u>z</u>]+ matches eight characters while <u>integer</u> matches only seven. If the input is <u>integer</u>, both rules match seven characters, and the keyword rule is selected because it is given first. Anything shorter (such as <u>int</u>) does not match the expression <u>integer</u>, so the identifier action is taken.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes, but it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
"[^'\n]*"
```

which, on the above input, stops after 'first'. The consequences of errors like this are mitigated by the fact that the . operator does not match new line. Thus, expressions

like .* stop on the current line. Do not try to defeat this with expressions like [.\n]+ or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflow.

Lex normally partitions the input stream rather than searching for all possible matches of each expression. This means that each character is accounted for once only. For example, to count occurrences of both she and he in an input text, some Lex rules might be

```
she   s++;
he    h++;
\n    |
.     ;
```

where the last two rules ignore everything besides he and she. This would, however, produce unexpected results; Lex does not recognize the instances of he included in she, since once it has passed she, those characters are not analyzed again.

To override this choice, use the action REJECT, which means "do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. To count the included instances of he, change the previous example to:

```
she   {s++; REJECT;}
he    {h++; REJECT;}
\n    |
.     ;
```

After being counted, each expression is rejected; whenever appropriate, the other expression is then counted. In this example, it is possible to omit the REJECT action on he; in other cases, however, it might not be possible to tell which input characters fit in both classes.

Consider the two rules

```
a[bc]+    { ... ; REJECT;}
a[cd]+    { ... ; REJECT;}
```

If the input is ab, only the first rule matches; only the second matches ad. The input string accb matches the first rule for four characters and the second rule for three characters. In contrast, the input accd agrees with the second rule for four characters and with the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is to detect all examples of some items in the input, and the instances of these items overlap or include each other. It is not useful if the purpose is to partition the input stream. Suppose a digram table of the input is desired. Normally the digrams overlap; for example, the word _the_ is considered to contain both _th_ and _he_. Assuming a two-dimensional array called _digram_ to be incremented, the appropriate source is

```
%%
[a-z][-z]       {digram[yytext[0]][yytext[1]]++; REJECT;}
\n              ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

SECTION 6

LEX SOURCE DEFINITIONS

As Lex turns the source rules into a program, any source not intercepted by Lex is copied into the generated program. This happens in the following three cases:

1. Any line beginning with a blank or tab that is not part of a Lex rule or action is copied into the Lex generated program. Such source input prior to the first %% delimiter is external to any function in the code. If it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex that contains the actions. This material must look like program fragments, and must precede the first Lex rule.

   As a side effect, lines beginning with a blank or tab that contain a comment are passed through to the generated program. This includes comments in either the Lex source or the generated code. The comments should follow the host language convention.

2. Anything included between lines containing only %{ and %} is copied out as in the previous case. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of format, is copied out after the Lex output.

In addition to the rules, options are required to define variables used by Lex or by a user program.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and beginning in column 1, is assumed to define Lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out

by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, abbreviates rules to recognize numbers, as follows:

```
D                       [0-9]
E                       [DEde][-+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?   |
{D}*"."{D}+({E})?   |
{D}+{E}                 printf("real");
```

The first two rules for real numbers require a decimal point and contain an optional exponent field, but the first rule requires at least one digit before the decimal point and the second rule requires at least one digit after the decimal point. To handle the problem posed by a Fortran expression such as 35.EQ.I, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ   printf("integer");
```

can be used in addition to the normal rule for integers.

The definitions section can also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs.

SECTION 7

COMPILING LEX

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library.

The library is accessed by the loader flag *-ll*. An example of a appropriate set of commands is

```
lex source
cc -u -main lex.yy.c -ll
```

The resulting program is placed on the usual file *a.out* for later execution. (To use Lex with Yacc, see Section 8.) Although the default Lex I/O routines use the C standard library, Lex itself does not; if private versions of *input*, *output*, and *unput* are given, the library can be avoided.

SECTION 8

LEX AND YACC

Lex is used with Yacc (yet another compiler-compiler) to write a program named *yylex*(), required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded and its main program is used, Yacc calls *yylex*(). In this case, each Lex rule must end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input.

To obtain the grammar named "good" and the lexical rules named "better," use the commands in the following sequence:

```
yacc good
lex better
cc -u -main y.tab.c -ly -ll
```

The -u -main must appear before y.tab.c, and the Yacc library (-ly) must be loaded before the Lex library to obtain a main program that invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

SECTION 9

EXAMPLES

9.1 Copy with Simple Arithmetic Changes

The following Lex source program copies an input file while adding three to every positive number divisible by seven.

```
%%
            int k;
 [0-9]+     {
            sscanf(yytext, "%d", &k);
            if (k%7 == 0)
                printf("%d", k+3);
            else
                printf("%d",k);
            }
```

The rule [0-9]+ recognizes strings of digits; _sscanf_ converts the digits to binary and stores the result in _k_. The operator % (remainder) checks whether _k_ is divisible by seven; if it is, it is incremented by three as it is written out.

This program alters such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by seven. To avoid this, add a few more rules after the active one, as follows:

```
%%
                           int k;
-?[0-9]+                   {
                           sscanf(yytext, "%d", &k);
                           printf("%d", k%7 == 0 ? k+3 : k);
                           }
-?[0-9.]+                  ECHO;
[A-Za-z][A-Za-z0-9]+       ECHO;
```

Numerical strings containing a . or preceded by a letter are picked up by one of the last two rules, and are not changed. The _if_-_else_ has been replaced by a C conditional expression to save space. The form _a_?_b_:_c_ means "if _a_ then _b_ else _c_."

9.2 Statistical Accumulations

The following program produces histograms of the lengths of words, where a word is defined as a string of letters.

```
                int lengs[100];
%%
[a-z]+          lengs[yyleng]++;
.               |
\n              ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
      if (lengs[i] > 0)
            printf("%5d%10d\n", i ,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input, it prints the table. The final statement (_return_(_1_);) tells Lex to perform wrapup. If _yywrap_ returns zero (false), further input is available and the program continues reading and processing. Providing a _yywrap_ that never returns true causes an infinite loop.

SECTION 10

LEFT CONTEXT SENSITIVITY

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems.

This section describes three means of dealing with different environments:

- using flags
- using start conditions for rules
- switching among distinct lexical analyzers

In each case, there are rules that recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change.

A flag explicitly tested by the user's action code is the simplest way of dealing with the problem, since Lex is not necessarily involved. It may be more convenient, however, to have Lex keep track of the flags as initial conditions on the rules.

Any rule can be associated with a start condition and is only recognized when Lex is in that start condition. The current start condition can be changed at any time.

Finally, if the sets of rules for the different environments are very dissimilar, write several distinct lexical analyzers and switch from one to another as desired.

The following examples copy the input to the output, changing the word _magic_ to _first_ on every line that begins with the letter _a_, changing _magic_ to _second_ on every line that begins with the letter _b_, and changing _magic_ to _third_ on every line that begins with the letter _c_. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start    name1 name2 ...
```

The conditions can be named in any order. The word _Start_ can be abbreviated to _s_ or _S_. The conditions can be referenced at the head of a rule with brackets (<>). The command

```
<name1>expression
```

is a rule that is only recognized when Lex is in the start condition _name1_. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to _name1_. To resume the normal state, the command

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule can be active in several start conditions. For example,

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The previous example can be written:

```
%START AA BB CC
%%
^a              {ECHO; BEGIN AA;}
^b              {ECHO; BEGIN BB;}
^c              {ECHO; BEGIN CC;}
\n              {ECHO; BEGIN 0;}
<AA>magic       printf("first");
<BB>magic       printf("second");
<CC>magic       printf("third");
```

The logic is the same as before, but Lex, rather than the user's code, does the work.

SECTION 11

CHARACTER SET

The programs generated by Lex handle character I/O only through the routines _input_, _output_, and _unput_. Thus the character representation provided in these routines is accepted by Lex and used to return values in _yytext_. For internal use, a character is represented as a small integer. If the standard library is used, this integer has a value equal to the integer value of the bit pattern representing the character on the host computer. If the interpretation of a character is changed by I/O routines that translate the characters, a translation table must notify Lex. This table must be in the definitions section and must be bracketed by lines containing only %T. The table must contain lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. A sample character table follows:

```
%T
 1   Aa
 2   Bb
...
26   Zz
27   \n
28   +
29   -
30   0
31   1
...
39   9
%T
```

This table maps the lower and uppercase letters together into the integers 1 through 26, new line into 27, + and - into 28 and 29, and the digits into 30 through 39. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character can be assigned the number 0, and no character can be assigned a bigger number than the size of the hardware character set. C users probably will not wish to use the character table feature.

SECTION 12

SUMMARY OF SOURCE FORMAT

The general format of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

- Definitions, in the form "name space translation"
- Included code, in the form "space code"
- Included code, in the form

  ```
  %{
  code
  %}
  ```

- Start conditions, given in the form

  ```
  %S name1 name2 ...
  ```

- Character set tables, in the form

  ```
  %T
  number space character-string
  ...
  %T
  ```

- Changes to internal array sizes, in the form

  %*x* *nnn*

  where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action can be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

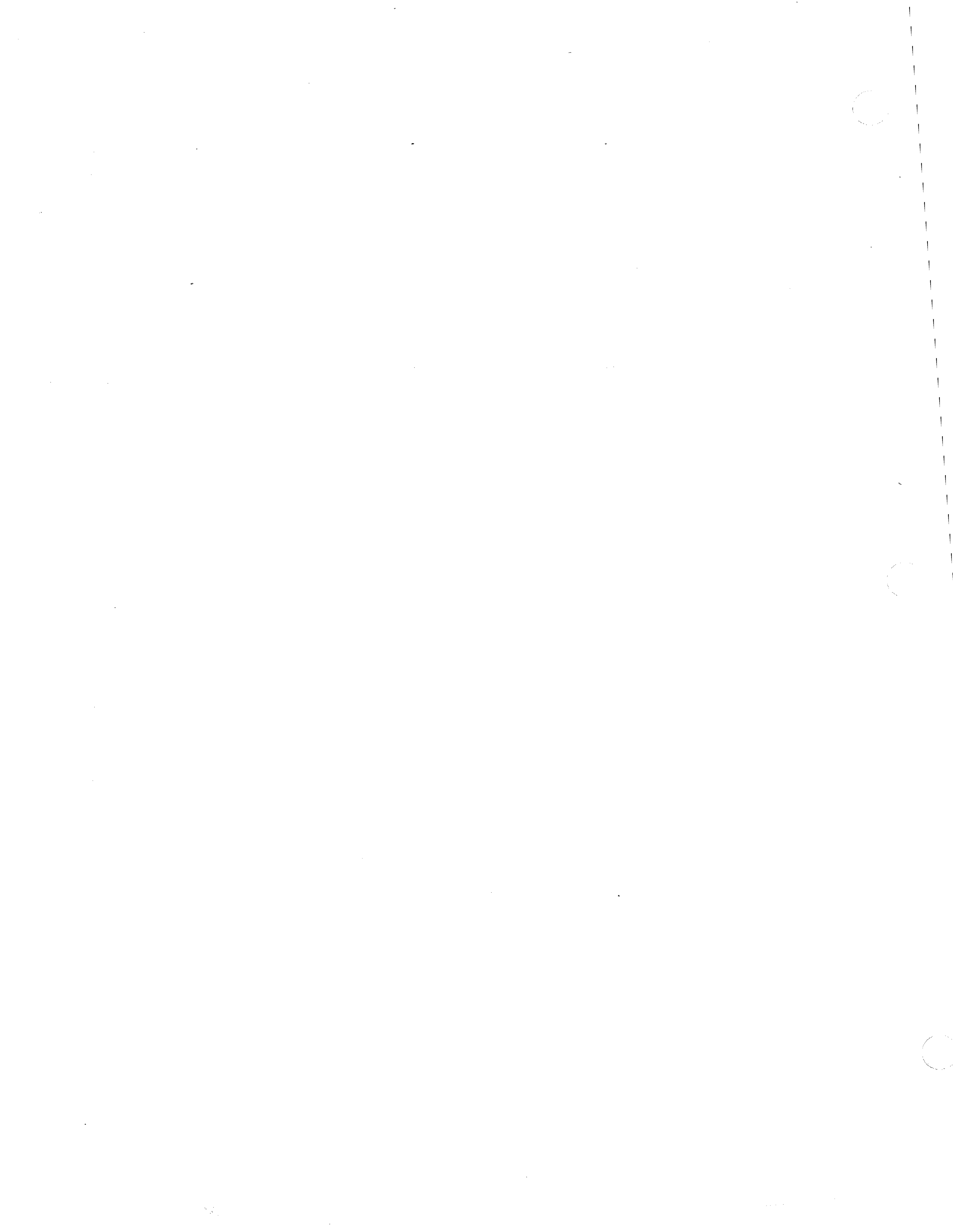
```
x        the character x
"x"      an x, even if x is an operator
\x       an x, even if x is an operator
[xy]     the character x or y
[x-z]    the characters x, y, or z
[^x]     any character but x
.        any character but new line
^x       an x at the beginning of a line
<y>x     an x when Lex is in start condition y
x$       an x at the end of a line
x?       an optional x
x*       0,1,2, ... instances of x
x+       1,2,3, ... instances of x
x|y      an x or a y
(x)      an x
x/y      an x, but only if followed by y
{xx}     the translation of xx from the definitions
         section
x{m,n}   m through n occurrences of x
```

SECTION 13

CAUTIONS

There are some expressions that produce exponential growth of the tables; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT is executed, unput must not have been used to change the characters coming from the input stream. This is the only restriction on manipulation of the not-yet-processed input.



Lint - A C Program Checker *

* This information is based on an article originally written by S.C. Johnson, Bell Laboratories.

1. Introduction and Usage

Suppose there are two C source files, *file1*.c and *file2*.*c*, which are ordinarily compiled and loaded together. (See *The C Programming Language*.) Then the command

```
lint  file1.c  file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

```
lint  -p  file1.c  file2.c
```

will produce, in addition to the above messages, additional messages which relate to the portability of the programs to other operating systems and machines. Replacing the -*p* by -*h* will produce messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying -*hp* gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. An appendix gives a summary of the *lint* options.

2. A Word About Philosophy

Many of the facts which *lint* needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether *exit* is ever called is equivalent to solving the famous ``halting problem,'' known to be recursively undecidable.

Thus, most of the *lint* algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, *lint* assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

Lint tries to give information with a high degree of relevance. Messages of the form ``*xxx* might be a bug'' are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages which *lint* produces.

3. Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These ``errors of commission'' rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!

Lint complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit *extern* statements but are never referenced; thus the statement

```
extern  float  sin();
```

will evoke no comment if *sin* is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the -*x* flag to the *lint* invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The -*v* option is available to suppress the printing of complaints about unused arguments. When -*v* is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when *lint* is applied to some, but not all, files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The -*u* flag may be used to suppress the spurious messages which might otherwise appear.

4. Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. *Lint* detects local variables (automatic and register storage classes) whose first use appears physically earlier in the

input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a ``use,'' since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that _lint_ can complain about some programs which are legal, but these programs would probably be considered bad on stylistic grounds (e.g. might contain at least two goto's). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

5. Flow of Control

Lint attempts to detect unreachable portions of the programs which it processes. It will complain about unlabeled statements immediately following goto, break, continue, or return statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases while(1) and for(;;) as infinite loops. _Lint_ also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to _exit_ may cause unreachable code which _lint_ does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement is not usually complained about by _lint_; a _break_ statement that cannot be reached causes no message. Programs generated by _yacc_, and especially _lex_ (see _YACC - Yet Another Compiler-Compiler_ and _LEX - A Lexical Analyzer_), may have literally hundreds of unreachable _break_ statements. The -_O_ flag in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up the _lint_ output. If these messages are desired, _lint_ can be invoked with the -_b_ option.

6. Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function ``values'' which have never been returned. *Lint* addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
        return( expr );
```

and

```
        return ;
```

statements is cause for alarm; *lint* will give the message

```
        function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
        f ( a ) {
            if ( a ) return ( 3 );
            g ();
            }
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a complaint from *lint*. If *g*, like *exit*, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the ``noise'' messages produced by *lint*.

On a global scale, *lint* detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. Amazingly, this bug has been observed on a couple of occasions in ``working'' programs; the desired function value just happened to have been computed in the function return register!

7. Type Checking

Lint enforces the type checking rules of C more strictly than the compilers do. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property; the argument of a return statement, and expressions used in initialization also suffer similar conversions. In these operations, char, short, int, long, unsigned, float, and double types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the -> be a pointer to structure, the left operand of the . be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types float and double may be freely matched, as may the types char, short, int, and unsigned. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

8. Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. *Lint* will quite rightly complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong

motivation for doing this, and has clearly signaled his intentions. It seems harsh for *lint* to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The -*c* flag controls the printing of comments about casts. When -*c* is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

9. Nonportable Character Use

On the S8000, characters are signed quantities, with a range from -128 to 127. On most of the other C implementations, characters take on only positive values. Thus, *lint* will flag certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
        char c;
            ...
        if( (c = getchar()) < 0 ) ....
```

works on the S8000, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, *lint* will say ``nonportable character comparison''.

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two bit field declared of type *int* cannot hold the value 3, the problem disappears if the bitfield is declared to have type *unsigned*.

10. Assignments of longs to ints

Bugs may arise from the assignment of *long* to an *int*, which loses accuracy. This may happen in programs which have been incompletely converted to use *typedefs*. When a *typedef* variable is changed from int to long, the program can stop working because some intermediate results may be assigned to ints, losing accuracy. Since there are a number of legitimate reasons for assigning longs to ints, the detection of these assignments is enabled by the -*a* flag.

11. Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by *lint*; the messages hopefully encourage better code quality, clearer style, and may even point out bugs. The -*h* flag is used to enable these checks. For example, in

the statement

```
*p++ ;
```

the * does nothing; this provokes the message ``null effect'' from *lint*. The program fragment

```
unsigned x ;
if( x < 0 ) ...
```

is clearly somewhat strange; the test will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. *Lint* will say ``degenerate unsigned comparison'' in these cases. If one says

```
if( 1 != 0 ) ....
```

lint will report ``constant in conditional context'', since the comparison of 1 with 0 gives a constant result.

Another construction detected by *lint* involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and *lint* encourages this by an appropriate message.

Finally, when the -*h* flag is in force *lint* complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many to be bad style, usually unnecessary, and frequently a bug.

12. Ancient History

There are several forms of older syntax which are being officially discouraged. These fall into two classes,

assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, . . .) could cause ambiguous expressions, such as

```
a =-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (+=, -=, etc.) have no such ambiguities. To spur the abandonment of the older forms, _lint_ complains about these old fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize _x_ to 1. This also caused syntactic difficulties: for example,

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function declaration:

```
int x ( y ) { . . .
```

and the compiler must read a fair ways past _x_ in order to sure what the declaration really is.. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

13. Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On the Honeywell 6000, double precision values must begin on even word boundaries; thus, not all such assignments make sense.

Lint tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message ``possible pointer alignment problem'' results from this situation whenever either the -*p* or -*h* flags are in effect.

14. Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on stack machines function arguments will probably be consistently evaluated either right-to-left or left-to-right. But on the S8000, with function arguments being passed in registers, the order of evaluation depends on the complexity of the arguments: more complex arguments are evaluated first. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

Lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++] ;
```

will draw the complaint:

```
warning: i evaluation order undefined
```

15. Implementation

Lint consists of two programs and a driver. The first program is a version of the Portable C Compiler which is the basis of the S8000 and several other C compilers. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator, as the other compilers do, *lint* produces an intermediate file which consists of lines of ascii text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression

trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of _lint_.

16. Portability

C is used in many installations, in part, to write system code for the host operating system. This means that the implementation of C tends to follow local conventions rather than adhere strictly to any one operating system's conventions. Despite these differences, many C programs have been successfully moved to various systems with little effort. This section describes some of the differences among implementations, and discusses the _lint_ features which encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as

```
int a ;
```

outside of any function. The ZEUS loader will resolve these declarations, and cause only a single word of storage to be set aside for _a_. Under some implementations of C, this is not feasible, so each such declaration causes a word of storage to be set aside and called _a_. When loading or library editing takes place, this causes fatal conflicts which prevent the proper operation of the program. If _lint_ is invoked with the -p flag, it will detect such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. On the ZEUS system, externally known names have seven significant characters, with the upper/lower case distinction kept. On some other systems there are from six to eight significant characters; the case distinction is often lost. This leads to situations where programs run on the ZEUS system, but encounter loader problems elsewhere. _Lint_ -_p_ causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in the ZEUS system are eight bit ascii; other systems may use a different number of bits or ebcidic in place of ascii. Moreover, character strings go from high to low bit positions (``left to right'') on ZEUS, but from low to high (``right to left'') on other systems. This means that code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. _Lint_ is of little help here, except to flag multi-character character constants.

Of course, the word sizes are different! This can cause trouble when moving code to ZEUS from a machine with a word size greater than 16 bits; moving from ZEUS to a larger word size should be less difficult. When problems do arise, they are likely to be in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

```
x &= 0177700 ;
```

to clear the low order six bits of _x_. This suffices on the S8000, but fails badly on some implementations. If the bit field feature cannot be used, the same effect can be obtained by writing

```
x &= ~ 077 ;
```

which should work on all machines.

The right shift operator is arithmetic shift on the S8000, and logical shift on many other machines. To obtain a logical shift on all machines, the left operand can be typed unsigned. Characters are considered signed integers on the S8000, and unsigned on many other machines. If there were a good way to discover the programs which would be affected, C could be changed; in any case, _lint_ is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out.

17. Shutting Lint Up

There are occasions when the programmer is smarter than _lint_. There may be valid reasons for ``illegal'' type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control

information produced by *lint* often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with *lint*, typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause a number of words to be recognized by *lint* when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, *lint* directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the *lint* directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to *lint*, this can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if it is desired to turn off strict type checking for the next expression, the directive

```
/* NOSTRICT */
```

can be used; the situation reverts to the previous default after the next expression. The -*v* flag can be turned on for one function by the directive

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by the directive

```
/* VARARGS */
```

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments which should be checked; thus,

```
/* VARARGS2 */
```

will cause the first two arguments to be checked, the others unchecked. Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

18. Library Declaration Files

Lint accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

Lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file, but are not used on a source file, draw no complaints. *Lint* does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

By default, *lint* checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the -*p* flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The -*n* flag can be used to suppress all library checking.

19. Bugs, etc.

A number of *lint* features remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of the *typedef* is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

Lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the

preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with _lint_ is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; _lint_ concentrates on issues of portability, style, and efficiency. _Lint_ can afford to be wrong, since incorrectness and over-conservatism are merely annoying, not fatal. The compiler can be fast since it knows that _lint_ will cover its flanks. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of _lint_, the desirable properties of universality and portability.

Appendix: Current Lint Options

The command currently has the form

```
lint [-options ] files... library-descriptors...
```

The options are

h Perform heuristic checks

p Perform portability checks

v Don't report unused arguments

u Don't report unused or undefined externals

b Report unreachable _break_ statements.

x Report unused external declarations

a Report assignments of _long_ to _int_ or shorter.

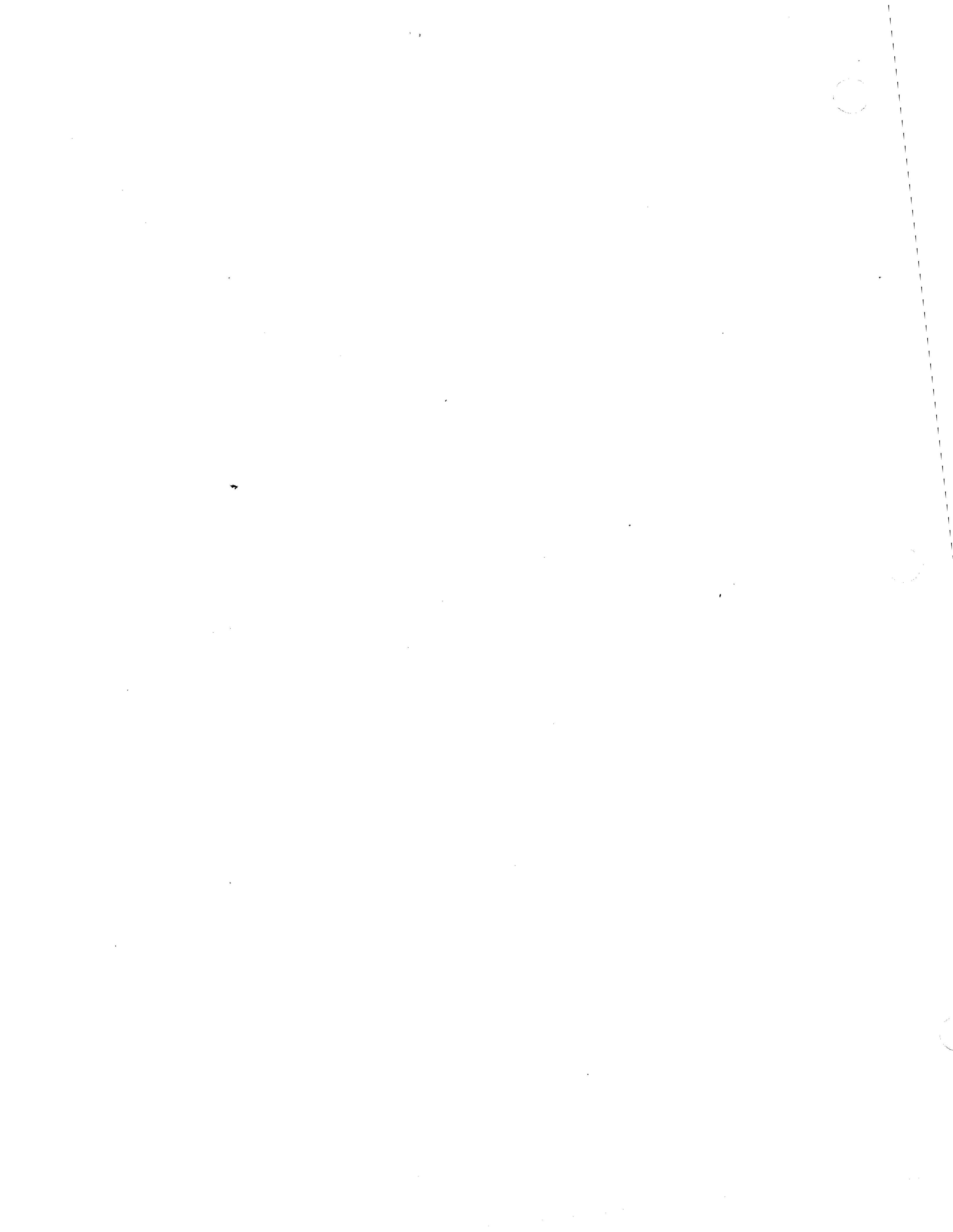
c Complain about questionable casts

n No library checking is done

s Same as _h_ (for historical reasons)

MAKE*

* This information is based on an article originally written by S.I. Feldman, Bell Laboratories.



PREFACE

This document describes *make*, a program that simplifies the process of updating program files. Section 1 describes the purpose and function of *make*. Sections 2 and 3 supply information needed to use the program. The reader should be familiar with the ZEUS Operating System and with programming in C or PLZ/SYS.

TABLE OF CONTENTS

SECTION 1 INTRODUCTION 4

1.1 Using *Make* 4

SECTION 2 BASIC FEATURES 5

2.1 Program Operation 5
2.2 Programming Example 5
2.3 File Generation and Macro Substitution ... 6
2.4 Description Files 7

SECTION 3 COMMAND USAGE 11

3.1 Arguments 11
3.2 Implicit Rules 12
3.3 Suffixes and Transformation Rules 13
3.4 Sample Program 14
3.5 Suggestions and Warnings 16

SECTION 1

INTRODUCTION

1.1 Using *Make*

In a programming project, it is common practice to divide large programs into smaller, more manageable pieces. Unfortunately, it is very easy for a programmer to forget which files depend on others, which files have been modified recently, and the exact sequence of operations needed to make or execute a new version of the program. After a long editing session, it is easy to lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations results in a program that does not work and a bug that can be very hard to track down. On the other hand, recompiling everything just to be safe is very wasteful.

Using the program *make* is a simple method for maintaining up-to-date versions of programs that are a product of many operations on numbers of files. If the information on interfile dependencies and command sequences is stored in a description file, the simple command

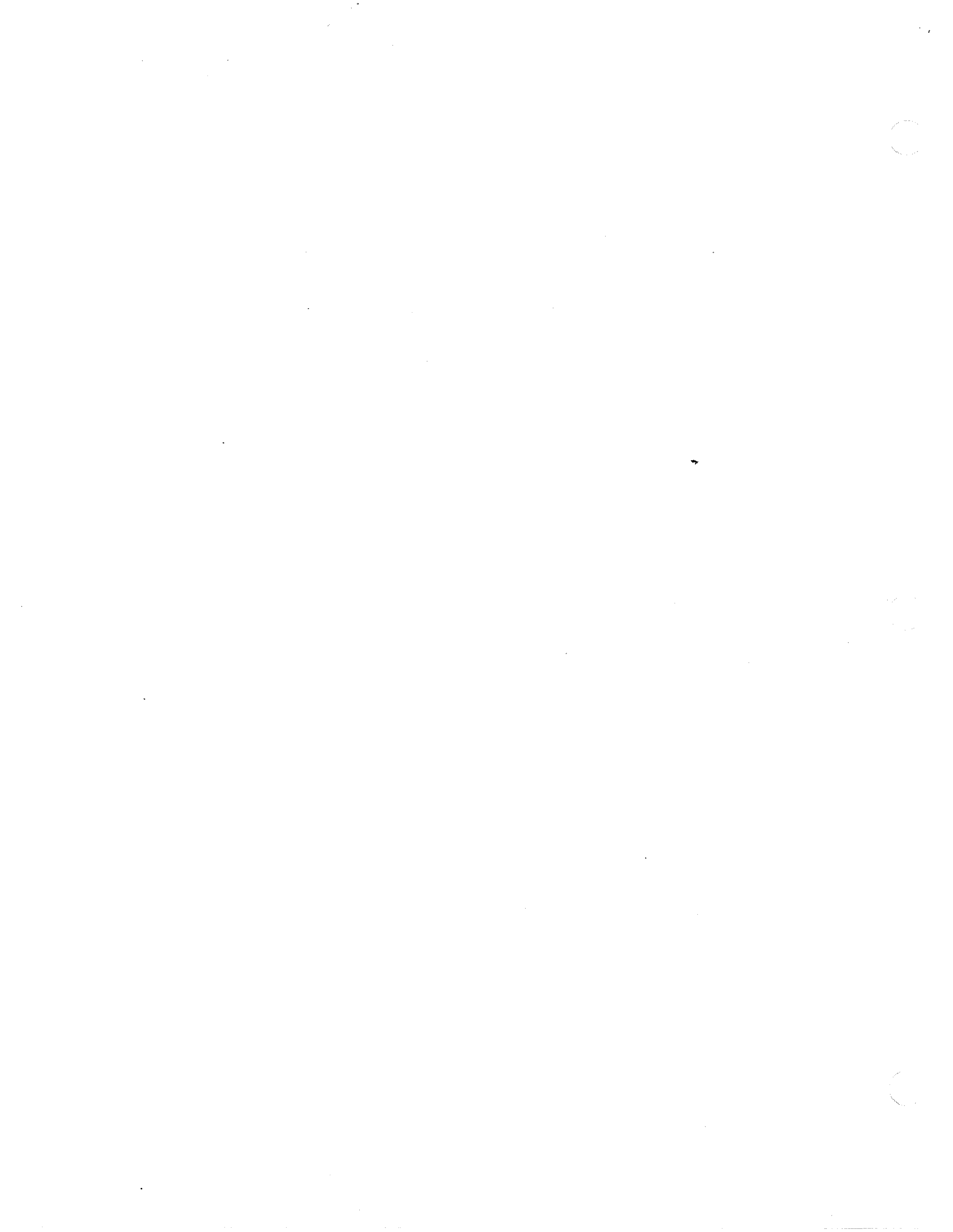
```
make
```

is usually sufficient to update the relevant files, regardless of the number that have been edited since the last *make*. In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

```
think - edit -  make - test  . . .
```

The *make* command creates the proper files simply, correctly, and with a minimum amount of effort. It also includes a simple macro substitution facility and encloses commands in a single file for convenient administration.

Make is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs.



SECTION 2

BASIC FEATURES

2.1 Program Operation

The basic operation of *make* is to find the name of a needed target file and update it by ensuring that all of the files on which it depends exist and are up to date. It then creates the target if it has not been modified since the last modification of its dependents. *Make* does a depth-first search of the graph of dependencies. The operation of the command depends on the availability of the date and time that a file was last modified.

2.2 Programming Example

A program named *prog* is made by compiling and loading three C language files, *x.c*, *y.c*, and *z.c*, with the *lc* library. By convention, the output of the C compilations is found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog: x.o  y.o  z.o
      cc  x.o  y.o  z.o   -lc  -o  prog

x.o y.o: defs
```

If this information is stored in a file named *makefile*, the command

```
make
```

performs the operations needed to recreate *prog* after any changes are made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

Make uses three sources of information, a user-supplied description file, file names and last-modified times from the file system, and built-in rules that bridge some of the gaps. In this example, the first line indicates that *prog* depends on three object (.o) files. Once these object files are current, the second line describes how to load them to

create *prog*. The third line indicates that *x.o* and *y.o* depend on the file *defs*. From the file system, *make* discovers that there are three C source (.c) files corresponding to the needed .o files, and uses built-in information on how to generate an object from a source file (issue a cc -c command).

The following description file is equivalent to *makefile* but does not take advantage of *make's* built-in information.

```
prog:   x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lc  -o  prog
x.o:    x.c  defs
        cc  -c  x.c
y.o:    y.c  defs
        cc  -c  y.c
z.o:    z.c
        cc  -c  z.c
```

If none of the source or object files have changed since the last time *prog* was made, all of the files are current. Issuing the command

```
make
```

causes the program to announce this fact and stop. If, however, the *defs* file has been edited, *x.c* and *y.c* (but not *z.c*) are recompiled, and *prog* is created from the new .o files. If only the file *y.c* has changed, that file alone is recompiled, but it is still necessary to reload *prog*.

If no target name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made.

In the *makefile* example,

```
make x.o
```

recompiles *x.o* if *x.c* or *defs* have changed.

2.3 File Generation and Macro Substitution

It is useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries use *make's* ability to generate files and substitute macros. For example, an entry called *save* can be included to copy a certain set of files, or an entry called *cleanup* can be used to throw away unneeded intermediate files. A zero-length file can be maintained to keep track of the time when certain actions were performed. This

technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for making substitutions in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name with a dollar sign. Macro names longer than one character must be enclosed in parentheses or braces. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
${Z}
```

The last three invocations are identical. All of these macros are assigned values during input, as shown below. (Four special macros change values during the execution of the command: $*, $@, $?, and $<. See Section 2.4.)

```
OBJECTS = x.o y.o z.o
LIBES = -lc
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
  . . .
```

The command

```
make
```

loads the three object files with the _lc_ library. The command

```
make  "LIBES= -lm -lc"
```

loads them with both the math (-lm) and the standard (-lc) libraries, since macro definitions on the command line override definitions in the description. (In ZEUS commands, it is necessary to enclose arguments with embedded blanks in quotes.)

2.4 Description Files

A description file contains three types of information: macro definitions, dependency information, and executable commands.

A macro definition is a line that contains an equal sign that is not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign is assigned the string of characters following the equal sign (trailing and leading blanks and tabs are stripped out). The following are valid macro definitions:

```
2 = xyz
abc = -lm -lmp -lc
LIBES =
```

The last definition assigns the null string to LIBES. A macro that is never explicitly defined has the null string as its value. Macro definitions can also appear on the make command line (Section 3.1).

Other lines give information about target files. The general form of an entry is:

```
target1 [target2...] :[:] [dependent1...] [; commands] [#...]
[(tab) commands] [#...]
...
```

Items inside brackets can be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters * and ? are expanded.) A command is any string of characters not including a # (unless in quotes) or new line. Commands can appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

If a line begins with a sharp (#), all characters after the # are ignored, as is the # itself. Blank lines also are totally ignored. If a noncomment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, new line, and following blanks and tabs are replaced by a single blank.

A dependency line can have either a single or a double colon. A target name can appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

For the single colon case, no more than one dependency line can have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default creation rule can be invoked.

In the double colon case, a command sequence can be associated with each dependency line; if the target is out of date

with any of the files on a particular line, the associated commands are executed. A built-in rule can also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in silent mode or if the command line begins with an @ sign. *Make* normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the -i flag has been specified on the *make* command line, if the target name .IGNORE appears in the description file, or if the command string in the description file begins with a hyphen. Some ZEUS commands return meaningless status.

Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (such as *cd* and shell control commands) that have meaning only within a single shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set. $@ is set to the name of the file to be made. $? is set to the string of names that are found to be newer than the target. If the command was generated by an implicit rule (Section 3.2), $< is the name of the related file that caused the action, and $* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used. If there is no such name, *make* prints a message and stops.

Targets and dependents are usually file names. A special notation exists for targets or dependents within archives *ar*(1). The notation

```
archive(file)
```

or

```
archive((entry point))
```

refers to the file within the archive. Modification dates are based on the dates stored within the archive, not the archive itself. For example,

```
libc.a (printf.o)
```

or

```
libc.a (_printf))
```

refer to the object module printf.o in the archive libc.a.

SECTION 3

COMMAND USAGE

3.1 Arguments

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

make [flags] [macro definitions] [targets]

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are:

-d Debug mode. Print out detailed information on files and times examined.

-f Description file name. The next argument is assumed to be the name of a description file. The file name dash (-) denotes the standard input. If there are no -f arguments, the file named *makefile* (or *Makefile*) in the current directory is read. The contents of the description files override the built-in rules if they are present.

-i Ignore error codes returned by invoked commands. This mode is entered if the target name .IGNORE appears in the description file.

-n No execute mode. Print commands, but do not execute them. Even lines beginning with an @ sign are printed.

-p Print out the complete set of macro definitions and target descriptions.

-q Question. The *make* command returns a zero or nonzero status code depending on whether the target file is or is not up to date.

-r Do not use the built-in rules.

-s Silent mode. Do not print command lines before executing. This mode is also entered if the target name .SILENT appears in the description file.

-t Touch the target files (causing them to be up to date) rather than issue the usual commands.

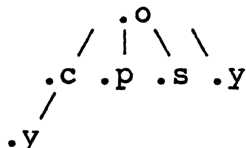
The remaining arguments are assumed to be the names of targets to be made; they are done in left-to-right order. If there are no such arguments, the first name in the description files that does not begin with a period is made.

3.2 Implicit Rules

The *make* program uses a table of common suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is:

.o	Object file
.c	C source file
.p	PLZ/SYS source file
.s	Assembler source file
.y	Yacc-C source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* is needed and there is an *x.c* in the description or directory, *x.c* is compiled. If there is also an *x.y*, that grammar is run through Yacc before the result is compiled. However, if there is no *x.c* but there is an *x.y*, *make* discards the intermediate C language file and uses the direct link in the graph above.

If the macro names being used are known, it is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked. The compiler names are the macros AS, CC, PLZ, and YACC. The command

```
make CC=newcc
```

causes the newcc command to be used instead of the usual C

compiler. The macros CFLAGS, PFLAGS, and YFLAGS can be set to cause these commands to be issued with optional flags. Thus,

```
make "CFLAGS=-O"
```

causes the optimizing C compiler to be used.

3.3 Suffixes and Transformation Rules

The _make_ program itself does not recognize whether or not file name suffixes are relevant; it cannot transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the -r flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name .SUFFIXES; _make_ looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, _make_ proceeds normally. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a PLZ/SYS source (.p) file to a .o file is thus .p.o. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule .p.o is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant; it is scanned from left to right, and _make_ uses the first name that is formed that has both a file and a rule associated with it. If new names are to be appended, just add an entry for .SUFFIXES in the description file; the dependents will be added to the usual list. A .SUFFIXES line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed.)

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .p .y .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
CC=cc
AS=as -u
CFLAGS=
```

```
PLZ=plz
PFLAGS=
.c.o :
    $(CC) $(CFLAGS) -c $<
.p.o :
    $(PLZ) $(PFLAGS) -c $<
.s.o :
    $(AS) -o $@ $<
.y.o :
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    rm y.tab.c
    mv y.tab.o $@
.y.c :
    $(YACC) $(YFLAGS) $<
       mv y.tab.o $@
```

3.4 Sample Program

As an example of the use of make, the description file used to maintain the make command itself is given. The code for make is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command

P = lpr
FILES = Makefile version.c defs main.c doname.c misc.c
        files.c dosys.c gram.y
OBJECTS = version.o main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O

make:         $(OBJECTS)
              cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
              size make

$(OBJECTS):    defs

cleanup:
              -rm *.o gram.c
              -du

install:
              @size make /usr/bin/make
              cp make /usr/bin/make ; rm make
```

```
print:      $(FILES)    # print recently changed files
            pr $? | $P
            touch print

test:
            make -dp | grep -v TIME >1zap
            /usr/bin/make -dp | grep -v TIME >2zap
            diff 1zap 2zap
            rm 1zap 2zap

lint:       dosys.c doname.c files.c main.c misc.c /
            version.c gram.c
            $(LINT) dosys.c doname.c files.c main.c /
            misc.c version.c gram.c rm gram.c
```

Make displays each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

```
cc  -c version.c
cc  -c main.c
cc  -c doname.c
cc  -c misc.c
cc  -c files.c
cc  -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  -c gram.c
cc  version.o main.o doname.o misc.o files.o dosys.o
    gram.o -ls -o make
    13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars are mentioned by name in the description file, _make_ finds them using its suffix rules and issues the needed commands. The string of digits results from the _size make_ command; the printing of the command line itself is suppressed by an @ sign. The @ sign on the _size_ command in the description file suppresses the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The _print_ entry prints only the files that have been changed since the last _make print_ command. A zero-length file _print_ is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since _print_ was touched. The printed output can be sent to a different printer or to a file by changing the definition of

the *P* macro:

```
make print "P = opr -sp"
            or
make print "P=  cat >zap"
```

3.5 Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has an #*include* "*defs*" line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, but it is necessary to recreate *x.o*.)

To discover what *make* would do, the -*n* option is very useful. The command

```
make -n
```

orders *make* to print out the commands it would issue without actually executing them.

If a change to a file is absolutely certain to be benign (for example, adding a new definition to an include file), the -t (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, *make* updates the modification times on the affected file. Thus, the command

```
make -ts
```

(touch silently) causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of *make* and destroys all memory of the previous relationships.

The debugging flag (-d) causes *make* to print out a very detailed description of what it is doing, including the file times. The output is verbose, so this option is recommended only as a last resort.

Typing Documents on the ZEUS System

Using the -ms Macros with Troff and Nroff*

* This information is based on an article originally written by M.E. Lesk, Bell Laboratories.

PREFACE

This document describes a set of macros for preparing documents using the ZEUS _troff_ and _nroff_ formatting programs. Documents can be output using either a phototypesetter or a computer terminal without changing the input.

Section 1 describes procedures for creating document files. Section 2 tells how to print the documents. Section 3 outlines the use of macros for producing tables and special symbols. The appendix summarizes the -ms commands.

Refer to the sections on nroff and troff for further information on producing documents.

TABLE OF CONTENTS

SECTION 1 PREPARING THE FILE 4

1.1 Text 4
1.2 Front Matter 4
1.3 Cover Sheets and First Pages 5
1.4 Page Headings 5
1.5 Multicolumn Formats 6
1.6 Section Headings 6
1.7 Indented Paragraphs 7
1.8 Emphasis 9
1.9 Footnotes 10
1.10 Displays and Tables 10
1.11 Boxing Words or Lines 11
1.12 Keeping Blocks Together 12
1.13 Nroff/Troff Commands 12
1.14 Date 12
1.15 Signature Line 12
1.16 Registers 13
1.17 Accents 13

SECTION 2 PRINTING THE DOCUMENT 15

SECTION 3 USING ADVANCED FORMAT OPTIONS 16

3.1 Special Symbols 16
3.2 Tables 16

APPENDIX A LIST OF COMMANDS 17

SECTION 1

PREPARING THE FILE

1.1 Text

Type normally, except that instead of indenting for paragraphs, place the line

```
.PP
```

before each paragraph. This produces indenting after an extra line space.

Alternatively, the command .LP produces a left-aligned (block) paragraph. The paragraph spacing can be changed (Section 1.16).

1.2 Front Matter

Start front matter as follows:

```
[optional overall format .RP, Section 1.3]
.TL
Title of document (one or more lines)
.AU
Author(s) (one or more lines)
.AI
Author's institution(s)
.AB
Abstract; to be placed on the cover sheet of a document.
Line length is 5/6 of normal; use .ll here to change.
.AE  (abstract end)
text ... (begins with .PP, Section 1.1)
```

To omit some of the standard headings (such as abstract or author's institution), omit the fields and corresponding command lines. Several interspersed .AU and .AI lines can be used for multiple authors. The headings are not compulsory; beginning with a .PP command starts the document with an ordinary paragraph.

NOTE

Do not begin a document with a line of text. Some -ms command must precede any text input. When in doubt, use .LP to get proper initialization. The commands .PP, .LP, .TL, .SH, and .NH are also allowed.

1.3 Cover Sheets and First Pages

The first line of a document signals the general format of the first page. In particular, if the first command is .RP, a cover sheet with title and abstract is generated. The default format of no cover sheet is useful for scanning drafts.

In general, -ms is arranged so that only one form of a document need be stored. The first command gives the format, and unnecessary items for that format are ignored.

NOTE

Do not put extraneous material between the .TL and .AE commands. Processing of the titling items is special, and other data placed between them may not be processed as expected. Some -ms command must precede any input text.

1.4 Page Headings

The -ms macros, by default, print a page heading containing a page number. A default page footing is provided only in _nroff_, where the date is used. Minor adjustments to the page headers/footers are made by redefining the strings LH, CH, and RH (which are the left, center, and right portions of the page headers), and the strings LF, CF, and RF (the left, center, and right portions of the page footer). To get the proper page number in these strings, use a backslash (\) as in:

```
.ds CH "\- \\n(PN \-
```

which defines a center header of the form

```
- 5 -
```

For page number, the number register PN should be used in preference to the register %.

For more complex formats, redefine the macros PT and BT, which are invoked (respectively) at the top and bottom of each page. The margins, taken from registers HM and for the top margin FM for the bottom margin, are normally one inch. The page header/footer is in the middle of that space. If these macros are redefined, be careful with parameters such as point size or font.

1.5 Multicolumn Formats

The command

```
.MC [column width [gutter width]]
```

makes multiple columns with the specified column and gutter width. The maximum is as many columns as fit across the page. Whenever the number of columns is changed (except going from full width to some larger number of columns), a new page is started.

This feature is more useful for typeset output than for output to the terminal. Placing the command .2C in your document causes it to be printed in double-column format beginning at that point. The command .1C produces one-column format. Changing column format causes a page break.

1.6 Section Headings

Two commands, .NH and .SH, are used to produce section headings. Entering

```
.NH
type section heading here
can be several lines
```

produces a numbered section heading in boldface. The .SH command produces an unnumbered heading.

Every section heading must be followed by a paragraph beginning with .PP or .LP to indicate the end of the heading. Headings can contain more than one line of text.

The .NH command also supports more complex numbering schemes. If a numerical argument is given, it is taken to be a level number, and an appropriate subsection number is generated. Larger level numbers indicate deeper subsections. For example,

```
.NH
Erie-Lackawanna
.NH 2
Morris and Essex Division
.NH 3
Gladstone Branch
.NH 3
Montclair Branch
.NH 2
Boonton Line
```

generates:

2. Erie-Lackawanna

2.1. Morris and Essex Division

2.1.1. Gladstone Branch

2.1.2. Montclair Branch

2.2. Boonton Line

1.7 Indented Paragraphs

Paragraphs with hanging numbers, such as references, are often handled with indented paragraphs.

Example 1. Simple Indentation

```
.IP [1]
Text for first paragraph, typed
normally for as long as necessary
on as many lines as needed.
.IP [2]
Text for second paragraph, ...
```

produces

[1] Text for first paragraph, typed normally for as long as necessary on as many lines as needed.

[2] Text for second paragraph, ...

A series of indented paragraphs can be followed by an ordinary paragraph by entering .PP or .LP.

Example 2. Block Indentation

More sophisticated uses of .IP are also possible. If the label is omitted, for example, a plain block indent is produced. The lines

```
.IP
This material will
just be turned into a
block indent suitable for quotations.
.LP
```

produce

> This material will just be turned into a block indent suitable for quotations.

Example 3. Nonstandard Indentation

If a nonstandard amount of indenting is required, it is specified after the label (in character positions) and remains in effect until the next .PP or .LP. Thus, the general form of the .IP command contains two additional fields: the label and the indenting length. For example,

```
.IP first: 9
Notice the longer label, requiring larger
indenting for these paragraphs.
.IP second:
And so forth.
.LP
```

produces the following:

first: Notice the longer label, requiring larger indenting for these paragraphs.

second: And so forth.

Example 4. Multiple Nested Indentations

It is also possible to produce multiple nested indents. The command .RS indicates that the next .IP starts from the current indentation level. Each .RE takes one level of indenting, so .RS and .RE commands must be balanced. The .RS command can be thought of as "move right" and the .RE command as "move left." For example,

```
.IP 1.
Customer Corporation
.RS
.IP 1.1
Murray Hill
.IP 1.2
Holmdel
.IP 1.3
Whippany
.RS
.IP 1.3.1
Madison
.RE
.IP 1.4
Chester
.RE
.LP
```

results in

1. Customer Corporation

    1.1 Murray Hill

    1.2 Holmdel

    1.3 Whippany

        1.3.1 Madison

    1.4 Chester

Example 5. Right Indentation

All of these variations on .LP leave the right margin untouched. Sometimes, for purposes such as setting off a quotation, a paragraph indented on both right and left is required.

> A single paragraph like this is obtained by preceding it with .QP. More complicated material (several paragraphs) is bracketed with .QS and .QE.

1.8 Emphasis

To produce italics on the typesetter or underlining on the terminal, use

```
.I
as much text as you want
```

```
can be typed here
.R
```

as was done for <u>these three words</u>. The .R command restores the normal (usually Roman) font. If only one word is to be italicized or underlined, it can be input on a separate line with the .I command,

```
.I word
```

In this case, no .R is needed to restore the previous font.

Boldface output on the typesetter is produced by

```
.B
Text to be set in boldface
goes here
.R
```

This is also underlined on the terminal or line printer. As with .I, a single word can be placed in boldface by placing it on a separate line with the .B command.

Size changes can be specified with the commands .LG (make larger), .SM (make smaller), and .NL (return to normal size). The size change is two points; the commands can be repeated for increased effect.

To specify an underlined word on the typesetter, use the command

```
.UL word
```

There is no way to underline multiple words on the typesetter.

1.9 Footnotes

Material placed between lines with the commands .FS for footnote and .FE for footnote end is collected and placed at the bottom of the current page after an asterisk (*). By default, footnotes are 11/12th the length of normal text, but this can be changed using the FL register (Section 1.16).

1.10 Displays and Tables

To prepare displays whose lines are not to be rearranged (such as tables), enclose the text in the commands .DS and .DE as follows:

```
.DS
table lines, like the
examples here, are placed
between .DS and .DE
.DE
```

By default, lines between .DS and .DE are indented and left-adjusted. It is also possible to center lines, or retain the left margin. Lines bracketed by .DS C and .DE commands are centered and not rearranged. Lines bracketed by .DS L and .DE are left-adjusted, not indented, and not rearranged. The command .DS is equivalent to .DS I, which indents and left-adjusts. For example,

these lines were preceded
by .DS C and followed by
a .DE command;

whereas

These lines were preceded
by .DS L and followed by
a .DE command.

There is also a variant, .DS B, that makes the display into a left-adjusted block of text, then centers that entire block.

Normally, a display is kept on one page. To produce a long display split across page boundaries, use .CD, .LD, or .ID in place of the commands .DS C, .DS L, or .DS I, respectively. An extra argument to the .DS I or .DS command specifies the amount to indent. There is no command to right-adjust lines.

1.11 Boxing Words or Lines

To draw a rectangular box around a word, use the command

```
.BX word
```

Longer pieces of text can be boxed by enclosing them with .B1 and .B2, as with

```
.B1
text...
.B2
```

Italics are preferred to boxes because boxes are not printed neatly on a terminal. However, col may be used to improve such terminal output. See ZEUS Reference Manual, Section 1.

1.12 Keeping Blocks Together

To keep a table or other block of lines together on a page, use the keep - release commands. If a block of lines preceded by .KS and followed by .KE does not fit on the remainder of the current page, it begins on a new page. (Lines bracketed by .DS and .DE commands are automatically kept together this way.) There is also a keep floating (.KF) command. If a block preceded by .KF (instead of .KS) does not fit on the current page, it is moved down through the text until the top of the next page. Thus, no large blank space is introduced in the document.

1.13 Nroff/Troff Commands

The following commands from the basic formatting programs work for both typesetter and computer terminal output:

```
.bp   begin new page
.br   "break" stop running text from line to line
.sp n insert n blank lines
.na   do not adjust right margins
```

1.14 Date

By default, documents produced on computer terminals have the date at the bottom of each page, and documents produced on the typesetter do not. To force the date, use the .DA command. To force no date, use the .ND command. To force a fixed date, enter the date after the .DA command; for example,

```
.DA July 4, 1776
```

The command ".ND May 8, 1945" in .RP format places the specified date on the cover sheet and nowhere else. Place this line before the title.

1.15 Signature Line

To obtain a signature line, use the command .SG. The author's name is output in place of the .SG line. An argument to .SG is used as a typing identification line, and placed after the signatures. The .SG command is ignored in released paper format.

1.16 Registers

Certain of the registers used by -ms can be altered to change default settings using commands beginning with .nr. For example,

    .nr PS 9

makes the default point size 9 point. If the effect is needed immediately, use the normal *troff* command in addition to changing the number register.

Reg.	Defines	Takes Effect	Default
PS	point size	next paragraph	10
VS	line spacing	next paragraph	12 pts
LL	line length	next paragraph	6 inches
LT	title length	next paragraph	6 inches
PD	para. spacing	next paragraph	0.3 VS
PI	para. indent	next paragraph	5 ens
FL	footnote length	next FS	11/12 LL
CW	column width	next 2C	7/15 LL
GW	intercolumn gap	next 2C	1/15 LL
PO	page offset	next page	26/27 inches
HM	top margin	next page	1 inch
FM	bottom margin	next page	1 inch

It is also possible to alter the strings LH, CH, and RH (left, center, and right headers), and LF, CF, and RF (strings in the page footers). The page number on *output* is taken from register PN to permit changing its output style. For more complicated headers and footers, the macros PT and BT can be redefined as explained, in Section 1.4.

1.17 Accents

To simplify typing certain foreign words, strings representing common accent marks are defined for use on photocomposition systems and terminals on which strikeover characters have been defined.

Input	Output Character with:
*'e	accute accent
*`e	grave accent
*:u	umlaut (diaeresis)
*^e	circumflex
*~a	tilde
*Ce	hacek (wedge)
*,c	cedilla

SECTION 2

PRINTING THE DOCUMENT

After the document is prepared and stored on a file, it can be displayed on a terminal with the command

```
nroff -ms file
```

If double-column format (2C) is being used, pipe the *nroff* output through *col* by making the first line of the input

```
.pi /z/bin/col
```

The document can be printed on the typesetter with the command

```
troff -ms file
```

Many options are possible. In each case, if the document is stored in several files, list all the file names used. If equations or tables are used, *eqn* and/or *tbl* must be invoked as preprocessors.

SECTION 3

USING ADVANCED FORMAT OPTIONS

3.1 Special Symbols

To use Greek or mathematics symbols, see _eqn_ for equation setting. To aid _eqn_ users, -_ms_ provides definitions of .EQ and .EN, which normally center the equation and set it off slightly. An argument on .EQ is taken to be an equation number and placed in the right margin near the equation. In addition, there are three special arguments to EQ; the letters C, I, and L indicate centered (default), indented, and left-adjusted equations. If there is both a format argument and an equation number, give the format argument first, as in

```
.EQ L (1.3a)
```

for a left-adjusted equation numbered (1.3a).

3.2 Tables

The macros .TS and .TE are defined to separate tables from text with white space. A very long table with a heading can be broken across pages by beginning it with .TS H instead of .TS, and placing the line .TH in the table data to repeat the heading. If the table has no heading repeated from page to page, use the ordinary .TS and .TE macros.

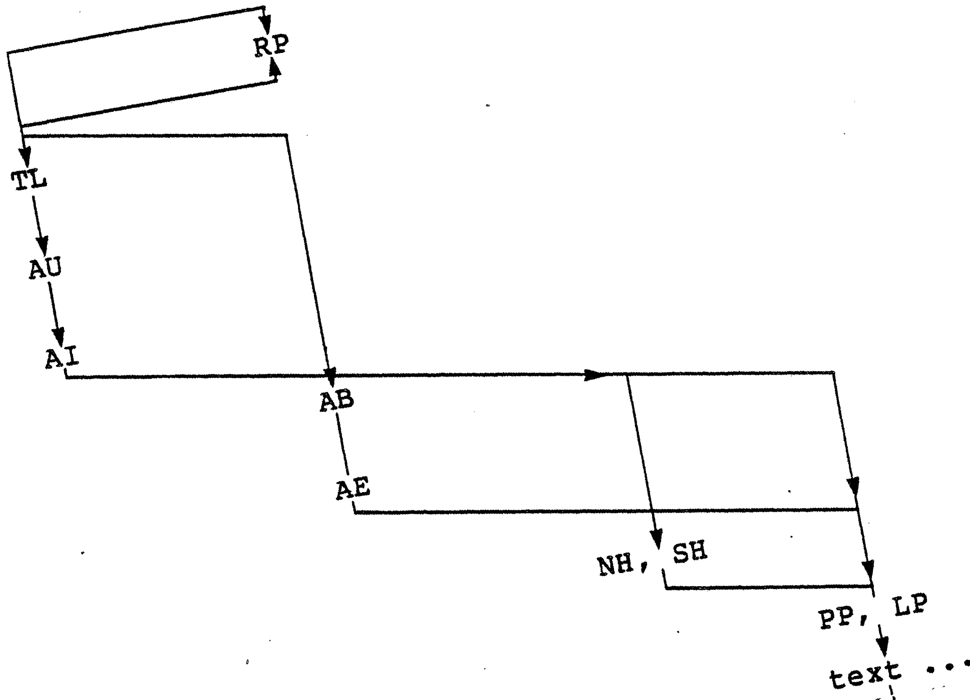
APPENDIX A

LIST OF COMMANDS

1C	Return to single-column format
2C	Start double-column format
AB	Begin abstract
AE	End abstract
AI	Specify author's institution
AU	Specify author
B	Begin boldface
DA	Provide the date on each page
DE	End display
DS	Start display (also CD, LD, ID)
EN	End equation
EQ	Begin equation
FE	End footnote
FS	Begin footnote
I	Begin italics
IP	Begin indented paragraph
KE	Release keep
KF	Begin floating keep
KS	Start keep
LG	Increase type size
LP	Left aligned block paragraph
ND	Change or cancel date
NH	Specify numbered heading
NL	Return to normal type size
PP	Begin paragraph
R	Return to regular font (usually Roman)
RE	End one level of relative indenting
RP	Use released paper format
RS	Relative indent increased one level
SG	Insert signature line
SH	Specify section heading
SM	Change to smaller type size
TL	Specify title
UL	Underline one word

MS

Order of Commands in Input



19

Register Names

The following register names are used by -ms internally. Independent use of these names in one's own macros may produce incorrect output. No lowercase letters are used in any -ms internal name.

Number Registers Used in -ms

:	FC	H4	IQ	MF	NS	PO	TC	YY
#T	FL	H5	IR	MM	OI	PQ	TD	ZN
1T	FP	HT	KI	MO	PD	PX	TQ	
AV	GW	IF	L1	NA	PE	RO	TV	
CW	H1	IK	LE	NC	PF	ST	VS	
DW	H2	IM	LL	ND	PI	T.	WF	
EF	H3	IP	LT	NF	PN	TB	YE	

String Registers Used in -ms

'	AI	CS	EM	I	LB	OK	RP	TL
`	AU	CT	EN	I1	LD	PP	RQ	TM
^	B	D	EQ	I2	LG	PT	RS	TQ
~	BG	DA	EZ	I3	LP	PY	RT	TS
:	BT	DE	FA	I4	ME	QF	SO	TT
,	C	DS	FE	I5	MF	R	S1	UL
1C	C1	DW	FJ	ID	MH	R1	S2	WB
2C	C2	DY	FK	IE	MN	R2	SG	WH
A1	CA	E1	FN	IM	MO	R3	SH	WT
A2	CB	E2	FO	IP	MR	R4	SM	XD
A3	CC	E3	FQ	IZ	ND	R5	SN	XF
A4	CD	E4	FS	KE	NH	RC	SY	XK
A5	CF	E5	FV	KF	NL	RE	TA	
AB	CH	EE	FY	KQ	NP	RF	TE	
AE	CM	EL	HO	KS	OD	RH	TH	

NROFF/TROFF USER'S MANUAL*

* This information is based on an article originally written by Joseph F. Ossanna, Bell Laboratories.

PREFACE

This document is a reference manual for the nroff/troff text processors. The reader is expected to have some experience with these text processors before using this manual. For an introductory text, see Troff Tutorial.

Each section of this document covers an nroff/troff command or set of related commands, and sections appear in order of use; that is, frequently used commands appear first. At the end of this documment are several summaries and appendixes.

Numerical parameters are indicated in this manual in two ways. ±N means that the argument can take the forms N, +N, or -N and that the corresponding effect is to set the affected parameter to N, to increment it by N, or to decrement it by N. N means that an initial algebraic sign is not an increment indicator, but merely the sign of N. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are sp, wh, ch, nr, and if. The requests ps, ft, po, vs, ls, ll, in, and lt restore the previous parameter value in the absence of an argument.

Single character arguments are indicated by single lowercase letters and one/two character arguments are indicated by a pair of lowercase letters. Character string arguments are indicated by multicharacter mnemonics.

NOTE

> The version of troff on ZEUS produces output for a Graphic Systems Inc. C/A/T phototypesetter. This device is not presently supported by ZEUS. Since the device is not present, it will always appear to be busy to troff.

TABLE OF CONTENTS

SECTION 1 BASIC INFORMATION 6

1.1 Introduction 6
1.2 Usage 6
1.3 Form of Input 8
1.4 Formatter and Device Resolution 9
1.5 Numerical Parameter Input 9
1.6 Numerical Expressions 10

SECTION 2 FONT AND CHARACTER SIZE CONTROL 11

2.1 Character Set 11
2.2 Fonts 11
2.3 Character Size 12

SECTION 3 PAGE CONTROL 15

SECTION 4 TEXT FILLING, ADJUSTING, AND CENTERING 18

4.1 Filling and Adjusting 18
4.2 Interrupted Text 19

SECTION 5 SPACING 21

5.1 Base-Line Spacing 21
5.2 Extra-Line Spacing 21
5.3 Blocks of Vertical Space 21
5.4 Line Length and Indenting 23

SECTION 6 MACROS, STRINGS, DIVERSIONS, AND POSITION TRAPS 25

6.1 Macros and Strings 25
6.2 Copy Mode and Input Interpretation ... 25
6.3 Arguments 26
6.4 Diversions 27
6.5 Traps 27

SECTION 7 NUMBER REGISTERS 31

TABLE OF CONTENTS (continued)

SECTION 8 TABS, LEADERS, AND FIELDS 33

8.1 Tabs and Leaders 33
8.2 Fields 33

SECTION 9 INPUT/OUTPUT CONVENTIONS AND CHARACTER TRANSLATIONS 35

9.1 Input Character Translation 35
9.2 Ligatures 35
9.3 Backspacing, Underlining, and Overstriking 36
9.4 Control Characters 37
9.5 Output Translation 37
9.6 Transparent Throughput 38
9.7 Comments and Concealed New Lines 38

SECTION 10 LOCAL MOTIONS AND THE WIDTH FUNCTION 39

10.1 Local Motions 39
10.2 Width Functions 39
10.3 Mark Horizontal Place 40

SECTION 11 OVERSTRIKE, LINE-DRAWING, AND ZERO-WIDTH FUNCTIONS 41

11.1 Overstriking 41
11.2 Line Drawing 41
11.3 Zero-Width Characters 42

SECTION 12 HYPHENATION 43

SECTION 13 THREE-PART TITLES 45

SECTION 14 OUTPUT LINE NUMBERING 46

SECTION 15 CONDITIONAL ACCEPTANCE OF INPUT 48

SECTION 16 ENVIRONMENT SWITCHING 50

TABLE OF CONTENTS (continued)

SECTION 17 INSERTIONS FROM THE STANDARD INPUT 51

SECTION 18 INPUT/OUTPUT FILE SWITCHING 52

SECTION 19 MISCELLLANEOUS 53

SECTION 20 OUTPUT AND ERROR MESSAGES 55

SECTION 21 EXAMPLES 56

21.1 Introduction 56
21.2 Page Margins 56
21.3 Paragraphs and Headings 58
21.4 Multiple Column Output 59
21.5 Footnote Processing 60
21.6 Last Page 62

APPENDIX A SUMMARY AND INDEX 63

A.1 Summary 63
A.2 Alphabetical Request and Section Number Cross Reference 70
A.3 Escape Sequences for Characters, Indicators, and Functions 70
A.4 Predefined General Number Registers 72
A.5 Predefined Read-only Number Registers 72

APPENDIX B SUMMARY OF RECENT CHANGES TO NROFF/TROFF.. 74

SECTION 1

BASIC INFORMATION

1.1 Introduction

Nroff and troff are text processors under the ZEUS Time-Sharing System that format text for typewriter-like terminals and for phototypesetters, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document with a user-designed style. Nroff and troff offer great freedom in document styling, including:

- Arbitrarily styled headers and footers
- Arbitrarily styled footnotes
- Multiple automatic sequence numbering for paragraphs, sections, etc.
- Multiple column output
- Dynamic font and point-size control
- Arbitrary horizontal and vertical local motions at any point
- A family of automatic overstriking, bracket construction, and line drawing functions

Nroff and troff are compatible with each other; it is possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input destined for either program. Nroff can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

1.2 Usage

The general form of invoking nroff or troff at ZEUS command level is

    nroff options files   (or troff options files)

where _options_ represents any of a number of option arguments and _files_ represents the list of files containing the document to be formatted. An argument consisting of a single minus (-) is taken to be a file name corresponding to the

standard input. If no file names are given, input is taken from the standard input. The following options can appear in any order as long as they appear before the files:

Option	Effect
-o*list*	Print only pages whose page numbers appear in a list that consists of comma-separated numbers and number ranges. A number range has the form N-M and means pages N through M; an initial -N means from the beginning to page N; a final N- means from N to the end.
-n*N*	Number first generated page N.
-s*N*	Stop every N pages. Nroff halts prior to every N pages (default N=1) to allow paper loading or changing, and resumes upon receipt of a new line. Troff stops the phototypesetter every N pages, produces a trailer to allow changing cassettes, and resumes after the phototypesetter START button is pressed.
-m*name*	Prepends the macro file /usr/lib/tmac.*name* to the input *files*.
-r*aN*	Register *a* (one-character) is set to N.
-i	Read standard input after the input files are exhausted.
-q	Invoke the simultaneous input-output mode of the rd request.

NROFF ONLY

Option	Effect
-T*name*	Specifies the name of the output terminal type. Currently defined names are: 37 for the Model 37 Teletypewriter TN300 (default) for the GE TermiNet 300 (or any terminal without half-line capabilities), 300S for the DASI-300S, 300 for the DASI-300, 450 for the DASI-450 (Diablo Hyterm).
-e	Produce equally-spaced words in adjusted lines, using full terminal resolution.

TROFF ONLY

-t	Direct output to the standard output instead of the phototypesetter.
-f	Refrain from feeding out paper and stopping phototypesetter at the end of the run.
-b	Troff reports whether the phototypesetter is busy or available. No text processing is done. (See NOTE in preface.)
-a	Send a printable (ASCII) approximation of the results to the standard output.
-pN	Print all characters in point size N, while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

Each option is invoked as a separate argument; for example,

```
nroff -o4,8-10 -T300S -mabc file1 file2
```

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named file1 and file2, specifies the output terminal as a DASI-300S, and invokes the macro package abc.

1.3 Form of Input

Input consists of text lines that are destined to be printed, interspersed with control lines that control subsequent processing. Control lines begin with a control character, normally a period (.) or acute accent (') followed by a one or two-character name that specifies a basic request or the substitution of a user-defined macro in place of the control line. The control character ' suppresses the break function (the forced output of a partially filled line) caused by certain requests. The control character can be separated from the request/macro name by white space (spaces and/or tabs). Names must be followed by either a space or a new line. Control lines with unrecognized names are ignored.

Various special functions can be introduced anywhere in the input by means of an escape character, normally a backslash (\). For example, the function \nR, causes the interpolation of the contents of the number register R in place of the function; here R is either a single-character name as in \nx, or a left-parenthesis-introduced, two-character name as

in \n(xx.

1.4 Formatter and Device Resolution

For internal processing, troff uses 432 units per inch, corresponding to the Graphic Systems phototypesetter, which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. Nroff uses 240 units per inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices. Troff rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the Graphic Systems typesetter. Nroff rounds numerical input to the actual resolution of the output device indicated by the -T option (default Model 37 Teletype).

1.5 Numerical Parameter Input

Both nroff and troff accept numerical input with the appended scale indicators shown in the following table, where S is the current type size in points, V is the current vertical line spacing in basic units, and C is a nominal character width in basic units.

Scale Indicator	Meaning	Number of Basic Units TROFF	NROFF
i	Inch	432	240
c	Centimeter	432x50/127	240x50/127
P	Pica = 1/6 inch	72	240/6
m	Em = S points	6xS	C
n	En = Em/2	3xS	C
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	V	V none
Default, see below			

In nroff, both the em and the en are taken to be equal to the C, which is output-device dependent; common values are 1/10 and 1/12 inch. Actual character widths in nroff need not be the same, and constructed characters such as -> are often extra wide. The default scaling is ems for the horizontally-oriented requests and functions ll, in, ti, ta, lt, po, mc, \h, and \l; the default is VS for the vertically-oriented requests and functions pl, wh, ch, dt, sp, sv, ne, rt, \v, \x, and \L; the default is p for the vs request; the default is u for the requests nr, if, and ie. All other requests ignore any scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator u needs to be appended to prevent an

additional inappropriate default scaling. The number (N) can be specified in decimal-fraction form, but the parameter finally stored is rounded to an integer number of basic units.

The absolute position indicator (|) can be prepended to a number N to generate the distance to the vertical or horizontal place N. For vertically-oriented requests and functions, |N becomes the distance in basic units from the current vertical place on the page or in a diversion (Section 6.4) to the the vertical place N. For all other requests and functions, |N becomes the distance from the current horizontal place on the input line to the horizontal place N. For example,

```
.sp  |3.2c
```

spaces in the required direction to 3.2 centimeters from the top of the page.

1.6 Numerical Expressions

Wherever numerical input is expected, the following can be used: an expression involving parentheses, the arithmetic operators +, -, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), and : (or). Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or - is stripped and interpreted as an increment or decrement indicator. In the presence of default scaling, the desired scale indicator must be attached to every number in an expression for which the desired scaling differs from the default scaling. For example, if the number register x contains 2 and the current point size is 10, then

```
.ll  (4.25i+\nxP+3)/2u
```

sets the line length to 1/2 the sum of 4.25 inches + 2 picas + 30 points.

SECTION 2

FONT AND CHARACTER SIZE CONTROL

2.1 Character Set

The troff character set consists of the Graphics Systems Commercial II character set plus a Special Mathematical Font character set. Each set has 102 characters. These character sets are shown in the Appendix. All ASCII characters are included, with some on the Special Font. With three exceptions, the ASCII characters are input as themselves, and non-ASCII characters are input in the form \(xx where xx is a two-character name given in the Appendix. The three ASCII exceptions are mapped as follows:

ASCII Input:		Printed by troff:	
Character	Name	Character	Name
'	acute accent	'	close quote
`	grave accent	`	open quote
-	minus	-	hyphen

The characters ', `, and - can be input by \', \`, and \-, respectively, or by their names (Section 2). The ASCII characters @, #, ", ', `, <, >, \, {, }, ~, ^, and - exist only on the Special Font and are printed as a 1-em space if that font is not mounted.

Nroff recognizes the entire troff character set, but can print only ASCII characters, additional characters as are available on the output device, such characters as are able to be constructed by overstriking or other combination, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device. The characters ', `, and - print as themselves.

2.2 Fonts

The default mounted fonts are Times Roman (R), Times Italic (I), Times Bold (B), and the Special Mathematical Font (S) on physical typesetter positions 1, 2, 3, and 4, respectively. The current font can be changed (among the mounted fonts) by use of the ft request, or by embedding at any desired point either \fx, \f(xx, or \fN, where x or xx is the name of a mounted font and N is a numerical font position. It is not necessary to change to the special font; characters on that font are automatically handled. A

request for a font that is named but not mounted is ignored. Troff can be informed that any particular font is mounted by use of the fp request. The list of known fonts is installation-dependent. In the subsequent discussion of font-related requests, F represents a one or two-character font name or the numerical font position, 1-4. The current font is available (as numerical position) in the read-only number register .f.

Nroff recognizes font control and (normally) underlines Italic characters.

2.3 Character Size

Character point sizes available on the Graphic Systems typesetter are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The ps request changes or restores the point size. Alternatively, the point size is changed between any two characters by embedding a \sN at the desired point to set the size to N, or a \s±N to increment/decrement the size by N; \s0 restores the previous size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the .s register. Nroff ignores type size control.

Request Form	Initial Value	If No Argument
.ps ±N	10 point	previous

Explanation: Point size set to ±N. Alternatively, embed \sN or \s±N. Any positive size value can be requested; if invalid, the next larger valid size results, with a maximum of 36. A paired sequence +N, -N works because the previous requested value is also remembered, but ignored in nroff. Relevant parameters are a part of the current environment

Request Form	Initial Value	If No Argument
.ss N	12/136 em	ignored

Explanation: Space-character size is set to N/36 ems. This size is the minimum word spacing in adjusted text. Ignored in nroff. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.cs	F N M	off

Explanation: Constant character space (width) mode is set on for font F (if mounted); the width of every character is taken to be N/36 ems. If M is absent, the em is that of the character's point size; if M is given, the em is M-points. All affected characters are centered in this space, including those with an actual width larger than this space. Special Font characters occurring while the current font is F are also so treated. If N is absent, the mode is turned off. The mode must be still or again in effect when the characters are physically printed. Ignored in nroff.

Request Form	Initial Value	If No Argument
.bd	F N	off

Explanation: The characters in font F are artificially made boldface by printing each one twice, separated by N^-1 basic units. A reasonable value for N is 3 when the character size is in the vicinity of 10 points. If N is missing the bold mode is turned off. The mode must be still or again in effect when the characters are physically printed. Ignored in nroff.

Request Form	Initial Value	If No Argument
.bd	S F N	off

Explanation: The characters in the Special Font are made boldface whenever the current font is F. The mode must be still or again in effect when the characters are physically printed.

Request Form	Initial Value	If No Argument
.ft F	Roman	previous

Explanation: Font changed to F. Alternatively, imbed \fF. The font name P is reserved to mean the previous font. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.fp N F	R,I,B,S	ignored

Explanation: Font position. This is a statement that a font named F is mounted on position N (1-4). It is a fatal error if F is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip that can be mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by troff is R, I, B, and S on positions 1, 2, 3 and 4.

SECTION 3

PAGE CONTROL

Top and bottom margins are not automatically provided; it is conventional to define two macros and to set traps for them at vertical positions 0 (top) and -N (N from the bottom). (See Sections 6 and 21.) A pseudo-page transition onto the first page occurs either when the first break occurs or when the first non-diverted text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the following, references to the current diversion (Section 6.4) mean that the mechanism being described works during both ordinary and diverted output.

The usable page width on the Graphic Systems phototypesetter is about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on nroff output are output-device dependent.

Request Form	Initial Value	If No Argument
.pl ±N	11 in	11 in

Explanation: Page length set to ±N. The internal limitation is about 75 inches in troff and about 136 inches in nroff. The current page length is available in the .p register. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.bp	±N=1	-

Explanation: Begin page. The current page is ejected and a new page is begun. If ±N is given, the new page number is ±N. Also see request ns. The default scale indicator is v (ignored if not specified). This request normally causes a break. The use of " ' " as control character (instead of .) suppresses the break function.

Request Form	Initial Value	If No Argument
.pn ±N	N=1	ignored

Explanation: Page number. The next page (when it occurs) has the page number ±N. A pn must occur before the initial

pseudo-page transition to effect the page number of the first page. The current page number is in the % register.

Request Form	Initial Value	If No Argument
.po ±N	0;26/127in*	previous

Explanation: Page offset. Values separated by ; are for nroff and troff, respectively. The current left margin is set to (+-N. The troff initial value provides about one inch of paper margin including the physical typesetter margin of 1/27 inch. In troff the maximum (line length)+(page offset) is about 7.54 inches (Section 5). The current page offset is available in the .o register. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.ne N	-	N=1 V

Explanation: Need N vertical space. If the distance, D, to the next trap position (Section 6.5) is less than N, a forward vertical space of size D occurs, which springs the trap. If there are no remaining traps on the page, D is the distance to the bottom of the page. If D < V, another line could still be output and spring the trap. In a diversion, D is the distance to the diversion trap, if any, or is very large. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.mk R	none	internal

Explanation: Mark the current vertical place in an internal register (both associated with the current diversion level), or in register R, if given. See rt request.

Request Form	Initial Value	If No Argument
.rt ±N	none	internal

Explanation: Return upward only to a marked vertical place in the current diversion. If ±N (with respect to current place) is given, the place is ±N from the top of the page or diversion or, if N is absent, to a place marked by a previous mk. The sp request (Section 5.3) can be used in all cases instead of rt by spacing to the absolute place stored

in a explicit register; for example using the sequence .mk R ... .sp |\nRu.

SECTION 4

TEXT FILLING, ADJUSTING, AND CENTERING

4.1 Filling and Adjusting

Normally, words are collected from input text lines and assembled into an output text line until some word does not fit. An attempt is then made the hyphenate the word in an effort to assemble a part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current line length minus any current indent. A word is any string of characters delimited by the space character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the unpaddable space character "\ " (backslash-space). The adjusted word spacings are uniform and the minimum interword spacing can be controlled with the ss request. In nroff, they are normally nonuniform because of quantization to character-size spaces; however, the command line option -e causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation can all be prevented or controlled. The text length on the last line output is available in the .n register, and text base-line position on the page for this line is in the nl register. The text base-line (lowest place) on the current page is in the .h register.

An input text line ending with ., ?, or ! is taken to be the end of a sentence, and an additional space character is automatically provided during filling. Multiple interword space characters found in the input are retained, except for trailing spaces; initial spaces also cause a break.

When filling is in effect, a \p can be embedded or attached to a word to cause a break at the end of the word and have the resulting output line spread out to fill the current line length.

A text input line that begins with a control character can be made to look like a regular text line by prefacing it with the nonprinting, zero-width filler character \&. Another way to do this is to specify output translation of some convenient character into the control character using tr.

4.2 Interrupted Text

The copying of an input line in nofill (non-fill) mode can be interrupted by terminating the partial line with a \c. The next encountered input text line is considered to be a continuation of the same line of input text. Similarly, a word within filled text is interrupted by terminating the word (or line) with \c; the next encountered text is taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line is forced out along with any partial word.

Request Form	Initial Value	If No Argument
.br	-	-

Explanation: Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.

Request Form	Initial Value	If No Argument
.fi	fill on	-

Explanation: Fill subsequent output lines. The register .u is 1 in fill mode and 0 in nofill mode. This request normally causes a break. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.nf	fill on	-

Explanation: No fill. Subsequent output lines are neither filled nor adjusted. Input text lines are copied directly to output lines without regard for the current line length. This request normally causes a break. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.ad c	adj,both	adjust

Explanation: Line adjustment is begun. If fill mode is not on, adjustment is deferred until fill mode is back on. If the type indicator c is present, the adjustment type is

changed as shown in the following table:

Indicator	Adjust Type
l	adjust left margin only
r	adjust right margin only
c	center
b or n	adjust both margins
absent	unchanged

Request Form	Initial Value	If No Argument
.na	adjust	-

Explanation: No adjust. Adjustment is turned off; the right margin is ragged. The adjustment type for ad is not changed. Output line filling still occurs if fill mode is on. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.ce N	off	N=1

Explanation: Center the next N input text lines within the current (line length minus indent). If N=0, any residual count is cleared. A break occurs after each of the N input lines. If the input line is too long, it is left adjusted. Relevant parameters are a part of the current environment.

SECTION 5

VERTICAL SPACING

5.1 Base-Line Spacing

The vertical spacing (V) between the base-lines of successive output lines is set using the vs request with a resolution of 1/144 inch = 1/2 point in troff, and to the output device resolution in nroff. V must be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set V to 2 points greater than the point size; troff default is 10-point type on 12-point spacing. The current V is available in the .v register. Multiple-V line separation (for example, double spacing) is requested with ls.

5.2 Extra Line-Space

If a word contains a construct that requires the output line containing it to have extra vertical space before and/or after it, the extra-line-space function (\x'N') can be embedded in or attached to that word. In this and other functions that have a pair of delimiters around their parameter (here '), the delimiter choice is arbitrary, except that it cannot look like the continuation of a number expression for N. If N is negative, the output line containing the word is preceded by N extra vertical spaces; if N is positive, the output line containing the word is followed by N extra vertical spaces. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the .a register.

5.3 Blocks of Vertical Space

A block of vertical space is ordinarily requested using sp, which honors the no-space mode and which does not space past a trap. A contiguous block of vertical space can be reserved using sv.

Request Form	Initial Value	If No Argument
.vs N	1/6in; 12 pts	previous

Explanation: Set vertical base-line spacing size V. Transient extra vertical space available with \x'N'. Relevant parameters are a part of the current environment. The default scale indicator is p (ignored if not specified).

Request Form	Initial Value	If No Argument
.ls N	N=1	previous

Explanation: Line spacing set to ±N. N-1 Vs (blank lines) are appended to each output text line. Appended blank lines are omitted if the text or previous appended blank line reached a trap position. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.sp N	-	N=1V

Explanation: Space vertically in either direction. If N is negative, the motion is backward (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. If the no-space mode is on, no spacing occurs (see ns, and rs below). This request normally causes a break. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.sv N	-	N=1V

Explanation: Save a contiguous vertical block of size N. If the distance to the next trap is greater than N, N vertical space is output. No-space mode has no effect. If this distance is less than N, no vertical space is immediately output, but N is retained for later output (see os). Subsequent sv requests overwrite any retained N. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.os	-	-

Explanation: Output saved vertical space. No-space mode has no effect. Used to output a block of vertical space requested by an earlier sv request.

Request Form	Initial Value	If No Argument
.ns	space	-

Explanation: No-space mode turned on. When on, the no-space mode inhibits sp requests and bp requests without a next page number. The no-space mode is turned off when a line of output occurs, or with rs. Relevant parameters are associated with the current diversion level.

Request Form	Initial Value	If No Argument
.rs	space	-

Explanation: Restore spacing. The no-space mode is turned off. Relevant parameters are associated with the current diversion level.

Request Form	Initial Value	If No Argument
Blank text line.		-

Explanation: Causes a break and output of a blank line exactly like sp 1.

5.4 Line Length and Indenting

The maximum line length for fill mode is set with ll. The indent is set with in; an indent applicable to only the next output line is set with ti. The line length includes indent space but not page offset space. The line length minus the indent is the basis for centering with ce. If a partially collected line exists, the effect of ll, in, or ti is delayed until after that line is output. In fill mode, the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers .l and .i respectively. The length of three-part titles produced by tl is independently set by lt.

Request Form	Initial Value	If No Argument
.ll ±N	6.5 in	previous

Explanation: Line length is set to . In troff the maximum (line length)+(page offset) is about 7.54 inches. Relevant parameters are a part of the current enviroment. The

default scale indicator is m (ignored if not specified).

Request Form	Initial Value	If No Argument
.in ±N	N=0	previous

Explanation: Indent is set to ±N. The indent is prepended to each output line. This request normally causes a break. Relevant parameters are a part of the current environment. The default scale indicator is m (ignored if not specified).

Request Form	Initial Value	If No Argument
.ti ±N	-	ignored

Explanation: Temporary indent. The next output text line will be indented a distance ±N with respect to the current indent. The resulting total indent cannot be negative. The current indent is not changed. This request normally causes a break. Relevant parameters are a part of the current environment. The default scale indicator is m (ignored if not specified).

SECTION 6

MACROS, STRINGS, DIVERSION, AND POSITION TRAPS

6.1 Macros and Strings

A macro is a named set of arbitrary lines that can be invoked by name or with a trap. A string is a named string of characters, not including a new line character, that can be interpolated by name at any point. Request, macro, and string names share the same name list. Macro and string names can be one or two characters long and can use previously defined request, macro, or string names. Any of these entities can be renamed with rn or removed with rm. Macros are created by de and di, and appended to by am and da; di and da cause normal output to be stored in a macro. Strings are created by ds and appended to by as. A macro is invoked in the same way as a request; a control line beginning .xx interpolates the contents of macro xx. The remainder of the line can contain up to nine arguments. The strings x and xx are interpolated at any desired point with *x and *(xx, respectively. String references and macro invocations can be nested.

6.2 Copy Mode Input Interpretation

During the definition and extension of strings and macros (not by diversion), the input is read in copy mode. The input is copied without interpretation except that:

- The contents of number registers indicated by \n are interpolated
- Strings indicated by * are interpolated
- Arguments indicated by \$ are interpolated
- Concealed new lines indicated by \(new line) are eliminated
- Comments indicated by \" are eliminated
- \t and \a are interpreted as ASCII horizontal tab and SOH, respectively
- \\ is interpreted as \
- \. is interpreted as .

These interpretations can be suppressed by prepending a \. For example, since \\ maps into a \, \\n copies as \n, which is interpreted as a number register indicator when the macro or string is reread.

6.3 Arguments

When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character, and arguments can be surrounded by double quotes to permit embedded space characters. Pairs of double quotes can be embedded in double quoted arguments to represent one double quote. If the desired arguments do not fit on a line, a concealed new line can be used to continue on the next line.

When a macro is invoked, the input level is pushed down and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at any point within the macro with \$N, which interpolates the Nth argument (1≤IN≤9). If an invoked argument does not exist, a null string results. For example, the macro xx is defined by

```
.de xx    \"begin definition
Today is \\$1 the  \\$2.
..         \"end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

```
Today is Monday the 14th.
```

Note that the \$ is concealed in the definition with a prepended \. The number of currently available arguments is in the .$ register.

No arguments are available at the top (nonmacro) level in this implementation. Because string referencing is implemented as an input-level push down, no arguments are available from within a string. No arguments are available within a trap-invoked macro.

Arguments are copied in copy mode onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a long string that is interpolated at copy time. It is advisable to

conceal string references with an extra \ to delay interpolation until argument reference time.

6.4 Diversions

Processed output can be diverted into a macro for purposes such as footnote processing (Section 21.5) or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap can be set at a specified vertical position. The number registers dn and dl contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in nofill mode, regardless of the current V. Constant-spaced (cs) or bold (bd) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way to do this is to embed in the diversion the appropriate cs or bd requests with the transparent mechanism described in Section 9.6.

Diversions can be nested, and certain parameters and registers are associated with the current diversion level. The top nondiversion level can be thought of as the 0th diversion level. These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (mk and rt), the current vertical place (.d register), the current text base-line (.h register), and the current diversion name (.z register).

6.5 Traps

Three types of trap mechanisms are available: page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps can be planted using wh at any page position including the top. This trap position is changed using ch. Trap positions at or below the bottom of the page have no effect unless or until moved within the page or rendered effective by an increase in page length. Two traps can be planted at the same position only by first planting them at different positions and then moving one of the traps; the first planted trap conceals the second unless and until the first one is moved (Appendix). If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size reaches or sweeps past the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the .t register; if there are no traps between the current

position and the bottom of the page, the distance returned is the distance to the page bottom.

A macro-invocation trap effective in the current diversion can be planted using dt. The .t register works in a diversion; if there is no subsequent trap, a large distance is returned. The following table describes the input-line-count traps:

Request Form	Initial Value	If No Argument
.de xx yy	-	.yy=..

Explanation: Define or redefine the macro xx. The contents of the macro begin on the next input line. Input lines are copied in copy mode until the definition is terminated by a line beginning with .yy, whereupon the macro yy is called. In the absence of yy, the definition is terminated by a line beginning with two periods (..). A macro can contain de requests, provided the terminating macros differ or the contained definition terminator is concealed. The .. can be concealed as \\.. (which copies as \..) and be reread as .. itself.

Request Form	Initial Value	If No Argument
.am xx yy	-	.yy=..

Explanation: Append to macro (append version of de).

Request Form	Initial Value	If No Argument
.ds xx string	-	ignored

Explanation: Define a string xx containing string. Any initial double quote in string is stripped off to permit initial blanks.

Request Form	Initial Value	If No Argument
.as xx string	-	ignored

Explanation: Append string to string xx (append version of ds).

Request Form	Initial Value	If No Argument
.rm xx	-	ignored

Explanation: Remove request, macro, or string. The name xx is removed from the name list and any related storage space is freed. Subsequent references have no effect.

Request Form	Initial Value	If No Argument
.rn xx yy	-	ignored

Explanation: Rename request, macro, or string xx to yy. If yy exists, it is first removed.

Request Form	Initial Value	If No Argument
.di xx	-	end

Explanation: Divert output to macro xx. Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request di or da is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used. Mode or relevant parameters are associated with the current diversion level.

Request Form	Initial Value	If No Argument
.da xx	-	end

Explanation: Divert, appending to xx (append version of di). Mode or relevant parameters are associated with the current diversion level.

Request Form	Initial Value	If No Argument
.wh N xx	-	-

Explanation: Install a trap to invoke xx at page position; a negative N is interpreted with respect to the page bottom. Any macro previously planted at N is replaced by xx. A zero N refers to the top of a page. In the absence of xx, the first found trap at N, if any, is removed. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.ch xx N	-	-

Explanation: Change the trap position for macro xx to be N. In the absence of N, the trap, if any, is removed. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.dt N xx	-	off

Explanation: Install a diversion trap at position N in the current diversion to invoke macro xx. Another dt redefines the diversion trap. If no arguments are given, the diversion trap is removed. Mode or relevant parameters are associated with the current diversion level. The default scale indicator is v (ignored if not specified).

Request Form	Initial Value	If No Argument
.it N xx	-	off

Explanation: Set an input-line-count trap to invoke the macro xx after N lines of text input have been read (control or request lines do not count). The text can be in-line text or text interpolated by in-line or trap-invoked macros. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.em xx	none	none

Explanation: The macro xx is invoked when all input has ended. The effect is the same as if the contents of xx had been at the end of the last file processed.

SECTION 7

NUMBER REGISTERS

A variety of parameters are available to the user as predefined, named number registers (Summary and Index). In addition, named registers can be user-defined. Register names are one or two characters long and do not conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register can be used any time numerical input is expected or desired and can be used in numerical expressions.

Number registers are created and modified using nr, which specifies the name, numerical value, and the auto-increment size. Registers are also modified if accessed with an auto-incrementing sequence. If the registers x and xx both contain N and have the auto-increment size M, the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
\nx	none	N
\n(xx	none N	
\n+x	x incremented by M	N+M
\n-x	x decremented by M	N-M
\n+(xx	xx incremented by M	N+M
\n-(xx	xx decremented by M	N-M

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lowercase Roman, uppercase Roman, lowercase sequential alphabetic, or uppercase sequential alphabetic, according to the format specified by af.

Request Form	Initial Value	If No Argument
.nr R±N	M	-

Explanation: The number register R is assigned the value ±N with respect to the previous value, if any. The increment for auto-incrementing is set to M. The default scale indicator is u (ignored if not specified).

Request Form	Initial Value	If No Argument
.af R c	arabic	-

Explanation: Assign format c to register R. The available formats are:

Numbering Format	Sequence
1	0,1,2,3,4,5,...
001	000,001,002,003,004,005,...
i	0,i,ii,iii,iv,v,...
I	0,I,II,III,IV,V,...
a	0,a,b,c,...,z,aa,ab,...,zz,aaa,...
A	0,A,B,C,...,Z,AA,AB,...,ZZ,AAA,...

An arabic format having N digits specifies a field width of N digits (second example above). The read-only registers and the width function are always arabic.

Request Form	Initial Value	If No Argument
.rr R	-	ignored

Explanation: Remove register R. If many registers are being created dynamically, it is necessary to remove unused registers to recapture internal storage space for newer registers.

SECTION 8

TABS, LEADERS, AND FIELDS

8.1 Tabs and Leaders

The ASCII horizontal tab character and the ASCII SOH (leader character) can both generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal tab stops specified with ta. The default difference is that tabs generate motion, and leaders generate a string of periods; tc and lc offer the choice of repeated character or motion. There are three types of internal tab stops: left adjusting, right adjusting, and centering. In the following table, D is the distance from the current position on the input line (where a tab or leader was found) to the next tab stop; next-string consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line; W is the width of next-string.

Tab Type	Length of Motion or Repeated Characters	Location of Next-String
Left	D	Following D
Right	D-W	Right adjusted with D
Centered	D-W/2	Centered on right end of D

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but can be used as next-string terminators.

Tabs and leaders are not interpreted in copy mode. \t and \a always generate a noninterpreted tab and leader respectively, and are equivalent to actual tabs and leaders in copy mode.

8.2 Fields

A field is contained between a pair of field delimiter characters, and consists of substrings separated by padding indicator characters. The field length is the distance on the input line from the position where the field begins to the next tab stop. The difference between the total length of all the substrings and the field length is incorporated

as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is # and the padding indicator is ^, #^xxx^right# specifies a right-adjusted string with the string xxx centered in the remaining space.

Request Form	Initial Value	If No Argument
.ta Nt ...	0.8; 0.5 in	none

Explanation: Set tab stops and types. t=R, right adjusting; t=C, centering; t absent, left adjusting. Troff tab stops are preset every 0.5in.; nroff every 0.8in. The stop values are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value. Relevant parameters are a part of the current environment. The default scale indicator is m (ignored if not specified).

Request Form	Initial Value	If No Argument
.tc c	none	none

Explanation: The tab repetition character becomes c, or is removed specifying motion. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.lc c		none

Explanation: The leader repetition character becomes c, or is removed specifying motion. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.fc a b	off	off

Explanation: The field delimiter is set to a; the padding indicator is set to the space character or to b, if given. In the absence of arguments, the field mechanism is turned off.

SECTION 9

INPUT/OUTPUT CONVENTIONS AND CHARACTER TRANSLATIONS

9.1 Input Character Translations

The new line delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted, and can be used as delimiters or translated into a graphic with tr. All others are ignored.

The escape character (\) introduces escape sequences and causes the following character to mean another character, or to indicate some function. (A complete list of such sequences is given in the Summary and Index.) The \ is not the ASCII control character ESC of the same name. The escape character can be input with the sequence \\. The escape character can be changed with ec, and all that has been said about the default \ becomes true for the new escape character. \e prints whatever the current escape character is. If necessary or convenient, the escape mechanism can be turned off with eo, and restored with ec.

Request Form	Initial Value	If No Argument
.ec c	\	\

Explanation: Set escape character to \, or to c, if given.

Request Form	Initial Value	If No Argument
.eo	on	-

Explanation: Turn escape mechanism off.

9.2 Ligatures

Five ligatures are available in the current troff character set: fi, fl, ff, ffi, and ffl. They are input by \(fi, \(fl, \(ff, \(Fi, and \(Fl respectively. The ligature mode is normally on in troff, and automatically invokes ligatures during input.

Request Form	Initial Value	If No Argument
.lg N	off; on	on

Explanation: Ligature mode is turned on if N is absent or non-zero, and turned off if N=0. If N=2, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, or file names, and in copy mode. No effect in nroff.

9.3 Backspacing, Underlining, and Overstriking

Unless in copy mode, the ASCII backspace character is replaced by a backward horizontal motion the width of the space character.

Nroff automatically underlines characters in the underline font, specifiable with uf, normally that on font position 2 (normally Times Italic, Section 2.2). In addition to ft and \fF, the underline font is selected by ul and cu. Underlining is restricted to an output-device-dependent subset of reasonable characters.

Request Form	Initial Value	If No Argument
.ul N	off	N=1

Explanation: Underline in nroff (italicize in troff) the next N input text lines. Actually, switch to underline font, saving the current font for later restoration; other font changes within the span of a ul take effect, but the restoration undoes the last change. Output generated by tl is affected by the font change, but does not decrement N. If N>1, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching prevents this. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.cu N	off	N=1

Explanation: A variant of ul that causes every character to be underlined in nroff. Identical to ul in troff. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.uf F	Italic	Italic

Explanation: Underline font set to F. In nroff, F may not be on position 1 (initially Times Roman).

9.4 Control Characters

Both the control character . and the no-break control character ' can be changed, if desired. Such a change must be compatible with the design of any macros used in the span of the change, and particularly of any trap-invoked macros.

Request Form	Initial Value	If No Argument
.cc c	.	.

Explanation: The basic control character is set to c, or reset to . . Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.c2 c	'	'

Explanation: The nobreak control character is set to c, or reset to ' . Relevant parameters are a part of the current environment.

9.5 Output Translation

One character can be made a stand-in for another character using tr. All text processing takes place with the input (stand-in) character which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

Request Form	Initial Value	If No Argument
.tr abcd....	none	-

Explanation: Translate a into b, c into d, etc. If an odd number of characters is given, the last one is mapped into the space character. To be consistent, a particular translation must stay in effect from input to output time.

9.6 Transparent Throughput

An input line beginning with a \! is read in copy mode and transparently output without the initial \!; the text processor makes no other response based on the line's presence. This mechanism is used to pass control information to a

post-processor or to embed control lines in a macro created by a diversion.

9.7 Comments and Concealed New Lines

A long input line that must stay one line (for example, a string definition, or nofilled text) can be split into many physical lines by ending all but the last one with the escape \. The sequence \(new line) is always ignored--except in a comment. Comments can be embedded at the end of any line by prefacing them with \". The new line at the end of a comment cannot be concealed. A line beginning with \" appears as a blank line and behaves like .sp 1; a comment can be placed on a line by itself by beginning the line with .\".

SECTION 10

LOCAL MOTIONS AND THE WIDTH FUNCTION

10.1 Local Motions

The functions \v'N' and \h'N' are used for local vertical and horizontal motion respectively. The distance N can be negative; the positive directions are rightward and downward. A local motion is one contained within a line. To avoid unexpected vertical dislocations, it is necessary that the net vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

<table>
<tr><th rowspan="2">Vertical Local Motion</th><th colspan="2">Effect in</th><th rowspan="2">Horizontal Local Motion</th><th colspan="2">Effect in</th></tr>
<tr><th>TROFF</th><th>NROFF</th><th>TROFF</th><th>NROFF</th></tr>
<tr><td>\v'N'</td><td colspan="2">Move distance N</td><td>\h'N'</td><td colspan="2">Move distance N</td></tr>
<tr><td></td><td></td><td></td><td>\(space)</td><td colspan="2">Unpaddable space-size space</td></tr>
<tr><td>\u</td><td>1/2 em up</td><td>1/2 line down</td><td>\0</td><td colspan="2">Digit-size space</td></tr>
<tr><td>\r</td><td>1 em up</td><td>1 line up</td><td>\|</td><td>1/6 em space</td><td>ignored</td></tr>
<tr><td></td><td></td><td></td><td>\^</td><td>1/12 em space</td><td>ignored</td></tr>
</table>

10.2 Width Function

The width function \w'string' generates the numerical width of string (in basic units). Size and font changes can be safely embedded in string, and do not affect the current environment. For example, .ti|-\w'1.|'u can temporarily indent leftward a distance equal to the size of the "1.|" string.

The width function also sets three number registers. The registers st and sb are set (respectively) to the highest and lowest extent of string relative to the baseline; then, for example, the total height of the string is \n(stu-\n(sbu. In troff the number register ct is set to a value between 0 and 3. 0 means that all of the characters in string are short, lowercase characters without descenders (like e); 1 means that at least one character has a descender (like y); 2 means that at least one character is tall (like H); and 3 means that both tall characters and characters with descenders are present.

10.3 Mark Horizontal Place

The escape sequence \kx causes the current horizontal position in the input line to be stored in register x. As an example, the construction \kxword\h'|\nxu+2u'word makes "word" bold by backing up to almost its beginning and overprinting it, resulting in word.

SECTION 11

OVERSTRIKE, LINE-DRAWING, AND ZERO-WIDTH FUNCTIONS

11.1 Overstriking

Automatically centered overstriking of up to nine characters is provided by the overstrike function (\o "string"). The characters in string are overprinted with centers aligned; the total width is that of the widest character. The string must not contain local vertical motion. For example, \o'e\'' produces é.

11.2 Line Drawing

The function \l "Nc" draws a string of repeated c's for a distance N. (\l is \(lowercase L). If c looks like a continuation of an expression for N, it can be insulated from N with a \&. If c is not specified, the _ (baseline rule or underline character) is used. If N is negative, a backward horizontal motion of size N is made before drawing the string. Any space resulting from N/(size of c) having a remainder is put at the beginning (left end) of the string. In the case of characters that are designed to be connected, such as baseline-rule _, underrule _, and root-en, the remainder space is covered by overlapping. If N is less than the width of c, a single c is centered on a distance N. As an example, a macro to underscore a string can be written

```
.de us
\\$1\l'|0\(ul'
..
```

such that

```
.us "underlined words"
```

yields

underlined words .

The function \L'Nc' draws a vertical line consisting of the (optional) character c stacked vertically apart 1em (1 line in nroff), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the box rule | (\(br); the other suitable character is the bold vertical | (\(bv). The line is begun without any initial motion relative to the current base line. A positive N specifies a line drawn downward and a negative N

specifies a line drawn upward. After the line is drawn, no compensating motions are made; the instantaneous base line is at the end of the line.

The horizontal and vertical line-drawing functions can be used in combination to produce large boxes. The zero-width box-rule and the 1/2-em wide underrule were designed to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
.sp -1      \"compensate for next automatic base-line spacing
.nf         \"avoid possibly overflowing word buffer
.fi
..
```

draws a box around some text whose beginning vertical place was saved in number register a (that is, using .mk a).

11.3 Zero-Width Characters

The function \zc outputs c without spacing over it, and is used to produce left-aligned overstruck combinations. As examples, \z\(ci\(pl produces ⊕, and \(br\z\(rn\(ul\(br produces the smallest possible constructed box.

SECTION 12

HYPHENATION

The automatic hyphenation can be switched off and on. When switched on with hy, several variants can be set. A hyphenation indicator character can be embedded in a word to specify desired hyphenation points, or can be prepended to suppress hyphenation. In addition, the user can specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) nonalphabetic strings are considered candidates for automatic hyphenation. Words that are input containing hyphens (minus), em-dashes (\(em), or hyphenation indicator characters, are always subject to splitting after those characters, whether or not automatic hyphenation is on or off.

Request Form	Initial Value	If No Argument
.nh	hyphenate	-

Explanation: Automatic hyphenation is turned off. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.hy N	on,N=1on,	N=1

Explanation: Automatic hyphenation is turned on for N ≥1, or off for N=0. If N=2, last lines (ones that cause a trap) are not hyphenated. For N=4 and 8, the last and first two characters of a word are not split off. These values are additive; for example, N=14 invokes all three restrictions. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.hc c	\%	\%

Explanation: Hyphenation indicator character is set to c or to the default \%. The indicator does not appear in the output. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.hw	word1 ...	ignored

Explanation: Specify hyphenation points in words with embedded minus signs. Versions of a word with terminal s are implied; for example, dig-it implies dig-its. This list is examined initially and after each suffix stripping. The space available is small--about 128 characters.

SECTION 13

THREE-PART TITLES

The titling function tl provides for automatic placement of three fields at the left, center, and right of a line with a title-length specified with lt. tl can be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

Request Form	Initial Value
.tl 'left'center'right'	-

Explanation: The strings left, center, and right are respectively left-adjusted, centered, and right-adjusted in the current title-length. Any of the strings can be empty, and overlapping is permitted. If the page-number character (initially %) is found within any of the fields, it is replaced by the current page number. The format is assigned to register %. Any character can be used as the string delimiter.

Request Form	Initial Value	If No Argument
.pc c	%	off

Explanation: The page number character is set to c, or removed. The page-number register remains %.

Request Form	Initial Value	If No Argument
.lt ±N	6.5 in	previous

Explanation: Length of title set to ±N. The line-length and the title-length are independent. Indents do not apply to titles; page-offsets do. Relevant parameters are a part of the current environment. The default scale indicator is m (ignored if not specified).

SECTION 14

OUTPUT LINE NUMBERING

Automatic sequence numbering of output lines can be requested with nm. When in effect, a three-digit, arabic number plus a digit-space is prepended to output text lines. The text lines are thus offset by four digit-spaces, and otherwise retain their line length; a reduction in line length can be used to keep the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by tl are not numbered. Numbering can be temporarily suspended with nn, or with an In addition, a line number indent I, and the number-text separation S can be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number M are to be printed. The others appear as blank number fields.

Request Form	Initial Value	If No Argument
.nm ±N M S I	M=1, S=1, I=0	off

Explanation: Line number mode. If ±N is given, line numbering is turned on, and the next output line numbered is numbered ±N. Default values are M=1, S=1, and I=0. Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register ln. Relevant parameters are a part of the current environment.

Request Form	Initial Value	If No Argument
.nn N	-	N=1

Explanation: The next N text output lines are not numbered. Relevant parameters are a part of the current environment.

As an example, the paragraph portions of this section are numbered with M=3: .nm 1 3 was placed at the beginning; .nm was placed at the end of the first paragraph; and .nm +0 was placed in front of this paragraph; and .nm finally placed at the end. Line lengths were also changed (by \w"0000"u) to keep the right side aligned. Another example is .nm +5 5 x 3, which turns on numbering with the line number of the next line to be 5 greater than the last numbered line, with M=5, with spacing S untouched, and with the indent I set to 3.

SECTION 15

CONDITIONAL ACCEPTANCE OF INPUT

In the following, c is a one-character, built-in condition name, ! signifies not, N is a numerical expression, string1 and string2 are strings delimited by any nonblank, non-numeric character not in the strings, and _anything_ represents what is conditionally accepted.

Request
Form

.if c _anything_

Explanation: If condition c true, accept _anything_ as input; in multi-line case use \{_anything_\}.

Request
Form

.if !c | _anything_

Explanation: If condition c false, accept _anything_.

Request
Form

.if N _anything_

Explanation: If expression N > 0, accept _anything_. The default scale indicator is u (ignored if not specified).

Request
Form

.if !N _anything_

Explanation: If expression N $\leq$ 0, accept _anything_. The default scale indicator is u (ignored if not specified).

Request
Form

.if 'string1'string2'_anything_

Explanation: If string1 is identical to string2, accept _anything_.

Request
Form

.if !'string1'string2'_anything_

Explanation: If string1 is not identical to string2, accept _anything_.

Request
Form

.ie c _anything_

Explanation: If portion of if-else; all above forms (like if). The default scale indicator is u (ignored if not specified).

Request
Form

.el _anything_

Explanation: Else portion of if-else.

The built-in condition names are:

Condition Name	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is troff
n	Formatter is nroff

If the condition c is true, or if the number N is greater than zero, or if the strings compare identically (including motions and character size and font), _anything_ is accepted as input. If a ! precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of _anything_ are skipped over. The _anything_ is either a single input line (for example, text or macro) or a number of input lines. In the multiline case, the first line must begin

with a left delimiter (\{) and the last line must end with a right delimiter (\}).

The request ie (if-else) is identical to if, except that the, acceptance state is remembered. A subsequent and matching el (else) request then uses the reverse sense of that state. ie|-|el pairs can be nested.

Some examples are:

```
.if e .tl 'Even Page %'''
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\
'sp 0.5i
.tl 'Page %'''
'sp |1.2i \}
.el .sp|2.5i
```

which treats page 1 differently from other pages.

SECTION 16

ENVIRONMENT SWITCHING

A number of the parameters that control the text processing are gathered together into an environment, that can be switched by the user. The environment parameters are those associated with requests noting E in their Notes column; in addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented parameters, diversion-oriented parameters, number registers, and macro and string definitions. All environments are initialized with default parameter values.

Request Form	Initial Value	If No Argument
.ev N	N=0	previous

Explanation: Environment switched to environment $0 \leq N \leq 2$. Switching is done in push-down fashion so that restoring a previous environment must be done with .ev rather than specific reference.

SECTION 17

INSERTIONS FROM THE STANDARD INPUT

The input can be temporarily switched to the system standard input with rd, which switches back when two new lines in a row are found (the extra blank line is not used). This mechanism is intended for insertions in form-letter types of documentation. On ZEUS, the standard input is the user's keyboard, a pipe, or a file.

Request Form	Initial Value	If No Argument
.rd prompt	-	prompt=BEL

Explanation: Read insertion from the standard input until two new lines in a row are found. If the standard input is the user's keyboard, prompt (or a BEL) is written onto the user's terminal. rd behaves like a macro, and arguments can be placed after prompt.

Request Form	Initial Value	If No Argument
.ex	-	-

Explanation: Exit from nroff/troff. Text processing is terminated exactly as if all input had ended.

If insertions are taken from the terminal keyboard while output is being printed on the terminal, the command line option -q turns off the echoing of keyboard input and prompts only with BEL. The regular input and insertion input cannot simultaneously come from the standard input.

As an example, multiple copies of a form letter are prepared by entering the insertions for all the copies in one file used as the standard input, and causing the file containing the letter to reinvoke itself using nx; the process is ended by an ex in the insertion file.

SECTION 18

INPUT/OUTPUT FILE SWITCHING

Request Form	Initial Value	If No Argument
.so	filename	-

Explanation: Switch source file. The top input (file reading) level is switched to _filename_. A _so_ encountered in a macro does not take effect until the input level returns to the file level. When the new file ends, input is again taken from the original file. _so_'s can be nested.

Request Form	Initial Value	If No Argument
.nx filename		end-of-file

Explanation: Next file is _filename_. The current file is considered ended, and the input is immediately switched to _filename_.

Request Form	Initial Value	If No Argument
.pi program	-	

Explanation: Pipe output to program (nroff only). This request must occur before any printing occurs. No arguments are transmitted to program.

SECTION 19

MISCELLANEOUS

Request Form	Initial Value	If No Argument
.mc c N	-	off

Explanation: Specifies that a margin character c appear a distance N to the right of the right margin after each nonempty text line (except those produced by tl). If the output line is too long (as can happen in nofill mode) the character is appended to the line. If N is not given, the previous N is used. The initial N is 0.2 inches in nroff and 1 em in troff. Relevant parameters are a part of the current environment. The default scale indicator is m (ignored if not specified).

Request Form	Initial Value	If No Argument
.tm string	-	newline

Explanation: After skipping initial blanks, string (rest of the line) is read in copy mode and written on the user's terminal.

Request Form	Initial Value	If No Argument
.ig yy	-	.yy=..

Explanation: Ignore input lines. Ig behaves like de except that the input is discarded. The input is read in copy mode, and any auto-incremented registers are affected.

Request Form	Initial Value	If No Argument
.pm t	-	all

Explanation: Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if t is given, only the total of the sizes is printed. The size is given in blocks of 128 characters.

Request Form	Initial Value	If No Argument
.fl	-	-

Explanation: Flush output buffer. Used in interactive debugging to force output. This request normally causes a break.

SECTION 20

OUTPUT AND ERROR MESSAGES

The output from tm, pm, and the prompt from rd, as well as various error messages are written onto the standard message output. The standard message output is different from the standard output, where nroff formatted output goes. By default, both are written onto the user's terminal, but they can be independently redirected.

Various error conditions can occur during the operation of nroff and troff. Certain less serious errors that have only local impact do not cause processing to terminate. Two examples are word overflow, caused by a word that is too large to fit into the word buffer (in fill mode), and line overflow, caused by an output line that grows too large to fit in the line buffer; in both cases, a message is printed, the excess is discarded, and the affected word or line is marked at the point of truncation with a * in nroff and a <= in troff. Processing continues, if possible, since output useful for debugging may be produced. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

SECTION 21

EXAMPLES

21.1 Introduction

It is almost always necessary to prepare at least a small set of macro definitions to describe most documents. Such common formatting needs as page margins and footnotes are deliberately not built into nroff and troff. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations. (Most documents can be prepared with either the -ms or -man macro sets.)

The following examples are intended to be useful and realistic, but do not cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers are used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization.

21.2 Page Margins

Header and footer macros are defined to describe the top and bottom page margin areas. A trap is planted at page position 0 for the header, and at -N (N from the page bottom) for the footer. The simplest such definitions are

```
.de hd          \"define header
'sp li
..              \"end definition
.de fo          \"define footer
'bp
..              \"end definition
.wh 0 hd
.wh -li fo
```

which provide blank one-inch top and bottom margins. The header only occurs on the first page if the definition and trap exist prior to the initial pseudo-page transition. In fill mode, the output line that springs the footer trap is forced out because some part or whole word does not fit on it. If anything in the footer and header that follows causes a break, that word or part word is forced out. In this and other examples, requests like bp and sp, which normally cause breaks, are invoked using the no-break control

character ' to avoid this problem. When the header/footer design contains material requiring independent text processing, the environment can be switched, avoiding most interaction with the running text.

Another example is

```
.de hd                          \"header
.if t .tl '\(rn''\(rn'          \"troff cut mark
.if \\n%>1 \{\
'sp |0.5i-1                     \"tl base at 0.5i
.tl ''- % -''                   \"centered page number
.ps                             \"restore size
.ft                             \"restore font
.vs  \}                         \"restore vs
'sp |1.0i                       \"space to 1.0i
.ns                             \"turn on no-space mode
..
.de fo                          \"footer
.ps 10                          \"set footer/header size
.ft R                           \"set font
.vs 12p                         \"set base-line spacing
.if \\n%=1 \{\
'sp |\\n(.pu-0.5i-1             \"tl base 0.5i up
.tl ''- % -'' \}                \"first page number
'bp
..
.wh 0 hd
.wh -1i fo
```

which sets the size, font, and base-line spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If troff is used, a cut mark is drawn in the form of root-en's at each margin. The sp's refer to absolute positions to avoid dependence on the base-line spacing. Another reason for this in the footer is that the footer is invoked by printing a line whose vertical spacing sweeps past the trap position by as much as the base-line spacing. The no-space mode is turned on at the end of hd to render ineffective and accidental occurrences of sp at the top of the running text.

This method of restoring size, font, etc. presupposes that such requests that set previous value are not used in the running text. A better scheme is to save and restore both the current and previous values for size as shown in the following:

```
.de fo
.nr s1 \\n(.s                  \"current size
.ps
```

```
.nr s2 \\n(.s                 \"previous size
. ---                         \"rest of footer
..
.de hd
. ---                         \"header
.ps \\n(s2                    \"restore previous size
.ps \\n(sl                    \"restore current size
..
```

Page numbers are printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn                        \"bottom number
.tl ''- % -''                 \"centered page number
..
.wh -0.5i-lv bn               \"tl base 0-5i up
```

21.3 Paragraphs and Headings

The housekeeping associated with starting a new paragraph is collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, base-line spacing, and indent, checks that enough space remains for more than one line, and requests a temporary indent.

```
.de pg                   \"paragraph
.br                      \"break
.ft R                    \"force font,
.ps 10                   \"size,
.vs 12p                  \"spacing,
.in 0                    \"and indent
.sp 0.4                  \"prespace
.ne 1+\\n(.Vu            \"want more than 1 line
.ti 0.2i                 \"temp indent
..
```

The first break in pg forces out any previous partial lines, and must occur before the vs. The forcing of font, etc. is a defense against prior error and permits things like section heading macros to set parameters only once. The prespacing parameter is suitable for troff; a larger space, at least as big as the output device vertical resolution, is more suitable in nroff. The choice of remaining space to test for in _ne_ is the smallest amount greater than one line.

A macro to automatically number section headings looks like:

```
.de sc                        \"section
. ---                         \"force font, etc.
.sp 0.4                       \"prespace
```

```
.ne 2.4+\\n(.Vu          \"want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1                \"init S
```

The usage is .sc, followed by the section heading text, followed by .pg. The ne test value includes one line of heading, 0.4 line in the following pg, and one line of the paragraph text. A word consisting of the next section number and a period is produced to begin the heading line. The format of the number is set by af.

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp                 \"labeled paragraph
.pg
.in 0.5i               \"paragraph indent
.ta 0.2i 0.5i          \"label, paragraph
.ti 0
\t\\$1\t\c             \"flow into paragraph
..
```

The intended usage is ".lp label"; label begins at 0.2 inch, and cannot exceed a length of 0.3 inch without intruding into the paragraph. The label is right-adjusted against 0.4 inch by setting the tabs instead with .ta 0.4iR 0.5i. The last line of lp ends with \c so that it becomes a part of the first line of the text that follows.

21.4 Multiple Column Output

The production of multiple column pages requires the footer macro to determine whether it was invoked by other than the last column, so that it begins a new column rather than produce the bottom margin. The header initializes a column register that the footer increments and test. The following is arranged for two columns, but is easily modified for more.

```
.de hd                 \"header
. ---
.nr cl 0 1             \"init column count
.mk                    \"mark top of text
..
.de fo                 \"footer
.ie \\n+(cl<2 \{\
.po +3.4i              \"next column; 3.1+0.3
.rt                    \"back to mark
.ns \}                 \"no-space mode
```

```
.el \{\
.po \\nMu              \"restore left margin
. ---
'bp \}
..
.ll 3.li               \"column width
.nr M \\n(.o           \"save left margin
```

Typically, a portion of the top of the first page contains full-width text; the request for the narrower line length, as well as another .mk is made where the two-column output begins.

21.5 Footnote Processing

The footnote mechanism is used by embedding the footnotes in the input text at the point of reference, demarcated by an initial .fn and a terminal .ef:

```
.fn
Footnote text and control lines...
.ef
```

In the following, footnotes are processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote does not completely fit in the available space.

```
.de hd                   \"header
. ---
.nr x 0 1                \"init footnote count
.nr y 0-\\nb             \"current footer place
.ch fo -\\nbu            \"reset footer trap
.if \\n(dn .fz           \"leftover footnote
..
.de fo                   \"footer
.nr dn 0                 \"zero last diversion size
.if \\nx \{\
.ev 1                    \"expand footnotes in evl
.nf                      \"retain vertical size
.FN                      \"footnotes
.rm FN                   \"delete it
.if "\\n(.z"fy" .di      \"end overflow diversion
.nr x 0                  \"disable fx
.ev  \}                  \"pop environment
. ---
'bp
..
.de fx                   \"process footnote overflow
.if \\nx .di fy          \"divert overflow
```

```
..
.de fn                  \"start footnote
.da FN                  \"divert (append) footnote
.ev 1                   \"in environment 1
.if \\n+x=1 .fs         \"if first, include separator
.fi                     \"fill mode
..
.de ef                  \"end footnote
.br                     \"finish output
.nr z                   \\n(.v\"save spacing
.ev                     \"pop ev
.di                     \"end diversion
.nr y -\\n(dn           \"new footer position,
.if                     \\nx=1 .nr y -(\\n(.v-\\nz) \
                        \"uncertainty correction
.ch fo \\nyu            \"y is negative
.if ( \\n(nl+1v)>(\\n(.p+\\ny) \
.ch fo \\n(nlu+1v       \"it didn't fit
..
.de fs                  \"separator
\l'1i'                  \"1 inch rule
.br
..
.de fz                  \"get leftover footnote
.fn
.nf                     \"retain vertical size
.fy                     \"where fx put it
.ef
..
.nr b 1.0i              \"bottom margin size
.wh 0 hd                \"header trap
.wh 12i fo              \"footer trap, temp position
.wh -\\nbu fx           \"fx at footer position
.ch fo -\\nbu           \"conceal fx with fo
```

The header hd initializes a footnote count register x, and sets both the current footer trap position register y and the footer trap itself to a nominal position specified in register b. In addition, if the register dn indicates a leftover footnote, fz is invoked to reprocess it. The footnote start macro fn begins a diversion (append) in environment 1, and increments the count x; if the count is one, the footnote separator fs is interpolated. The separator is kept in a separate macro to permit user redefinition. The footnote end macro ef restores the previous environment and ends the diversion after saving the spacing size in register z. y is then decremented by the size of the footnote, available in dn; then on the first footnote, y is further decremented by the difference in vertical base-line spacings of the two environments to prevent the late triggering of the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the

lower (on the page) of y or the current page position (nl) plus one line, to allow for printing the reference line. If indicated by x, the footer fo rereads the footnotes from FN in nofill mode in environment 1, and deletes FN. If the footnotes are too large to fit, the macro fx is trap-invoked to redivert the overflow into fy, and the register dn later indicates to the header whether fy is empty. Both fo and fx are planted in the nominal footer trap position in an order that causes fx to be concealed unless the fo trap is moved. The footer then terminates the overflow diversion, if necessary, and zeros x to disable fx, because the uncertainty correction together with a not-too-late triggering of the footer can result in the footnote rereading and finishing before reaching the fx trap.

A good exercise is to combine the multiple-column and footnote mechanisms.

21.6 Last Page

After the last input file has ended, nroff and troff invoke the end macro, if any, and when it finishes, eject the remainder of the page. During the eject, any traps encountered are processed normally. At the end of this last page, processing terminates unless a partial line, word, or partial word remains. To start another page, use the end-macro

```
.de en                  \"end-macro
\c
'bp
..
.em en
```

to deposit a null partial word, and effect another last page.

APPENDIX A
SUMMARY AND INDEX

A.1 Summary

* Values separated by ; are for nroff and troff, respectively.

Notes are explained at the end of this Summary and Index.

† No effect in nroff.

‡ The use of ` as control character (instead of .) suppressess the break functon.

1. General Explanation

2. Font and Character Size Control

Request Form	Initial Value	If No Argument	Notes	Explanation
.ps ±N	10 point	previous	E	Point size; also \s±N.
.ss N	12/36 em	ignored	E	Space-character size set to N/36 em.†
.cs F N M	off	-	P	Constant character space (width) mode (font F).†
.bd F N	off	-	P	Embolden font F by N-1 units.†
.bd S F N	off	-	P	Embolden Special Font when current font is F.†
.ft F	Roman	previous	E	Change to font F=x, xx, or 1-4. Also \fx,\f(xx, \fN.
.fp N F	R,I,B,S	ignored	-	Font named F mounted on physical position 1≤N≤4.

3. Page Control

Request Form	Initial Value	If No Argument	Notes	Explanation
.pl ±N	11in	11in	v	Page length.
.bp ±N	N=1	-	B,v	Eject current page; next page number N.
.pn ±N	N=1	ignored	-	Next page number N.
.po ±N	0;26/27in	previous	v	Page offset.
.ne N	-	N=1V	D,v	Need N vertical space (V=vertical spacing).
.mk	none	internal	D	Mark current vertical place in register R.
.rt ±N	none	internal	D,v	Return (upward only) to marked vertical place.

4. Text Filling, Adjusting, and Centering

Request Form	Initial Value	If No Argument	Notes	Explanation
.br	-	-	B	Break.
.fi	fill	-	B,E	Fill output lines.
.nf	no fill	-	B,E	No filling or adjusting of output lines.
.ad c	adj,both	adjust	E	Adjust output lines with mode c.
.na	adjust	-	E	No output line adjusting.
.ce N	off	N=1	B,E	Center following N input text lines.

5. Spacing

Request Form	Initial Value	If No Argument	Notes	Explanation
.vs N	1/6in; 12pts	previous	E,p	Vertical base line spacing (V).
.ls N	N=1	previous	E	Output N-1 Vs after each text output

				line.
.sp N	-	N=1V	B,v	Space vertical distance N in either direction.
.sv N	-	N=1V	B,v	Save vertical distance N.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.
.ll ±N	6.5in	previous	E,m	Line length.
.in ±N	N=0	previous	B,E,m	Indent.
.ti ±N	-	ignored	B,E,m	Temporary indent.

6. Macros, Strings, Diversion, and Position Traps

Request Form	Initial Value	If No Argument	Notes	Explanation
.de xx yy	-	.yy=..	-	Define or redefine macro xx; end at call of yy.
.am xx yy	-	.yy=..	-	Append to a macro.
.ds xx string	-	ignored	-	Define a string xx containing string.
.as xx string	-	ignored	-	Append string to string xx.
.rm xx	-	ignored	-	Remove request, macro, or string.
.rn xx yy	-	ignored	-	Rename request, macro, or string xx to yy.
.di xx	-	end	D	Divert output to macro xx.
.da xx	-	end	D	Divert and append to xx.
.wh N xx	-	-	v	Set location trap; negative is with respect to page bottom.
.ch xx N	-	-	v	Change trap location.
.dt N xx	-	off	D,v	Set a diversion trap.
.it N xx	-	off	E	Set an input-line count trap.
.em xx	none	none	-	End macro is xx.

7. Number Registers

Request Form	Initial Value	If No Argument	Notes	Explanation
.nr R	±N M	-	u	Define and set number register R; auto-increment by M.
.af R c	arabic	-	-	Assign format to register R (c=1, i, I, a, A).
.rr R	-	-	-	Remove register R.

8. Tabs, Leaders, and Fields

Request Form	Initial Value	If No Argument	Notes	Explanation
.ta Nt ...	0.8;0.5in	none	E,m	Tab settings; left type, unless t=R(right), C(centered).
.tc c	none	none	E	Tab repetition character.
.lc c	.	none	E	Leader repetition character.
.fc a b	off	off	-	Set field delimiter a and pad character b.

9. Input and Output Conventions and Character Translations

Request Form	Initial Value	If No Argument	Notes	Explanation
.ec c	\	\	-	Set escape character.
.eo	on	-	-	Turn off escape character mechanism.
.lg N	-; on	on	-	Ligature mode on if N>0.
.ul N	off	N=1	E	Underline (italicize in troff) N input lines.
.cu N	off	N=1	E	Continuous underline in nroff; like ul in troff.

.uf F	Italic	Italic	-	Underline font set to F (to be switched to by ul).
.cc c	.	.	E	Set control character to c.
.c2 c	'	'	E	Set nobreak control character to c.
.tr abcd....	none	-	O	Translate a to b, etc. on output.

10. Local Horizontal and Vertical Motions, and the Width Function

11. Overstrike, Line-drawing, and Zero-width Functions

12. Hyphenation

Request Form	Initial Value	If No Argument	Notes	Explanation
.nh	hyphenate	-	E	No hyphenation.
.hy N	hyphenate	hyphenate	E	Hyphenate; N = mode.
.hc c	\%	\%	E	Hyphenation indicator character c.
.hw word1...		ignored	-	Exception words.

13. Three Part Titles.

Request Form	Initial Value	If No Argument	Notes	Explanation
.tl 'left'center'right'		-	-	Three-part title.
.pc c	%	off	-	Page number character.
.lt ±N	6.5in	previous	E,m	Length of title.

14. Output Line Numbering.

Request Form	Initial Value	If No Argument	Notes	Explanation
.nm ±N M S I		off	E	Number mode on or off, set parameters.
.nn N	-	N=1	E	Do not number next N lines.

15. Conditional Acceptance of Input

Request Form	Initial Value	If No Argument	Notes	Explanation
.if c anything		-	-	If condition c true, accept anything as input, for multi-line use \{anything\}.
.if !c anything		-	-	If condition c false, accept anything.
.if N anything		-	u	If expression N>0, accept anything.
.if !N anything		-	u	If expression N ≤0, accept anything.
.if 'string1'string2' anything			-	If string1 identical to string2, accept anything.
.if ! 'string1 'string2 'anything			-	If string1 not identical to string2, accept anything.
.ie c anything		-	u	If portion of if-else; all above forms (like if).
.el anything		-	-	Else portion of if-else.

16. Environment Switching

Request Form	Initial Value	If No Argument	Notes	Explanation
.ev N	N=0	previous	-	Environment switched (pushed down).

17. Insertions from the Standard Input

Request Form	Initial Value	If No Argument	Notes	Explanation
.rd prompt	-	prompt=BEL	-	Read insertion.
.ex	-	-	-	Exit from nroff/troff.

18. Input/Output File Switching

Request Form	Initial Value	If No Argument	Notes	Explanation
.so filename		-	-	Switch source file (push down).
.nx filename		EOF	-	Next file.
.pi program		-	-	Pipe output to program (nroff only).

19. Miscellaneous

Request Form	Initial Value	If No Argument	Notes	Explanation
.mc c N	-	off	E,m	Set margin character c and separation N.
.tm string	-	newline	-	Print string on terminal (ZEUS stand ard message output).
.ig yy	-	.yy=..	-	Ignore till call of yy.
.pm t	-	all	-	Print macro names and sizes; if t present, print only total of sizes.
.fl	-	-	B	Flush output buffer.

20. Output and Error Messages

NOTES

B Request normally causes a break.

D Mode or relevant parameters associated with current diversion level.

E Relevant parameters are a part of the current environment.

O Must stay in effect until logical output.

P — Mode must be still or again in effect at the time of physical output.

v,p,m,u — Default scale indicator; if not specified, scale indicators are ignored.

A.2 Alphabetical Request and Section Number Cross Reference

ad	4	dt	6	ig	19	nn	14	rs	5
af	7	ec	9	in	5	nr	7	rt	3
am	6	ei	15	it	6	ns	5	so	18
as	6	em	6	lc	8	nx	18	sp	5
bd	2	eo	9	lg	9	os	5	ss	2
bp	3	ev	16	li	9	pc	13	sv	5
br	4	ex	17	ll	5	pi	18	ta	8
c2	9	fc	8	ls	5	pl	3	tc	8
cc	9	fi	4	lt	13	pm	19	ti	5
ce	4	fl	19	mc	19	pn	3	tl	13
ch	6	fp	2	mk	3	po	3	tm	19
cs	2	ft	2	na	4	ps	2	tr	9
cu	9	hc	12	ne	3	rd	17	uf	9
da	6	hw	12	nf	4	rm	6	ul	9
de	6	hy	12	nh	12	rn	6	vs	5
di	6	ie	15	nm	14	rr	7	wh	6
ds	6	if	15						

A.3 Escape Sequences for Characters, Indicators, and Functions

Section Reference	Escape Sequence	Meaning
9.1	\\	\ (to prevent or delay the interpretation of \)
9.1	\e	Printable version of the current escape character.
2.1	\'	\' (acute accent); equivalent to \(aa
2.1	\`	` (grave accent); equivalent to \(ga
2.1	\-	- Minus sign in the current font
6	\.	Period (dot) (see de)
10.1	\(space)	Unpaddable space-size space character
10.1	\0	Digit-width space
10.1	|	1/6 em narrow space character (zero-width in nroff)
10.1	\^	1/12 em half-narrow space character (zero width in nroff)

4.1	\&	Nonprinting, zero width character
9.6	\!	Transparent line indicator
9.7	\"	Beginning of comment
6.3	\$N	Interpolate argument $1 \leq N \leq 9$
12	\%	Default optional hyphenation character
2.1	\(xx	Character named xx
6.1	*x,\(xx	Interpolate string x or xx
8.1	\a	Noninterpreted leader character
11.2	\b'abc...'	Bracket building function
4.2	\c	Interrupt text processing
10.1	\d	Forward (down) 1/2 em vertical motion (1/2 line in nroff)
2.2	\fx,\f(xx,\fN	Change to font named x or xx or position N
10.1	\h'N'	Local horizontal motion; move right N (negative left)
10.3	\kx	Mark horizontal input place in register x
11.3	\l'Nc'	Horizontal line drawing function (optionally with c)
11.3	\L'Nc'	Vertical line drawing function (optionally with c)
8	\nx,\n(xx	Interpolate number register x or xx
11.1	\o'abc...'	Overstrike characters a, b, c, ...
4.1	\p	Break and spread output line
10.1	\r	Reverse 1 em vertical motion (reverse line in nroff)
2.3	\sN,\s±N	Point-size change function
8.1	\t	Noninterpreted horizontal tab
10.1	\u	Reverse (up) 1/2 em vertical motion (1/2 line in nroff)
10.1	\v'N'	Local vertical motion; move down N (negative up)
10.2	\w'string'	Interpolate width of string
5.2	\x'N'	Extra line-space function (negative before, positive after)
11.4	\zc	Print c with zero width (without spacing)
15	\{	Begin conditional input
15	\}	End conditional input
9.7	\(newline)	Concealed (ignored) new line
-	\X	X, any character not listed above

The escape sequences \\, \., \", \$, *, \a, \n, \t, and (new line) are interpreted in copy mode (Section 7.2).

A.4 Predefined General Number Registers

Section Reference	Register Name	Description
3	%	Current page number
10.2	ct	Character type (set by width function)
6.4	dl	Width (maximum) of last completed diversion
6.4	dn	Height (vertical size) of last completed diversion
-	dw	Current day of the week (1-7)
-	dy	Current day of the month (1-31)
10.3	hp	Current horizontal place on input line
14	ln	Output line number
-	mo	Current month (1-12)
4.1	nl	Vertical position of last printed text base-line
10.2	sb	Depth of string below base line (generated by width function)
10.2	st	Height of string above base line (generated by width function)
-	yr	Last two digits of current year

A.5 Predefined Read-Only Number Registers

Section Reference	Register Name	Description
6.3	$	Number of arguments available at the current macro level
-	A	Set to 1 in troff if -a option used; always 1 in nroff
10.1	H	Available horizontal resolution in basic units
-	T	Set to 1 in nroff, if -T option used; always 0 in troff
10.1	V	Available vertical resolution in basic units
5.2	a	Post-line extra line-space most recently utilized using ex "N"
-	c	Number of lines read from current input file
6.4	d	Current vertical place in current diversion; equal to nl, if no diversion
2.2	f	Current font as physical quadrant (1-4)
4	h	Text base-line mark on current page or diversion
5	i	Current indent
5	l	Current line length
4	n	Length of text portion on previous output

		line
3	o	Current page offset
3	p	Current page length
2.3	s	Current point size
6.5	t	Distance to the next trap
4.1	u	Equal to 1 in fill mode and 0 in nofill mode
5.1	v	Current vertical line spacing
10.2	w	Width of previous character
-	x	Reserved version-dependent register
-	y	Reserved version-dependent register
6.4	z	Name of current diversion

APPENDIX B

SUMMARY OF RECENT CHANGES TO NROFF/TROFF

Options

-h (Nroff only) Output tabs used during horizontal spacing to speed output as well as reduce output byte count. Device tab settings are assumed to be every eight nominal character widths. The default settings of input (logical) tabs is also initialized to every eight nominal character widths.

-z Efficiently suppresses formatted output. Only message output occurs (from "tm"s and diagnostics).

Old Requests

.ad c The adjustment type indicator "c" is now also a number previously obtained from the ".j" register.

.so name The contents of file "name" are interpolated at the point the "so" is encountered. Previously, the interpolation was done upon return to the file-reading input level.

New Request

.ab text Prints "text" on the message output and terminates without further processing. If "text" is missing, "User Abort." is printed. Does not cause a break. The output buffer is flushed.

.fz F N Forces font "F" to be in size N. N can have the form N, +N, or -N. For example,

```
.fz 3 -2
```

causes an implicit \s-2 every time font 3 is entered, and a corresponding \s+2 when it is left. Special font characters occurring during the use of font F have the same size modification. If special characters are treated differently,

```
.fz S F N
```

is used to specify the size treatment of special characters during font F. For example,

```
.fz 3 -3
.fz S 3 -0
```

causes automatic reduction of font 3 by 3 points while the special characters is not affected. Any ".fp" request specifying a font on some position must precede ".fz" requests relating to that position.

New Predefined Number Registers

.k Read-only. Contains the horizontal size of the text portion (without indent) of the current partially collected output line, if any, in the current environment.

.j Read-only. A number representing the current adjustment mode and type. Can be saved and later given to the "ad" request to restore a previous mode.

.P Read-only. 1 if the current page is being printed, and zero otherwise.

.L Read-only. Contains the current line-spacing parameter ("ls").

.c General register access to the input line-number in the current input file. Contains the same value as the read-only ".c" register.

ZEUS PROGRAMMING*

* This information is based on an article originally written by Brian W. Kernighan, Bell Laboratories.

PREFACE

This document introduces programming using ZEUS. The emphasis is on how to write programs that interface with the operating system, either directly or through the standard I/O library. The topics discussed include:

- Handling command arguments
- Standard I/O
- Standard I/O file access
- Low-level I/O
- Processes
- Signals

The material discussed in this document is also covered in the _ZEUS Reference Manual_ and in _ZEUS for Beginners_. All programming is done in C; refer to _The C Programming Language_ by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978) for more information on C.

TABLE OF CONTENTS

SECTION 1 BASICS 5

1.1 Program Arguments 5
1.2 The Standard Input and Output 5

SECTION 2 THE STANDARD I/O LIBRARY 8

2.1 Introduction 8
2.2 File Access 8
2.3 Error Handling 11
2.4 Miscellaneous I/O Functions 12
2.5 General Usage 12
2.6 Calls 13
2.7 Macros 18

SECTION 3 LOW-LEVEL I/O 20

3.1 General 20
3.2 File Descriptors 20
3.3 Read and Write 21
3.4 Open, Creat, Close, Unlink 23
3.5 Random Access with lseek 25
3.6 Error Processing 25

SECTION 4 PROCESSES 27

4.1 System Function 27
4.2 Low-Level Process Creation 27
4.3 Control of Processes 28
4.4 Pipes 30

SECTION 5 SIGNALS 34

5.1 General 34
5.2 Signal Routine 34
5.3 Interrupts 35

SECTION 1

BASICS

1.1 Program Arguments

When a C program is run as a command, the arguments on the command line are available to the function _main_ as an argument count (_argc_) and an array (_argv_) of pointers to character strings that contain the arguments. By convention, _argv_[_0_] is the command name itself, so _argc_ is always greater than 0. The following program illustrates the method used. It simply echoes its arguments back to the terminal.

```
main(argc, argv)    /* echo arguments */
int argc;
char *argv[];
{
      int i;

      for (i = 1; i < argc; i++)
       printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

The array _argv_ is a pointer to an array whose individual elements are pointers to arrays of characters. Each array of characters is terminated by \0, so it can be treated as a string. The program starts by printing _argv_[_1_] and loops until it has printed all of the arrays.

The argument count and the arguments are parameters to _main_. To save them so that other routines can use them, they must be copied to external variables.

1.2 The Standard Input and Output

The simplest input mechanism is to read the standard input, which is data from the user's terminal. The function _getchar_ returns the next input character each time it is called. Input from a file can be substituted for input from the terminal by using the < convention as defined in _ZEUS for Beginners_. If _prog_ uses _getchar_, then the command line

```
prog < file
```

causes _prog_ to read _file_ instead of the terminal; _prog_ itself is not affected by the origin of its input. This is

also true if the input comes from another program using a *pipe*.

```
otherprog | prog
```

provides the standard input for *prog* from the standard output of *otherprog*.

The function *getchar* returns *EOF* when it encounters the end of file or an error on what is being read.

The function *putchar*(*c*) puts the character *c* on the standard output. The output can be captured on a file by using >. If *prog* uses *putchar*,

```
prog > outfile
```

writes the standard output on *outfile* instead of on the terminal. If *outfile* does not exist, it is created. If it already exists, its previous contents are overwritten. A pipe can be used.

```
prog | otherprog
```

puts the standard output of *prog* into the standard input of *otherprog*.

The function *printf*, which formats output in various ways, uses the same mechanism as *putchar*. Therefore, calls to *printf* and *putchar* can be intermixed in any order. The output appears in the order of the calls.

Similarly, the function *scanf* provides formatted input conversion; it reads the standard input and breaks it into strings, numbers, and so on, as desired. The function *scanf* uses the same mechanism as *getchar*, so calls to either can be intermixed.

Many programs read only one input and write only one output. For such programs, I/O with *getchar*, *putchar*, *scanf*, and *printf* can be adequate, and it is enough to get started. This is particularly true if the ZEUS pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ASCII control characters from its input (except for new line and tab).

```
#include <stdio.h>

main()    /* ccstrip: strip nongraphic characters */
{
   int c;
   while ((c = getchar()) != EOF)
    if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
      putchar(c);
   exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (/*usr*/*include*/*stdio*.*h*) of standard routines and symbols that includes the definition of *EOF*.

If it is necessary to treat multiple files, *cat* can be used to collect the files:

```
cat file1 file2 ... | ccstrip > output
```

thereby avoiding the necessity of learning how to access files from a program. The *exit* at the end of the program is not necessary, but it ensures that any caller of the program sees a normal termination status (conventionally 0) from the program when it completes. (Section 6 discusses status returns in more detail.)

SECTION 2

THE STANDARD I/O LIBRARY

2.1 Introduction

The standard I/O library is a collection of routines providing efficient and portable I/O services for most C programs. The standard I/O library is available on System 8000, which supports C. Programs that confine their system interactions to the library's facilities can be easily transported from System 8000 to another system or from another system to System 8000.

The standard I/O library was designed with the following goals in mind.

1. Maximal time and space efficiency so that it can be used in all applications no matter how critical.

2. Simple to use and free from unexplained numbers and calls that interfere with the understandability and portability of many programs using older packages.

3. The interface provided is applicable on all machines, whether or not the programs that implement it are directly portable to other systems.

In Sections 2.2 through 2.4, the basics of the standard I/O library are discussed. Sections 2.5, 2.6, and 2.7 contain a more complete description of its capabilities.

2.2 File Access

The programs described so far read the standard input and write the standard output. Programs can also access a file not already connected to the program. One example, _wc_, counts the number of lines, words, and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words, and characters in the file _x.c_ and the file _y.c_ and then prints the combined total lines, words, and characters for these files.

It is necessary to connect the file system names to the I/O statements that read the data. Before a file is read or written, it is opened by the standard library function

fopen, which takes an external name (like *x.c* or *y.c*), interfaces with the operating system, and returns an internal name that must be used in subsequent reads or writes of the file.

This internal name is a pointer (called a *file pointer*) to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, and whether the file is being read or written. Part of the standard I/O definitions obtained by including *stdio.h* is a structure definition called *FILE*. The only declaration needed for a file pointer is one such as:

```
FILE *fp, *fopen();
```

Here, *fp* is a pointer to a *FILE*, and *fopen* returns a pointer to a *FILE*. (*FILE* is a type name, like integer (*int*), not a structure tag.)

The actual call to *fopen* in a program is:

```
fp = fopen(name, mode);
```

The first argument of *fopen* is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how the file is to be used. The only allowable modes are read ("r"), write ("w"), and append ("a").

If a file opened for writing does not exist, it is created, if possible. Opening an existing file for writing destroys the old contents. Trying to read a file that does not exist is an error. There can be other causes of error as well, such as trying to read a file without having read permission. If there is any error, *fopen* returns the null pointer value *NULL* (defined as zero in *stdio.h*).

There are several ways to read or write the file once it is open. The simplest are *getc* and *putc*. The function *getc* returns the next character from a file--it needs the file pointer to tell it what file to read. For example,

```
c = getc(fp)
```

places the next character from the file referred to by *fp* in *c*. *EOF* is returned when end of file is reached. The inverse of *getc* is *putc*.

```
putc(c, fp)
```

puts the character *c* on the file *fp* and returns *c*. *EOF* is returned on error.

When a program is started, three files--predefined in the I/O library as the standard input (*stdin*), the standard output (*stdout*), and the standard error output (*stderr*) files--are opened automatically, and file pointers are provided for them. Normally, these file pointers are all connected to the terminal, but they can be redirected to files or pipes as described in Section 1.2. The files *stdin*, *stdout*, and *stderr* can be used wherever an object of type *FILE* can be used. However, they are constants, not variables, so nothing can be assigned to them.

With some of the preliminaries out of the way, *wc* can now be written. The basic design of *wc* is convenient for many programs. If there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. Thus, the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)     /* wc: count lines, words, chars */
int argc;
char *argv[];
{
   int c, i, inword;
   FILE *fp, *fopen();
   long linect, wordct, charct;
   long tlinect = 0, twordct = 0, tcharct = 0;

   i = 1;
   fp = stdin;
   do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
             fprintf(stderr, "wc: can't open %s\n", argv[i]);
             continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
             charct++;
             if (c == '\n')
                   linect++;
             if (c == ' ' || c == '' || c == '\n')
                   inword = 0;
             else if (inword == 0) {
                   inword = 1;
                   wordct++;
             }
         }
         printf("%7ld %7ld %7ld", linect, wordct, charct);
```

```
            printf(argc > 1 ? " %s0\n" : "0\n", argv[i]);

            fclose(fp);
            tlinect += linect;
            twordct += wordct;
            tcharct += charct;
        } while (++i < argc);
        if (argc > 2)
        printf("%7ld %7ld %7ld total0\n", tlinect, twordct, tcharct);
        exit(0);
}
```

The function _fprintf_ is identical to _printf_, except that the first argument in _fprintf_ is a file pointer that specifies the file to be written.

The function _fclose_ is the inverse of _fopen_. It breaks the connection between the file pointer and the external name that is established by _fopen_, freeing the file pointer for another file. Since there is a limit on the number of files that a program can have open simultaneously, files should be freed when they are no longer needed. The function _fclose_ also flushes the buffer in which _putc_ is collecting output (_fclose_ is called automatically for each open file when a program terminates normally).

2.3 Error Handling

The file _stderr_ is assigned to a program in the same way as _stdin_ and _stdout_. Output written on _stderr_ appears on the terminal, even if the standard output is redirected. The command _wc_ writes its diagnostics on _stderr_ instead of _stdout_, so that if one of the files cannot be accessed, the message goes to the terminal instead of disappearing down a pipeline or into an output file.

The program signals errors by using the function _exit_ to terminate program execution. The argument of _exit_ is available to the process that called it (see Section 5), so the success or failure of the program can be tested by another program that uses it as a subprocess. By convention, a return value of 0 signals that all is well; nonzero values signal abnormal situations.

The _exit_ command calls _fclose_ for each open output file to flush out any buffered output. It then calls the routine __exit_, which causes immediate termination without any buffer flushing. The __exit_ routine can be called directly if desired.

2.4 Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those illustrated above.

Normally, output with *putc*, *getc*, etc., is buffered (except to *stderr*). To force it out immediately, use *fflush*(*fp*).

The function *fscanf* is identical to *scanf*, except that its first argument is a file pointer that specifies the file from which the input comes. It returns *EOF* at end of file.

The functions *sscanf* and *sprintf* are identical to *fscanf* and *fprintf*, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for *sscanf* and to the string for *sprintf*.

The function *fgets*(*buf*, *size*, *fp*) copies the next line from *fp* (up to and including a new line) into *buf*. At most, *size*-*1* characters are copied. *NULL* is returned at end of file. The function *fputs*(*buf*, *fp*) writes the string in *buf* onto file *fp*.

The function *ungetc*(*c*, *fp*) "pushes back" the character *c* onto the input stream *fp*. A subsequent call to *getc*, *fscanf*, etc., encounters *c*. Only one character of pushback per file is permitted.

2.5 General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

to define certain macros and variables. These routines are in the normal C library. All names in the include file intended only for internal use begin with an underscore (_) to reduce the possibility of confusion by these files having the same name as user named files. The following names are to be visible outside the package.

stdin	Standard input file
stdout	Standard output file
stderr	Standard error file
EOF	Defined to be -1, the value returned by the read routines on end-of-file or error

NULL — Notation for the null pointer returned by pointer-valued functions to indicate an error

FILE — Expands to *struct _iob*; useful shorthand when declaring pointers to streams

BUFSIZ — size number suitable for an I/O buffer (see *setbuf* in Section 2.6)

getc, getchar, putc, putchar, feof, ferror, fileno
Macros, whose actions are described below. They are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions. Therefore, they cannot have breakpoints set on them.

The routines discussed here offer automatic buffer allocation and output flushing where appropriate. The names *stdin*, *stdout*, and *stderr* are constants and nothing can be assigned to them.

2.6 Calls

FILE **fopen*(*filename*, *type*) *char* **filename*, **type*;

This call opens the file and, if needed, allocates a buffer for it. The character string *filename* specifies the name. The argument *type* is a character string, not a single character. It can be "*r*", "*w*", or "*a*" to indicate read, write, or append. The value returned is a file pointer. If it is *NULL*, the attempt to open failed.

FILE **freopen*(*filename*, *type*, *ioptr*) *char* **filename*, **type*;
FILE **ioptr*;

The stream named by *ioptr* is closed, if necessary, and then reopened as if by *fopen*. If the attempt to open fails, *NULL* is returned; otherwise, *ioptr* is returned (*ioptr* now refers to the new file). The reopened stream is often *stdin* or *stdout*.

int *getc*(*ioptr*) *FILE* **ioptr*;

This call returns the next character from the stream named by *ioptr*, a pointer to a file (similar to one returned by *fopen*), or the name *stdin*. The integer *EOF* is returned on end-of-file or when an error occurs. The null character \0 is a legal character.

int _fgetc_(_ioptr_) _FILE_ *_ioptr_;

This call acts like _getc_, but it is a genuine function, not a macro, so it can be pointed to or passed as an argument.

putc(_c_, _ioptr_) _char_ _c_; _FILE_ *_ioptr_;

The _putc_ call writes the character _c_ on the output stream named by _ioptr_, which is a value returned from _fopen_, _stdout_, or _stderr_. The character _c_ is passed as value; _EOF_ is returned on error.

fputc(_c_, _ioptr_) _char_ _c_; _FILE_ *_ioptr_;

This call acts like _putc_, but it is a function, not a macro.

fclose(_ioptr_) _FILE_ *_ioptr_;

The file corresponding to _ioptr_ is closed after any buffers are emptied, and a buffer allocated by the I/O system is freed. The _fclose_ function is automatic on normal termination of the program.

fflush(_ioptr_) _FILE_ *_ioptr_;

Any buffered information on the output stream named by _ioptr_ is written out. Output files are normally buffered only if they are not directed to the terminal. However, _stderr_ always starts unbuffered and remains so, unless _setbuf_ is used or unless it is reopened.

exit(_errcode_);

This call terminates the process and returns its argument as status to the parent. This is a special version of the routine that calls _fflush_ for each output file. The call __exit_ terminates without flushing.

feof(_ioptr_) _FILE_ *_ioptr_;

This call returns nonzero when _EOF_ has occurred on the specified input stream.

ferror(_ioptr_) _FILE_ *_ioptr_;

This call returns nonzero when an error has occurred while the named stream is being read or written. The error indication lasts until the file has been closed.

getchar();

This call is identical to _getc_(_stdin_).

putchar(_c_) _char_ _c_;

This call is identical to _putc_(_c_, _stdout_).

char *_fgets_(_s_, _n_, _ioptr_) _char_ *_s_; _int_ _n_; _FILE_ *_ioptr_;

This call reads into the character pointer _s_, up to n-1 characters from the stream _ioptr_. The read terminates with a new line character, which is placed in the buffer followed by a null character. The function _fgets_ returns the first argument or _NULL_ if error or _EOF_ occurred.

fputs(_s_, _ioptr_) _char_ *_s_; _FILE_ *_ioptr_;

This call writes the null-terminated string (character array) _s_ on the stream _ioptr_. A new line is not appended, and no value is returned.

ungetc(_c_, _ioptr_) _char_ _c_; _FILE_ *_ioptr_;

The argument character _c_ is pushed back on the input stream named by _ioptr_. Only one character at a time can be pushed back.

printf(_format_, _a1_, ...) _char_ *_format_;
fprintf(_ioptr_, _format_, _a1_, ...) _FILE_ *_ioptr_; _char_ *_format_;
sprintf(_s_, _format_, _a1_, ...)_char_ *_s_, *_format_;

The function _printf_ writes on the standard output. The function _fprintf_ writes on the named output stream, and _sprintf_ puts characters in the character array named by _s_. The specifications are as described in _printf_(_3_) of the _ZEUS Reference Manual_.

scanf(_format_, _a1_, ...) _char_ *_format_;
fscanf(_ioptr_, _format_, _a1_, ...) _FILE_ *_ioptr_; _char_ *_format_;
sscanf(_s_, _format_, _a1_, ...) _char_ *_s_, *_format_;

The _scanf_ function reads from the standard input; _fscanf_ reads from the named input stream; _sscanf_ reads from the character string supplied as _s_; and _scanf_ reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects, as arguments, a control string _format_ and a set of arguments, each of which must be a pointer that indicates where the converted input is to be stored. The function _scanf_ returns the number of

successfully matched and assigned input items as its value. This can be used to decide how many input items were found. _EOF_ is returned on end of file. Note that this is different from 0, which means that the next input character does not match what was called for in the control string.

fread(_ptr_, _sizeof_(*_ptr_), _nitems_, _ioptr_) _char_ *_ptr_; _int_ _nitems_; _FILE_ *_ioptr_;

This call reads _nitems_ of data from file _ioptr_, beginning at _ptr_. Advance notification of binary I/O is not required. When, for portability reasons, binary I/O becomes required, an additional character is added to the mode-string on the _fopen_ call.

fwrite(_ptr_, _sizeof_(*_ptr_), _nitems_, _ioptr_) _char_ *_ptr_; _int_ _nitems_; _FILE_ *_ioptr_;

This call is similar to _fread_, except that it writes _nitems_ of data from file _ioptr_, beginning at _ptr_.

rewind(_ioptr_) _FILE_ *_ioptr_;

This call rewinds the stream named by _ioptr_. It is not very useful except for input, since a rewound output file is open only for output.

system(_string_) _char_ *_string_;

The _string_ is executed by the shell as if it were typed at the terminal.

getw(_ioptr_) _FILE_ *_ioptr_;

This call returns the next word from the input stream named by _ioptr_. _EOF_ is returned on end of file or error, but since this is a good integer, _feof_ and _ferror_ should be used. (System 8000 uses 16-bit words.)

putw(_w_, _ioptr_) _int_ _w_; _FILE_ *_ioptr_;

This call writes the integer _w_ on the named output stream.

setbuf(_ioptr_, _buf_) _FILE_ *_ioptr_; _char_ *_buf_;

The function _setbuf_ can be used after a stream has been opened, but before I/O has started. If _buf_ is _NULL_, the stream is unbuffered. Otherwise, the buffer supplied, which must be a character array of sufficient size, is used:

```
char buf[BUFSIZ];
```

fileno(_ioptr_) _FILE_ *_ioptr_;

This call returns the integer file descriptor associated with the file.

fseek(_ioptr_, _offset_, _ptrname_) _FILE_ *_ioptr_; _long_ _offset_; _int_ _ptrname_;

The location of the next byte in the stream named by _ioptr_ is adjusted. The argument _offset_ is a long integer. If _ptrname_ is 0, the offset is measured from the beginning of the file. If _ptrname_ is 1, the offset is measured from the current read or write pointer. If _ptrname_ is 2, the offset is measured from the end of the file. This routine accounts for any buffering. When this routine is used on non-ZEUS systems, the offset must be a value returned from _ftell_ and the _ptrname_ must be 0.

long _ftell_(_ioptr_) _FILE_ *_ioptr_;

The byte offset (measured from the beginning of the file) associated with the named stream is returned. Any buffering is accounted for. On non-ZEUS systems, the value of this call is useful only for handing to _fseek_, to position the file to the same place it was when _ftell_ was called.

getpw(_uid_, _buf_) _int_ _uid_; _char_ *_buf_;

The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array _buf_, and 0 is returned. If no line is found corresponding to the user ID, 1 is returned.

char *_malloc_(_num_); _int_ _num_;

This call allocates _num_ bytes. Because the pointer returned is sufficiently well aligned, it can be used for any purpose. _NULL_ is returned if no space is available.

char *_calloc_(_num_, _size_); _int_ _num_, _size_;

This call allocates space for _num_ items, each of size _size_. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. _NULL_ is returned if no space is available.

cfree(_ptr_) _char_ *_ptr_;

Space is returned to the operating system used by _calloc_. If the pointer was not obtained from _calloc_, this will not function properly.

2.7 Macros

The definitions of the following macros can be obtained by including <_ctype_._h_>.

isalpha(_c_)

returns nonzero if the argument is alphabetic

isupper(_c_)

returns nonzero if the argument is upper-case alphabetic

islower(_c_)

returns nonzero if the argument is lower-case alphabetic

isdigit(_c_)

returns nonzero if the argument is a digit

isspace(_c_)

returns nonzero if the argument is a spacing character (tab, new line, carriage return, vertical tab, form feed, or space)

ispunct(_c_)

returns nonzero if the argument is any punctuation character (not a space, letter, digit, or control character)

isalnum(_c_)

returns nonzero if the argument is alphanumeric

isprint(_c_)

returns nonzero if the argument is printable (a letter, digit, or punctuation character)

iscntrl(_c_)

returns nonzero if the argument is a control character

isascii(_c_)

returns nonzero if the argument is an ASCII character

toupper(_c_)

returns the upper-case character corresponding to the lower-case letter _c_

tolower(_c_)

returns the lower-case character corresponding to the upper-case letter _c_

SECTION 3

LOW-LEVEL I/O

3.1 General

The bottom level of I/O on ZEUS is described in this section, and it does not provide buffering or any other services. It is a direct entry into the operating system. The calls and usage are simple and the user has control over what happens.

3.2 File Descriptors

In the ZEUS operating system, all input and output is done by reading or writing files, because all peripheral devices (including the user's terminal) are files in the file system. This means that a single, homogeneous interface handles all communication between a program and the peripheral devices.

Before reading or writing a file, the file must be opened. If a file to be written on does not exist, it is created. The system checks to see if the user has permission to write on a file and if the file exists. If everything is in order, the system returns a small, positive integer called a file descriptor. Whenever I/O occurs, the file descriptor identifies the file. All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in Section 3 are similar to file descriptors, except that file descriptors are more fundamental. A file pointer points to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the shell) runs a program, it opens three files (with file descriptors 0, 1, and 2) called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can perform terminal I/O without opening the files.

If I/O is redirected to and from files with < and >, as in

```
prog < infile > outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. If the input or output is associated with a pipe, the results are similar. Normally, file descriptor 2 remains attached to the terminal. Therefore, error messages can go to the terminal. To redirect the standard error output, type an ampersand (&) after the >. For example:

```
prog >& errsmgs
```

In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from or where its output goes, as long as it uses file 0 for input, and files 1 and 2 for output.

3.3 Read and Write

All input and output is done by the functions _read_ and _write_. For both read and write operations, the first argument is a file descriptor. The second argument is a buffer in the program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are:

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count of the number of bytes actually transferred. When reading, the number of bytes returned can be less than the number asked for, if fewer than _n_ bytes remain to be read. (When the file is a terminal, _read_ normally reads only up to the next new line, which is generally less than what was requested.) A return value of zero bytes implies _EOF_ and -1 indicates an error of some sort. For writing, the returned value is the number of bytes actually written; an error is returned if this number is not equal to the number of bytes requested.

The number of bytes to be read or written is arbitrary. The two most common values are 1, which means one unbuffered character at a time, and 512, which corresponds to a physical block size on some peripheral devices.

A simple program to copy the program's input to its output can now be written. This program copies anything to any-

thing, since the input and output can be redirected to any file or device.

```
#define   BUFSIZE   512  /* best size for ZEUS */

main()    /* copy input to output */
{
     char buf[BUFSIZE];
     int  n;

     while ((n = read(stdin, buf, BUFSIZE)) > 0)
          write(stdout, buf, n);
     exit(0);
}
```

If the file size is not a multiple of _BUFSIZE_, a _read_ returns a smaller number of bytes to be written by _write_. The next call to _read_ returns zero.

It is instructive to see how _read_ and _write_ can be used to construct higher-level routines like _getchar_ and _putchar_. For example, the followng is a version of _getchar_ that does unbuffered input.

```
#define   CMASK     0377 /* for making char's > 0 */

getchar() /* unbuffered single character input */
{
     char c;
     return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

The variable _c_ must be declared _char_, because _read_ accepts a character pointer. The character being returned must be masked with _0377_ (octal) to ensure that it is positive; otherwise, sign extension can make it negative.

The second version of _getchar_ inputs in big chunks and outputs the characters one at a time.

```
#define   CMASK     0377 /* for making char's > 0 */
#define   BUFSIZE   512

getchar() /* buffered version */
{
     static char    buf[BUFSIZE];
     static char    *bufp = buf;
     static int     n = 0;

     if (n == 0) {  /* buffer is empty */
          n = read(0, buf, BUFSIZE);
          bufp = buf;
```

```
        }
        return((--n >= 0) ? *bufp++ & CMASK : EOF);
    }
```

3.4 Open, Creat, Close, Unlink

Files must be explicitly opened to be read or written (unless they are the default standard input, output, and error files). The two system entry points for explicitly opening files are _open_ and _creat_.

The entry point _open_ is similar to _fopen_ (discussed in Section 3.2) except that instead of returning a file pointer, _open_ returns a file descriptor, which is an integer.

```
int fd;

fd = open(name, rwmode);
```

As with _fopen_, the name argument is a character string corresponding to the external file name. The access mode argument is different, however. The _rwmode_ argument is 0 for read, 1 for write, and 2 for read and write access. If any error occurs, _open_ returns -1; otherwise, it returns a valid file descriptor.

Trying to open a file that does not exist results in an error. The entry point _creat_ is provided to create new files or to rewrite old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called _name_, and -1 if not. If the file already exists, _creat_ truncates it to zero length. It is not an error to _creat_ a file that already exists.

If the file is new, _creat_ creates it with the _protection mode_ specified by the _pmode_ argument. In the ZEUS file system, there are nine bits of protection information associated with a file, controlling read, write, and execute permission for the owner of the file, for the owner's group, and for all others. A three-digit octal number is most convenient for specifying the permissions. For example, 0755 (octal) specifies read, write, and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the ZEUS utility _cp_, a program that copies one file to another.

(This version copies only one file and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)    /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
     int  f1, f2, n;
     char buf[BUFSIZE];

     if (argc != 3)
          error("Usage: cp from to", NULL);
     if ((f1 = open(argv[1], 0)) == -1)
          error("cp: can't open %s", argv[1]);
     if ((f2 = creat(argv[2], PMODE)) == -1)
          error("cp: can't create %s", argv[2]);

     while ((n = read(f1, buf, BUFSIZE)) > 0)
          if (write(f2, buf, n) != n)
               error("cp: write error", NULL);
     exit(0);
}

error(s1, s2)  /* print error message and die */
char *s1, *s2;
{
     printf(s1, s2);
     printf(`\n');
     exit(1);
}
```

As stated earlier, there is a limit to the number of files (typically 15-25) that a program can have open simultaneously. Accordingly, any program that processes many files must be prepared to reuse file descriptors. The routine _close_ breaks the connection between a file descriptor and an open file, freeing the file descriptor for use with some other file. Termination of a program via _exit_, or return from the main program, closes all open files.

The function _unlink_(_filename_) removes the file _filename_ from the file system.

3.5 Random Access With lseek

File I/O is normally sequential: each *read* or *write* is performed after the previous one. When necessary, however, a file can be read or written in an arbitrary order. The system call *lseek* provides a way to move around in a file without reading or writing.

```
lseek(fd, offset, origin);
```

forces the current position in the file, whose descriptor is *fd*, to move to position *offset*, which is taken relative to the location specified by *origin*. Subsequent reading or writing begins at that position. The argument *offset* is a long integer; *fd* and *origin* are integers. The argument *origin* can be 0, 1, or 2 to specify that *offset* is to be measured from the beginning, from the current position, or from the end of the file. For example, to append to a file and seek to the end before writing, type:

```
lseek(fd, 0L, 2);
```

To get back to the beginning (rewind), type:

```
lseek(fd, 0L, 0);
```

The *0L* argument can also be written as (*long*) *0*.

With *lseek*, it is possible to treat files like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from an arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
     lseek(fd, pos, 0);  /* get to pos */
     return(read(fd, buf, n));
}
```

3.6 Error Processing

All routines that are direct entries into the system can incur errors. Usually an error is indicated by the return of a value. To enable the user to learn what sort of error occurred, all these routines leave an error number in the external cell *errno*. The meanings of the various error

numbers are listed in Section 2 of the _ZEUS Reference Manual_. If the reason for failure is to be printed out, the routine _perror_ must be used; this prints a message associated with the value of _errno_. The routine _sys_errno_ is an array of character strings that can be indexed by _errno_ and printed by the user's program.

SECTION 4

PROCESSES

4.1 System Function

This section describes how to execute a program from within another program.

The easiest way to execute a program from another program is to use the standard library routine *system*, which takes one argument, a command string exactly as typed at the terminal (except for the new line at the end), and executes it. For instance, to time-stamp the output of a program:

```
main()
{
	system("date");
	/* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of *sprintf* can be useful.

Remember that *getc* and *putc* normally buffer their input; terminal I/O is not properly synchronized unless this buffering is avoided. For output, use *fflush*; for input, see *setbuf* in Section 3.6.

4.2 Low-Level Process Creation

If the standard I/O library is not used, or if finer control is needed, calls to other programs must be contructed using the routines on which the standard library's *system* routine is based.

The most basic operation is execution of another program without returning, using the routine *execl*. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to *execl* is the file name of the command, whose address in the file system must be known. The second argument is conventionally the program name, but it is seldom used except as a place holder. If the command takes arguments, they are strung out after the program name. The end of the list is marked by a *NULL* argument.

The *execl* call overlays the existing program with the new one; it runs the new program and then exits. There is no return to the original program.

It is more common, however, for a program to fall into two or more phases that communicate only through temporary files. If this happens, it is natural to make the second pass simply an *execl* call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error (for example, if the file can't be found or is not executable). If the location of *date* is not known, enter

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

Use *execv*, a variant of *execl*, when the number of arguments is not known in advance. The call is

```
execv(filename, argp);
```

where *argp* is an array of pointers to the arguments. The last pointer in the array must be *NULL* so that *execv* can tell where the list ends. As with *execl*, *filename* is the file in which the program is found, and *argp*[*0*] is the name of the program. (This arrangement is identical to the *argv* array for program arguments.)

Because neither of these routines provides automatic search of multiple directories, the location of the command must be precisely known. The expansion of metacharacters like <, >, *, ?, and [] in the argument list cannot be obtained. If these metacharacters are desired, use *execl* to invoke the shell (*sh*), which then does all the work. A string *commandline* that contains the complete command as it would have been typed at the terminal is constructed. Then enter:

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, /*bin*/*sh*. Its argument -*c* means that the next argument should be treated as a whole command line. The only problem is in constructing the right information in *commandline*.

4.3 Control of Processes

The following explains how to regain control after running a program with *execl* or *execv*. Since these routines simply overlay the new program on the old one, to save the old one

requires that it first be split into two copies; one of these copies can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called _fork_.

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of the process ID (_proc_id_). In one of these processes (the child), _proc_id_ is zero. In the other (the parent), _proc_id_ is nonzero--it is the process number of the child. Thus, the basic way to call and return from another program is

```
if (fork() == 0)
execl("/bin/sh", "sh", "-c", cmd, NULL);  /* in child */
```

In fact, except for handling errors, this is sufficient. The _fork_ makes two copies of the program. In the child, the value returned by _fork_ is zero. It calls _execl_, which does the command and then dies. In the parent, _fork_ returns nonzero, so it skips the _execl_. (If there is any error, _fork_ returns -1).

More often, the parent waits for the child to terminate before it continues. This is done with the function _wait_:

```
int status;

if (fork() == 0)
     execl(...);
wait(&status);
```

This still does not handle any abnormal conditions, such as a failure of _execl_ or _fork_, or the possibility that there might be more than one child running simultaneously. (The _wait_ returns the process ID of the terminated child, which can be checked against the value returned by _fork_.) Also, this fragment does not deal with any abnormal behavior on the part of the child (which is reported in _status_). However, these three lines are the heart of the standard library's _system_ routine.

The _status_ returned by _wait_ encodes in its eight low-order bits the child's termination status. A 0 indicates normal termination, and a nonzero indicates various kinds of problems. The next higher eight bits are taken from the argument of the call to _exit_, which causes a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors (0, 1, and 2) point to the correct files; all other possible file descriptors are available for use. When this program calls another program, make certain the same conditions hold. Neither _fork_ nor the _exec_ calls affect open files. If the parent is buffering output that must be output before the output from the child, the parent must flush its buffers before the _execl_. Conversely, if a caller buffers an input stream, the called program loses any information that has been read by the caller.

4.4 Pipes

A _pipe_ is an I/O channel used between two processes. One process writes into the pipe, while the other reads. The system buffers the data and synchronizes the two processes. Most pipes are created by the shell, as in:

```
ls | pr
```

which connects the standard output of _ls_ to the standard input of _pr_. Sometimes, however, it is more convenient for a process to set up its own commands.

The system call _pipe_ creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned. The actual usage is like the following:

```
int  fd[2];

stat = pipe(fd);
if (stat == -1)
     /* there was an error ... */
```

The _fd_ is an array of two file descriptors, where _fd_[_0_] is the read side of the pipe and _fd_[_1_] is the write side. These can be used in _read_, _write_, and _close_ calls, just like any other file descriptors.

If a process attempts to read a pipe that is empty, it waits until data arrives. If a process attempts to write into a pipe that is full, it waits until the pipe empties. If the write side of the pipe is closed, a subsequent _read_ encounters _EOF_.

The following example illustrates the use of pipes. A function called _popen_(_cmd_, _mode_) creates a process _cmd_ and returns a file descriptor that either reads or writes the process, according to _mode_. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the _pr_ command. Subsequent _write_ calls using the file descriptor _fout_ send data to that process through the pipe.

The function _popen_ first creates the pipe with a _pipe_ system call, then forks to create two copies of itself. The child determines whether to read or write. It closes the other side of the pipe, then calls the shell (via _execl_) to run the desired process. The parent, likewise, closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests execute properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file because there is one potentially active writer.

```
#include <stdio.h>

#define   READ 0
#define   WRITE     1
#define   tst(a, b) (mode == READ ? (b) : (a))
static    int  popen_pid;

popen(cmd, mode)
char *cmd;
int  mode;
{
     int p[2];
     if (pipe(p) < 0)
          return(NULL);
     if ((popen_pid = fork()) == 0) {
          close(tst(p[WRITE], p[READ]));
          close(tst(0, 1));
          dup(tst(p[READ], p[WRITE]));
          close(tst(p[READ], p[WRITE]));
          execl("/bin/sh", "sh", "-c", cmd, 0);
          _exit(1); /* disaster has occurred if we get here */
     }
     if (popen_pid == -1)
          return(NULL);
     close(tst(p[READ], p[WRITE]));
     return(tst(p[WRITE], p[READ]));
}
```

The sequence of closes in the child is as follows. The task is to create a child process that reads data from the parent. The first _close_ closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The close closes file descriptor 0, the standard input. The system call dup returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order, and the first available one is returned, so the effect of the dup is to copy the file descriptor for the pipe (read side) to file descriptor 0. Thus, the read side of the pipe becomes the standard input. Finally, the old read side of the pipe is closed. A similar sequence of operations takes place when the child process is supposed to write from the parent instead of read.

The function pclose closes the pipe created by popen. The main reason for using a function other than close is to wait for the termination of the child process. The return value from pclose indicates whether or not the process succeeded. Equally important, when a process creates several children, is that only a certain number of unwaited-for children can exist, even if some of them have terminated. Performing the wait removes the child from the unwaited-for status. For example,

```
#include <signal.h>

pclose(fd)      /* close pipe fd */
int fd;
{
     register r, (*hstat)(), (*istat)(), (*qstat)();
     int  status;
     extern int popen_pid;

     close(fd);
     istat = signal(SIGINT, SIG_IGN);
     qstat = signal(SIGQUIT, SIG_IGN);
     hstat = signal(SIGHUP, SIG_IGN);
     while ((r = wait(&status)) != popen_pid && r != -1);
     if (r == -1)
          status = -1;
     signal(SIGINT, istat);
     signal(SIGQUIT, qstat);
     signal(SIGHUP, hstat);
     return(status);
}
```

The calls to signal ensure that no interrupts occure during the wait process.

The routine as written is limited in that only one pipe can be open at one time because of the single shared variable popen_pid. A popen function, with slightly different arguments and return values, is available as part of the

standard I/O library discussed in Section 3. As currently written, it shares the same limitation.

SECTION 5

SIGNALS

5.1 General

This section discusses external signals and program faults. Since nothing useful can be done within C about program faults that arise from illegal memory references or from execution of peculiar instructions, the following discussion concerns only external signals:

- *interrupt*: sent when the DEL character is typed
- *quit*: generated by control backslash
- *hangup*: caused by hanging up the phone
- *terminate*: generated by the *kill* command

When one of these events occurs, the signal is sent to all processes that were started from the corresponding terminal. Unless other arrangements have been made, the signal terminates the process. In *quit*, a core image file is written for debugging purposes.

5.2 Signal Routine

The routine *signal* alters the default action. It has two arguments. The first specifies the signal, and the second specifies how to treat it. The first argument is a number code. The second, the address, is either a function or a code that requests that the signal either be ignored or be given the default action. The include file *signal.h* gives names for the various arguments and must be included when *signal* is used. For example,

```
#include <signal.h>
 ...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, *signal* returns the previous value of the signal. The second argument to *signal* can be the name of a function (which has to be declared explicitly if it has not been

compiled). In this case, the named routine is called when the signal occurs. This facility is generally used by the program to clean up unfinished business before it terminates. For example, to delete a temporary file:

```
#include <signal.h>

main()
{
     int onintr();

     if (signal(SIGINT, SIG_IGN) != SIG_IGN)
          signal(SIGINT, onintr);

     /* Process ... */

     exit(0);
}

onintr()
{
     unlink(tempfile);
     exit(1);
}
```

5.3 Interrupts

Signals like INTERRUPT are sent to all processes started from a particular terminal. When a program is to be run noninteractively (started by &), the shell pervents it from receiving interrupts. If the program begins by announcing that all interrupts are to be sent to the _onintr_ routine, this command cancels the shell's effort to protect it when the program is run in the background.

The solution to this is to test the state of interrupt handling and continue to ignore interrupts if they are already being ignored. The program code depends on the fact that _signal_ returns the previous state of a particular signal. If signals are already being ignored, the process continues to ignore them; otherwise, they are caught.

A more sophisticated program can intercept an interrupt and interpret the interrupt as a request for the program to stop executing and return to its own command-processing loop. In a text editor, interrupting a long printout should not cause the editor to terminate and lose the work already done. The outline of the code for this can be written as follows:

```
#include <signal.h>
#include <setret.h>
ret_buf   sjbuf;

main()
{
     int (*istat)(), onintr();

     istat = signal(SIGINT, SIG_IGN);
                              /* save original status */
     setret(sjbuf); /* save current stack position */
     if (istat != SIG_IGN)
          signal(SIGINT, onintr);

     /* main processing loop */
}

onintr()
{
     printf("0nterrupt0);
     longret(sjbuf);     /* return to saved state */
}
```

The include file _setret.h_ declares the type _ret buf_ to be an object in which the state can be saved. The _sjbuf_ type, an array, is such an object. The _setret_ routine saves the state. When an interrupt occurs, a call is forced to the _onintr_ routine, which can print a message and set flags. The _longret_ routine takes as an argument an object stored by _setret_ and restores control to the location after the call to _setret_. Thus, control is returned to the position in the main routine where the signal is set up and where the main loop entered. Notice that the signal gets set again after an interrupt occurs. This is necessary because most signals are automatically reset to their default action when they occur. Functions containing calls to _setret_() should not have any register variable declarations.

Some programs that need to detect signals cannot be stopped at an arbitrary point. If the routine calls on the occurrence of a signal, sets a flag, and then returns instead of calling _exit_ or _longret_, execution continues at the exact point it was interrupted. The interrupt flag can be tested later.

One difficulty associated with the above approach arises if the program is reading data from the terminal when the interrupt is sent. The specified routine is called, and it sets its flag and returns. If execution resumes at the exact point it was interrupted, the program continues reading data from the terminal until another line is entered. This response could be confusing, since it might not be

obvious that the program is reading. It is better to have the signal take effect instantly. To resolve this difficulty, terminate the terminal read when execution resumes, this returns an error code indicating what happened.

Programs that catch and resume execution after signals should be designed to handle errors that are caused by interrupted system calls such as reads from a terminal, _wait_, and _pause_. A program whose _onintr_ program only sets _intflag_, resets the interrupt signal, and returns, should include code such as the following, when it reads the standard input:

```
if (getchar() == EOF)
     if (intflag)
          /* EOF caused by interrupt */
     else
          /* true end-of-file */
```

When signal-catching is combined with execution of other programs, the code should look something like the following:

```
if (fork() == 0)
     execl(...);
signal(SIGINT, SIG_IGN); /* ignore interrupts */
wait(&status); /* until the child is done */
signal(SIGINT, onintr);  /* restore interrupts */
```

If the program called catches its own interrupts, when the subprogram is interrupted, it gets the signal and returns to its main loop, and probably reads data from the terminal. But the calling program also pops out of its wait for the subprogram and reads the terminal. The system does not have a protocol for determining which program gets each line of input. A simple solution is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function _system_:

```
#include <signal.h>

system(s) /* run command string s */
char *s;
{
     int status, pid, w;
     register int (*istat)(), (*qstat)();

     if ((pid = fork()) == 0) {
          execl("/bin/sh", "sh", "-c", s, 0);
          _exit(127);
     }
     istat = signal(SIGINT, SIG_IGN);
     qstat = signal(SIGQUIT, SIG_IGN);
     while ((w = wait(&status)) != pid && w != -1)
          ;
     if (w == -1)
          status = -1;
     signal(SIGINT, istat);
     signal(SIGQUIT, qstat);
     return(status);
}
```

S8000 PLZ/SYS USER GUIDE

PREFACE

This document describes how PLZ/SYS source programs are run under ZEUS on the S8000. Details about invoking the compiler and code generator, and information about execution requirements and conventions are included.

PLZ/SYS source programs running under ZEUS are discussed in Section 2.

The operation of the PLZ/SYS compiler is described in Section 3. Use of the code generator is discussed in Section 4.

The implementation conventions used in the representation and execution of PLZ/SYS programs on the S8000 are described in Section 5. Programmers writing PLZ/ASM modules that are linked with PLZ/SYS modules will find the necessary information in this section.

Examples of a PLZ/SYS module and an equivalent PLZ/ASM module are given in Section 6.

The compiler and code generator error number explanations appear in Appendices A and B.

For a description of the language PLZ/SYS, refer to _Report on the Programming Language PLZ/SYS_ by Snook, Bass, Roberts, Nahapetian, and Fay (Springer-Verlag, 1978) and to _Introduction to Microprocessor Programming Using PLZ_ by Conway, Gries, Fay, and Bass (Winthrop, Cambridge, Massachusetts, 1979).

Other documents describing PLZ program preparation for S8000 include:

- _Z8000 PLZ/ASM Assembly Language Programming Manual_, Zilog part number 03-3055
- _Z8000 PLZ/ASM Assembler User Guide_, Zilog part number 03-3078
- _ZEUS Reference Manual, Zilog part number 03-3195 (plz(1), plzsys(1), plzcg(1), and uimage(1))_

The implementation of PLZ/SYS on the S8000 incorporates several extensions to the original language. These extensions are documented in _Addendum to the Report on the Programming Language PLZ/SYS_ (Zilog part number 03-3136).

TABLE OF CONTENTS

SECTION 1 INTRODUCTION 7

SECTION 2 PLZ/SYS RUNNING UNDER ZEUS 11

2.1 Overview 11
2.2 Limitations 11
2.3 Run-Time Conventions 11

SECTION 3 PLZ/SYS COMPILER 13

3.1 Overview 13
3.2 Plzsys Command Line 13
3.3 PLZ/SYS Version 3.1 Features and Limitations 14

3.3.1 Character Conventions 15
3.3.2 Character Sequence and Identifier Length 15
3.3.3 Source Line Length 15
3.3.4 Procedure, Data, and Program Size Limitations 15
3.3.5 Error Recovery 15
3.3.6 Compiler Evaluation of Constant Expressions 16
3.3.7 Literal Constants or Compile-Time Constant Expressions 16
3.3.8 Constant Type Determination 17
3.3.9 Structured Return Parameters 18

SECTION 4 CODE GENERATOR 21

4.1 Overview 21
4.2 Plzcg Command Line 21

SECTION 5 PLZ/SYS IMPLEMENTATION CONVENTIONS FOR THE Z8000 23

5.1 Overview 23
5.2 Data Representation 23

TABLE OF CONTENTS (continued)

5.2.1 Primitive Data Type Representation 23
5.2.2 Structured Data Type Representation 25

5.3 Data Alignment 25
5.4 Data Access Methods 27
5.5 Run-Time Storage Administration 28

5.5.1 Nonsegmented Code 28
5.5.2 Segmented Code 30

5.6 Register Conventions 32

5.6.1 Nonsegmented Code 32
5.6.2 Segmented Code 33

5.7 Execution Preparation 33

5.7.1 Nonsegmented Code 33
5.7.2 Segmented Code 33

SECTION 6 PLZ/SYS - PLZ/ASM INTERFACE EXAMPLE 35

6.1 Purpose 35
6.2 Nonsegmented Code 37
6.3 Segmented Code 40

APPENDIX A PLZ/SYS ERROR MESSAGES 47

APPENDIX B PLZCG ERROR NUMBERS AND EXPLANATIONS 51

TABLE OF CONTENTS (continued)

LIST OF TABLES

Table

3-1 Evaluation of Constant Expressions 17

LIST OF ILLUSTRATIONS

Figure

1-1 Linking of PLZ/SYS and PLZ/ASM Source Code 9

5-1 Nonsegmented Run-Time Stack--General Layout 29
5-2 Segmented Run-Time Stack--General Layout 31

6-1 Example 2: PLZ/ASM Module for the Nonsegemented S8000 36
6-2 Nonsegmented Run-Time Stack Detail After Entry Sequence 38
6-3 Nonsegmented Run-Time Stak Detail Before Recursive Call 39
6-4 Example 3: PLZ/ASM Module for the Segmented S8000.. 42
6-5 Segmented Run-Time Stack Detail After Entry Sequence... 44
6-6 Segmented Run-Time Stack Detail Before Recursive Call.. 45

SECTION 1

INTRODUCTION

The PLZ/SYS compiler (plzsys), code generator (plzcg), and the package driver program (plz) are described in this document. When used in conjuntion with a ZEUS editor, PLZ/SYS source files can be created and processed into a linked, relocatable object module suitable for running under ZEUS or loading into a standard Z8000. Programs can be prepared for either the segmented or nonsegmented version of the Z8000 microprocessor.

A PLZ/SYS program is composed of separately compiled source modules. A PLZ/SYS source module can contain control lines of the form

```
#include "filename"
```

Such a control line causes the replacement of itself by the entire contents of the file _filename_..

There are four stages in the process of a PLZ/SYS source module. They are:

1. Replace all control lines in the source module.

2. Use plzsys to generate an intermediate z-code module.

3. Use plzcg to generate a machine code module in zobj format.

4. Use uimage to translate the result of Step 3 into a.out format.

After all the PLZ/SYS source modules in a program are processed, the ZEUS linker (ld) can be invoked to link all these machine code modules with possibly other existing machine code modules (libraries, assembler output, or C compiler output) to produce an object module that can be run under ZEUS (Figure 1-1).

The PLZ.IO I/O package is contained in the library /lib/libp.a. Plz-callable versions of the ZEUS system calls are also in /lib/libp.a.

In this document, all file extensions are written in lowercase. However, uppercase extensions .P and .Z are also acceptable by these programs.

BLOCK DIAGRAM

TO BE SUPPLIED

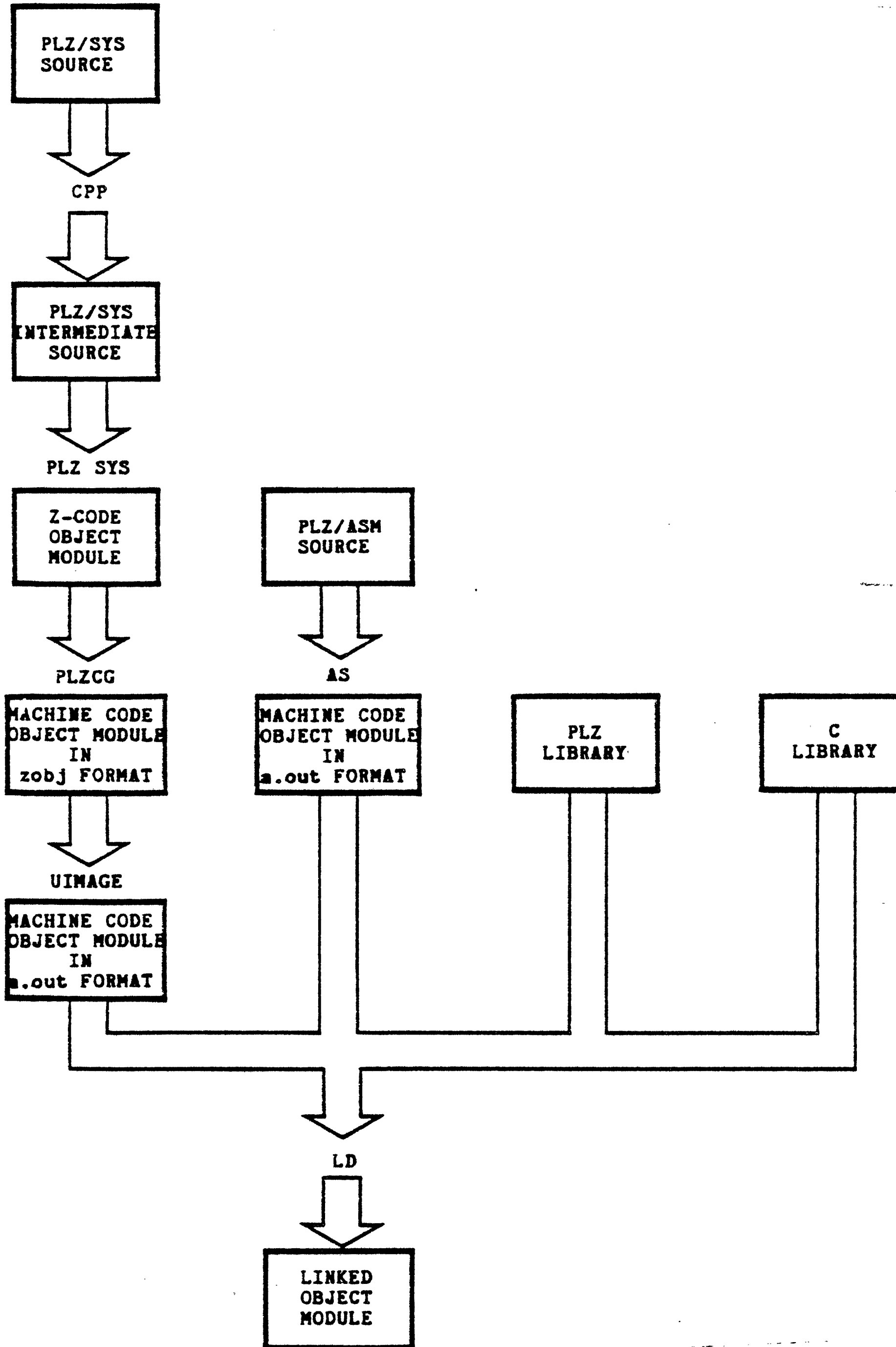


Figure 1-1. Linking of PLZ/SYS and PLZ/ASM Source Code

SECTION 2

PLZ/SYS RUNNING UNDER ZEUS

2.1 Overview

PLZ/SYS source programs intended to run under ZEUS can be compiled, code generated, and linked using the simplified user interface, plz(1). Plz is a driver for the compiler and code generator which, along with the assembler, C preprocessor, and ZEUS linker, are invoked automatically with default command line options. Together they produce an object module that is loaded and run by ZEUS.

The plz driver programs work similarly to cc(1) for C programs. To compile a plz source program composed of several modules, a single command must be issued to produce a ZEUS-loadable program. For example, a program consisting of three PLZ/SYS modules, a.p, b.p, c.p, and the PLZ/ASM modules d.s and e.s is compiled by:

```
%plz a.p b.p c.p d.s e.s -o program
```

leaving the output on the file program. Default output is a.out. Several options are accepted by plz and are explained in the ZEUS Reference Manual under plz(1).

2.2 Limitations

The plz programs created with the plz driver program are limited because they cannot contain z-code modules. This is because the ZEUS linker cannot create the appropriate tables to link z-code. (See ld(1) in the ZEUS Reference Manual.)

2.3 Run-Time Conventions

PLZ/SYS programs running under ZEUS must have an entry point called main. The declaration for main is:

```
global

main procedure (argc integer, argv ^^byte)
     returns (retcd integer)
```

where argc is the number of arguments supplied by ZEUS to the program, and argv is a pointer to an array of pointers,

one for each argument. The return parameter _retcd_ is zero for normal termination. An error is indicated by a nonzero return.

ZEUS system calls are supported and can be called from PLZ/SYS programs. The library /lib/libp.a contains a ZEUS implementation of the PLZ.IO I/O package and plz-callable versions of the system call library. There are some limitations, however. The variable number of argument forms of _exec_ (_execl_, _execle_, etc.) are not supported. The exit system call is renamed Exit to differentiate it from the plz "exit" reserved word. The signal system call requires function parameters that plz/sys does not allow. Therefore, the signal system call cannot be called.

NOTE

Releases of the PLZ/SYS compiler dated from September 30, 1981, will conform to the S8000 calling conventions instead of those described in Sections 5 and 6. Programs compiled under these releases will be able to declare ZEUS Utilities and C functions as external procedures and invoke them directly. The library /lib/libp.a will no longer be necessary.

SECTION 3

PLZ/SYS COMPILER

3.1 Overview

The PLZ/SYS compiler translates source code modules into intermediate code. The ZEUS editor is used to create PLZ/SYS source modules. The source file name must end with the file name extension .p.

With the -l option, the PLZ/SYS compiler creates a listing file with the default source file name with the extension .l rather than .p, and an object file with the default extension .z. In creating the object file, plzsys uses a temporary scratch file that is deleted when compilation is finished. The listing file contains the source code with line numbers, statement numbers, and syntax error messages. The messages consist of a pointer to each erroneous token, followed by an error number for each pointer. The list of error numbers in Appendix A can be used to determine the corresponding compilation error. Occasionally, the pointer does not point directly at the incorrect token. Error messages can be copied to a separate file with the error (-e) option described in Section 3.2.

The object file contains z-code. The plzcg code generator compiles z-code to Z8000 machine code.

3.2 Plzsys Command Line

In the following description, the word _filename_ is used to specify an arbitrary ZEUS path name.

The compiler is invoked by the following general shell command line. Do not type the square brackets; they simply indicate that options are not required.

```
plzsys [options] filename
```

where _filename_ contains the source for a single plz module. The extension .p is optional; if it is missing, the compiler appends it before attempting to open the file. The options listed below can appear in any order, separated by delimiters.

Option	Function
-l	Creates a listing file with .l substituted for the .p extension of source file. Default is no listing.
-o	Assigns the name filename to the object file, instead of the default source file name with the extension .z. If no object is desired, use /dev/null for filename.
-e	Copies error messages to the file whose name is the same as the source file with extension .e. If no errors occur, the error file is deleted at the end of compilation.
-nd	Omits symbol, type, constant, and statement number information for a hypothetical debugger. The default is to generate debug symbols.
-nc	Omits debug symbol information for any CONSTANT names. The default is to generate named constants when generating debug symbols.
-t Z80	Generates output suitable for the Z80. Does not allow extensions to plzsys such as long variables and structure comparison and assignment. The output can run on MCZ only.
-t Z8000s	Generates output suitable for the segmented Z8000. Treats pointers as four-byte objects, instead of two-byte objects. Aligns word-size data on even addresses. Allows long variables and structure comparison and assignment.
-t Z8000ns	Generates output suitable for the nonsegmented Z8000. Allows long variables and structure comparison and assignment. This is the default.

3.3 PLZ/SYS Version 3.1 Features and Limitations

The following Z8000 PLZ/SYS features and limitations are dependent on site implementation.

3.3.1 Character Conventions

The PLZ/SYS compiler uses the standard ASCII character set. Upper or lowercase characters are recognized and treated as different characters; therefore, keywords are recognized only if they are either all upper or all lowercase. For example, GLOBAL and global are recognized as keywords, but Global is not. Hexadecimal numbers and special string characters can be either upper or lowercase.

3.3.2 Character Sequence and Identifier Length

A character sequence cannot be less than one character or more than 255 characters. Identifiers can be any length less than 256 characters; however, only the first 127 characters determine the uniqueness of the name.

3.3.3 Source Line Length

Source lines of more than 120 characters are accepted, but are truncated in the listing. The entire listing line, including line numbers and statement numbers, can be up to 132 characters. Comments and quoted character sequences can extend over an arbitrary number of lines. Mismatched comment delimiters (!) or character sequence delimiters (') must be avoided.

3.3.4 Procedure, Data, and Program Size Limitations

A single procedure cannot be larger than 1000 bytes of intermediate code.

Data and program addressing within a module are limited to 16-bit quantities. Consequently, a module cannot contain more than 65536 bytes of data or z-code.

3.3.5 Error Recovery

Error recovery by the compiler is limited. If an error is discovered, symbols can be scanned without being checked until the compiler can continue. Within CONSTANT, TYPE, or variable declarations, the compiler can skip ahead until it finds the next keyword (CONSTANT, TYPE, GLOBAL, EXTERNAL, or INTERNAL) that starts a declaration class. Within procedure declarations, the compiler skips ahead until it finds the next keyword (IF, DO, EXIT, REPEAT, RETURN, END, etc.) that starts a new statement. This skipping ahead can cause

several compilations before all errors are detected and removed.

3.3.6 Compiler Evaluation of Constant Expressions

Numeric constants are represented internally as 16-bit quantities. Each operand in a constant expression is evaluated as if it is declared to be of type WORD. Thus, 4/2 equals 2, but 4/-2 equals 0, since -2 is represented as a very large positive number. There is no overflow checking during evaluation of a constant expression. Since constants are represented as 16-bit values, a maximum of two characters are allowed in a character sequence used as a constant. The order of bytes within a WORD quantity is implementation-dependent when stored in memory. Programs that depend on a certain order (high-order, then low-order as in the PLZ/SYS implementation on the Z8000) cannot transport easily to other machines or translators.

3.3.7 Literal Constants or Compile-Time Constant Expressions

Error 240 occurs if a literal constant greater than 65535 is used. Constant expressions that must be evaluated at compile time (such as initial values or CASE-select elements) are restricted. Constant expressions are evaluated using 16-bit operations on 16-bit quantities so no error message is given.

When used with long (32-bit) types, a constant or constant expression must be converted to 32 bits. This conversion is performed by the compiler as follows:

- If the constant or constant expression must be LONG, then the 16-bit quantity is assumed to be WORD, and a WORD-to-LONG conversion is performed. (The WORD is right-justified in a field of zero bits.)

- If the constant or constant expression must be LONG_INTEGER, then the 16-bit quantity is assumed to be INTEGER, and an INTEGER-to-LONG_INTEGER conversion is performed. (The INTEGER is sign-extended.)

When a constant appears in a LONG executable expression (assignment or parameter), the constant is always treated as a 32-bit quantity with the high 16 bits all zeros, and any operations on the constant are full 32-bit operations. This includes negation (-) and operations with other constants.

Unlike initial values and CASE-select elements, executable expressions are evaluated at run time by the target machine (the Z8000), which accommodates long operations.

Run-time and compile-time long constant expressions have the same value in many cases, such as when the type is LONG__INTEGER and the value is in the range -32768 to 32767 (using the "-" operator to represent negative constants), or the type is LONG and the value is in the range 0 to 65535.

Table 3-1 gives examples of compile-time and run-time evaluation of constant expressions. An executable expression must be used to create a 32-bit value whose high-order word is neither %FFFF nor 0.

Table 3-1. Evaluation of Constant Expressions

Compile-Time Constant Expression	Value	Run-Time Constant Expression	Value
L LONG := -1	%0000FFFF	L := -1	%FFFFFFFF (*)
LI LONG_INTEGER := -1	%FFFFFFFF	LI := -1	%FFFFFFFF (*)
L LONG := %FFFF	%0000FFFF	L := %FFFF	%0000FFFF
LI LONG_INTEGER := %FFFF	%FFFFFFFF	LI := %FFFF	%0000FFFF
L LONG := -%FFFF	%00000001	L := -%FFFF	%FFFF0001 (*)
LI LONG_INTEGER := -%FFFF	%00000001	LI := -%FFFF	%FFFF0001 (*)
L LONG := %AAAA*(%FFFF+1) + %BBBB	%0000BBBB	L := %AAAA*(%FFFF+1) + %BBBB	%AAAABBBB

(*) "-" is a run-time unary operator in these cases.

3.3.8 Constant Type Determination

The compiler can usually determine from context the type of constant load (long or word) to generate. For example, in the assignment statement

```
X := 24
```

the compiler generates a word if X is a 16-bit quantity, and a long word if X is a 32-bit quantity. Similarly, it determines the type of constant in parameter lists, case expressions, and most relational expressions. The only instance in which the compiler cannot determine from context what

type of constant to generate is in a relational expression where the constant appears lexically before any variable appears (0<X).

To generate the correct constant, the compiler functions as if long constants are required. When the compiler finally determines what the type should be, it backs up and generates the proper constants. There are two important consequences of this:

1. A maximum of 16 constants can be corrected. Error 236 occurs if more than 16 constants are encountered before their type can be established.

2. If the proper type of the constant is WORD, then one or more NOP (No-op) instructions (one byte each) appears in the z-code. This lengthens the code and slows execution slightly.

To avoid these problems, reverse the order of operands in the relational expression; use X>0 instead of 0<X.

3.3.9 Structured Return Parameters

The compiler does not allow field selection of a record-return value or indexing of an array-return value.

Thus, in the context of

```
EXTERNAL
    PROCA   PROCEDURE RETURNS  (ARRAY [10 BYTE])
    PROCR   PROCEDURE RETURNS  (RECORD [F1 F2 BYTE])
```

the following expressions are not accepted by the compiler:

```
PROCA()[2] PROCR().F1
```

The only operations allowed on array- and record-return parameters are assignment and comparison.

The compiler allows dereferencing of pointer-valued procedures:

```
EXTERNAL PROCP PROCEDURE RETURNS (^BYTE)
...
PROCP()^
```

When the return value is a structure that will not be copied, it can be replaced by a pointer-return value that can be dereferenced and then indexed or field selected:

```
TYPE
    ATYPE ARRAY [S BYTE]
EXTERNAL
    PROCA PROCEDURE RETURNS (^ATYPE)
...
    PROCA()^[I]
...
```

SECTION 4

CODE GENERATOR

4.1 Overview

PLZ/SYS compiler output is a z-code object module that cannot be executed directly on the S8000. The z-code must be processed by the code generator to produce a machine-code object module.

This section describes how to invoke the code generator and select from the available options. The Z8000 plz code generator accepts a file of intermediate z-code as input and produces a file of Z8000 relocatable object code in Zobj format as output. This output must be translated into a.out format by uimage. The output of uimage is linked with other a.out format modules to form the complete executable load module.

4.2 Plzcg Command Line

The code generator is invoked by the following ZEUS command line:

```
plzcg [-o filename2] [-s] [-l] [-v] filenamel
```

where filenamel can have the extension .z. The extension .z in the command line is optional; if missing, the code generator appends it before attempting to open the file. In the absence of the -o filename2 option, the generated object file has the name t.out; otherwise, the object code is generated in the file named filename2.

The shared code (-s) option is significant only for code destined for the segmented Z8000 processor. The procedures in a shared code module can be invoked and executed by different programs with independently allocated stacks, without altering the shared code module. This is possible because the local variable and parameters of a shared code module are accessed on the calling program. Nonshared code modules contain stack references in the code that are unchangeable during execution.

The -l options produces a pseudo-assembly languge listing of the module. The listing file has the same name as the input file with .l substituted for the .z suffix. No assembly listing is produced for the data in the module and there are

no symbolic labels. References to code are prefaced by the letter P; local data by L; global data by G.

The -v option causes plzcg to announce its presence when it starts and to tell how much code and data were produced when it finishes.

On the Z8000, stack-independent addressing of local and parameter data is achieved with loss of speed and compactness, so only modules that must be shared should be code-generated with the shared code option. The effects of the shared code option are described in more detail in Section 5.4.

SECTION 5

PLZ/SYS IMPLEMENTATION CONVENTIONS FOR THE Z8000

NOTE

Refer to Section 2 for applicability of these conventions to your release of PLZ/SYS and use of library functions.

5.1 Overview

This section describes PLZ/SYS program conventions for the Z8000. Included are details on data representation, data alignment, data access methods, run-time storage administration, and register conventions. This section concludes with a specification of the run-time environment required for proper program execution. It is assumed that the reader is familiar with the information in the Z8000 PLZ/ASM Assembly Language Programming Manual.

5.2 Data Representation

This section defines the representation of the seven predefined simple types available in PLZ/SYS on the Z8000: BYTE, SHORT_INTEGER, WORD, INTEGER, LONG, LONG_INTEGER, and pointer, and the storage layout of the structured types ARRAY and RECORD.

5.2.1 Primitive Data Type Representation

The seven predefined simple data types available in PLZ/SYS are represented on the Z8000 as follows:

BYTE

A BYTE value is a nonnegative integer in the range 0 to 255 (decimal) and is represented on the Z8000 as an unsigned eight-bit byte.

SHORT_INTEGER

A SHORT_INTEGER value is an integer in the range -128 to 127 and is represented on the Z8000 as a signed eight-bit byte in twos-complement notation.

WORD

A WORD value is a nonnegative integer in the range 0 to 65535 (decimal) and is represented on the Z8000 as an unsigned 16-bit word.

INTEGER

An INTEGER value is an integer in the range -32768 to 32767 and is represented on the Z8000 as a signed 16-bit word in twos-complement notation.

LONG

A LONG value is a nonnegative integer in the range 0 to 4,294,967,295 (decimal) and is represented on the Z8000 as an unsigned 32-bit long word.

LONG_INTEGER

A LONG_INTEGER value is an integer in the range -2,147,483,648 to 2,147,483,647 and is represented on the Z8000 as a signed 32-bit long word in twos-complement notation.

Pointer

Nonsegmented code:

A pointer value on the nonsegmented Z8000 is a storage address represented as a 16-bit word. The distinguished value NIL is represented by the value zero (0).

Segmented code:

A pointer value on the segmented Z8000 is a storage address composed of a seven-bit segment number and a 16-bit offset, represented as a 32-bit long word. The value of the pointer literal NIL is the long value zero (address 0): segment zero, offset zero.

NOTE

Because the size of a pointer is inherently dependent on specific machine configurations, programs that are to be easily transported

from one machine to another the user must avoid mixing pointer and nonpointer values in expressions.

5.2.2 Structured Data Type Representation

The PLZ/SYS structured types ARRAY and RECORD are represented on the Z8000 as follows:

ARRAY

Elements of an array are allocated consecutively into ascending storage addresses, beginning with element zero. Arrays are subject to the alignment constraints described in the next section.

RECORD

Fields within a record are stored in the order of declaration, subject to the alignment rules in Section 5.3.

5.3 Data Alignment

On the Z8000, all word and long data must begin on even addresses. The compiler aligns the data on even addresses relative to the start of a module. The compiler, code generator, and Z8000 assembler also extend each module to an even length. Thus, if the first module begins on an even address, all word and long data in that module and the PLZ/SYS modules that follow are correctly aligned.

The amount of storage wasted by aligning data is usually negligible, but becomes significant with the creation of certain structures. To avoid excessive waste, it is important to understand the following rules. The same rules are used by the Z8000 assembler, so that global data in PLZ/SYS can be accessed from assembly language and vice versa.

Rule 1: A structure (array or record) is aligned only if it contains a component that must be aligned.

Rule 2: A structure is padded to even length only if it contains a component that must be aligned.

Rule 3: Record fields are stored in the order declared and individually aligned as needed.

The following examples illustrate these rules:

```
TYPE
    RREC RECORD [F1, F2, F3 BYTE]
    SREC RECORD [F1 BYTE; F2 WORD; F3 BYTE]
    TREC RECORD [F1, F3 BYTE; F2 WORD]

INTERNAL

A ARRAY [9 BYTE]     ! Unaligned; 9 bytes !

B ARRAY [3 WORD]     ! Aligned; 6 bytes !

C RREC               ! Unaligned; 3 bytes !


D ARRAY [5 RREC]     ! Unaligned; 15 bytes !

E SREC               ! Aligned; 6 bytes (alignment byte !
                     ! after F1 and padding byte after F3) !

F ARRAY [5 SREC]     ! Aligned; 30 bytes--10 bytes wasted !

G TREC               ! Aligned; 4 bytes--no waste !

H ARRAY [5 TREC]     ! Aligned; 20 bytes--no waste !
```

Example D shows an array of records that do not have to be aligned. Examples G and H show how the information contained in variables E and F can be arranged more compactly. Such compactness is achieved by placing the fields that require alignment before the fields that do not require alignment in a record or by ensuring that all fields requiring alignment occur at an even offset from the start of the record.

In PLZ/SYS, the same storage area can be treated through pointers and type conversion. However, the contents of an unaligned structure cannot be treated as aligned objects. For example, the following program section might not execute as intended:

```
TYPE
    PTREC ^TREC  ! TREC defined above !
INTERNAL
    PTRT PTREC
    A ARRAY [(SIZEOF TREC) BYTE]
```

```
        W WORD
    ...
    PTRT := PTREC #A[0]
    W := PTRT^.F2      ! Fails if A begins on odd address !
```

To force A to be aligned, use

```
    A ARRAY [(SIZEOF TREC)/2 WORD]
```

There is no guarantee that the compiler will allocate data variables in order; consequently, a variable or structure that does not require alignment (for example, a byte array) might not be aligned even if it is declared immediately after an aligned variable of even length. However, a variable appearing alone in a module is aligned.

5.4 Data Access Methods

Data accessible to PLZ/SYS programs is divided into two storage classes: static data that is declared GLOBAL, INTERNAL, or EXTERNAL, and dynamic data that is declared LOCAL or declared as parameters.

Static data is allocated once, before execution begins, and is accessed by absolute addresses embedded in the code. On the Z8000, Direct Addressing mode is used to access static data.

Dynamic data is allocated during program execution on a run-time stack. The input and output parameters in a procedure are allocated on the stack before invoking the procedure. The called procedure allocates its local variables when it receives control. Within the body of the procedure, the input parameters passed to it, as well as the output parameters it yields, are accessed in exactly the same manner as local variables.

Based Addressing is the appropriate mode for accessing dynamic data on the Z8000. However, Based Addressing is available on only a restricted set of machine instructions. On the nonsegmented Z8000, Indexed mode is equivalent to Based mode, and can be used in most instructions to achieve the effect of Based Addressing. On the segmented Z8000, Indexed and Based modes are functionally distinct. However, Indexed mode can be used to access dynamic data by embedding the segment number of the Local Stack in the code. Use of Indexed Addressing implies that the Local Stack is restricted to the segment specified by the code. This segment

number is placed in the code during absolute address assignment, usually performed by the Imager.

If Indexed Addressing, instead of Based Addressing, is used for accessing dynamic data on the segmented Z8000, the code is more compact; it cannot be shared by independent programs. Because Indexed Addressing mode specifies the segment number in the code, it is impossible for distinct programs to share the code and not share local and parameter data as well. To allow for sharing at the expense of less efficient code, the code generator option SHARED can be specified on the command line. This ensures that local and parameter data is always accessed using Based Addressing mode.

5.5 Run-Time Storage Administration

PLZ/SYS procedures allocate local variables, expression temporaries, and parameters on a run-time stack. Stacks on the Z8000 grow toward lower addresses, so the most recently allocated word (top) of a stack is at the lowest address. Storage is allocated by decrementing a stack pointer and is released by incrementing the pointer. Stack pointers always refer to the top word on the stack, which must be at an even address.

The diagrams in this section show stacks as 16 bits wide, growing up toward the top of the page. Nonsegmented stacks are drawn with their base at storage address FFFE; actual stacks can begin anywhere. Segmented diagrams show each stack occupying an entire segment, growing up from storage address FFFE in each segment. To move the stack pointer up the stack, it must be decremented. In the following discussion, the word "above" means "closer to the top of the stack." An item above another on the stack is closer to the top of the diagram and is located at a lower memory address.

5.5.1 Nonsegmented Code

Nonsegmented code uses a single run-time stack. The portion of the stack visible to a single procedure can be divided into several zones (Figure 5-1).

The address of the top word on the stack is maintained in register R15, the Stack Pointer (SP) register. The two lowest zones, at the highest storage addresses, contain the return and input parameters passed by the caller. These zones are allocated on the stack by the calling procedure during the calling sequence. Immediately above the input

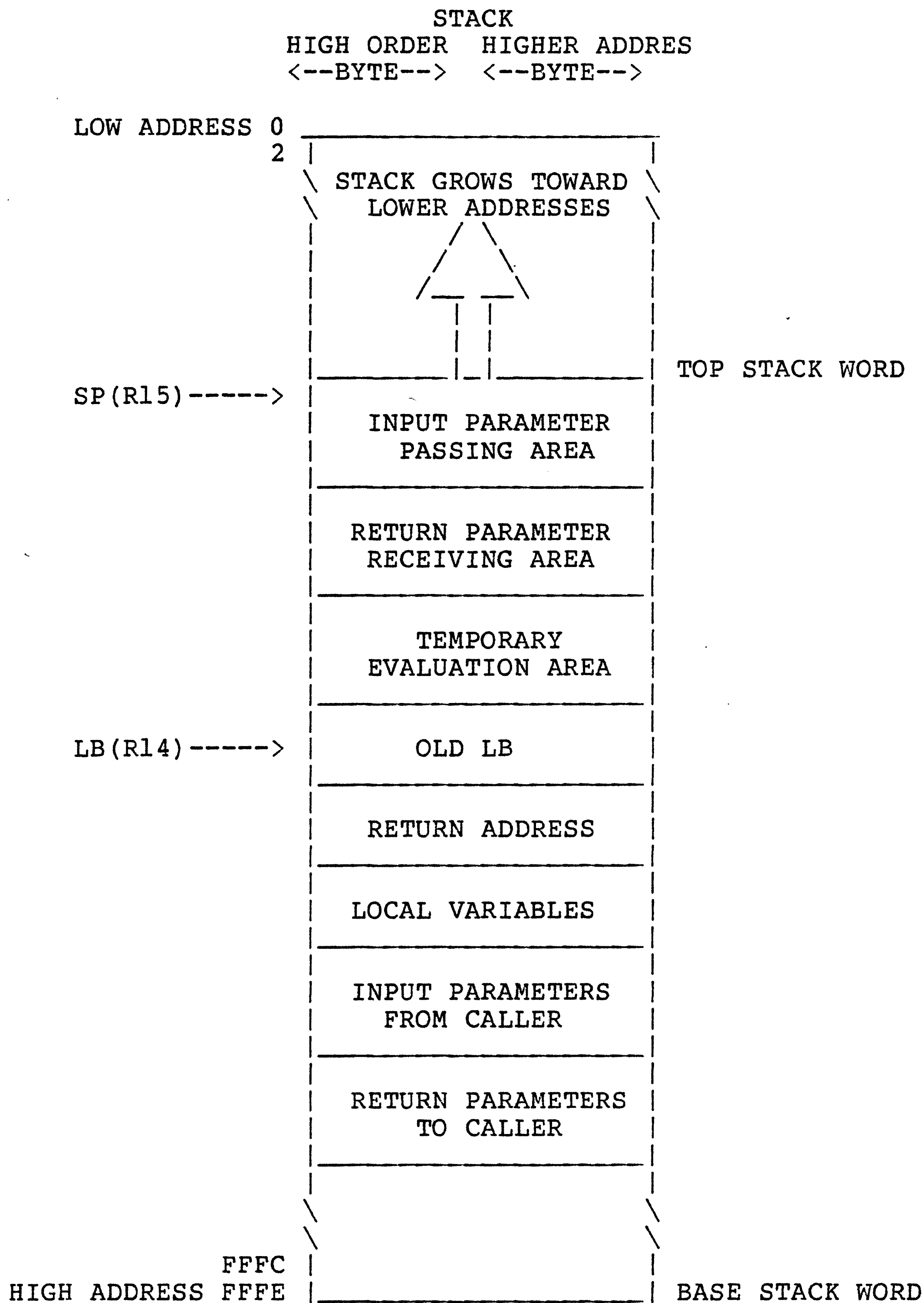


Figure 5-1. Nonsegmented Run-Time Stack--General Layout

parameters are the local variables. These are allocated at the start of the procedure before execution of the procedure body.

Above the local storage area are two words of control linkage information. The word immediately above local storage contains the return address. The next word contains the caller's Local Base address, which must be restored in register R14 before it is returned to the caller. During execution of the procedure body, register R14, the Local Base (LB) register, addresses this word. Local and parameter data is referenced by a positive offset from the LB register.

Above the control information is a dynamically changing expression evaluation area. The expression stack provides temporary storage for immediate results during the evaluation of arithmetic and logical expressions and receives parameters from, and passes arguments to, procedures invoked during evaluation of the body.

All parameters reside in an even number of bytes. Parameters of odd length are padded to an even number of bytes, and aligned on an even address. Byte parameters reside in the low-order eight bits of a word value, with an undefined high-order byte.

5.5.2 Segmented Code

Segmented code uses two stacks, Control and Local (Figure 5-2). The Local Stack contains all local variables, expression temporaries, and input and return parameters. The Control Stack contains return addresses and fixed-base pointers to the Local Stack. Neither stack is allowed to span segments. By separating return addresses from parameters, the two-stack scheme enables faster procedure linkage than that achieved using a single stack.

The portion of the Local Stack visible to any one procedure can be divided into the following zones. Return parameters delivered by the procedure occupy the lowest zone on the page, at the highest memory address. Above the return parameters in the diagram are the input parameters passed to the procedure. Storage for local variables declared within the procedure occupy the next zone. The uppermost zone, at the lowest memory address, is the expression evaluation area. This includes temporaries and parameters passed to, and slots for results received from, procedures called by this routine. The location of parameters is such that input

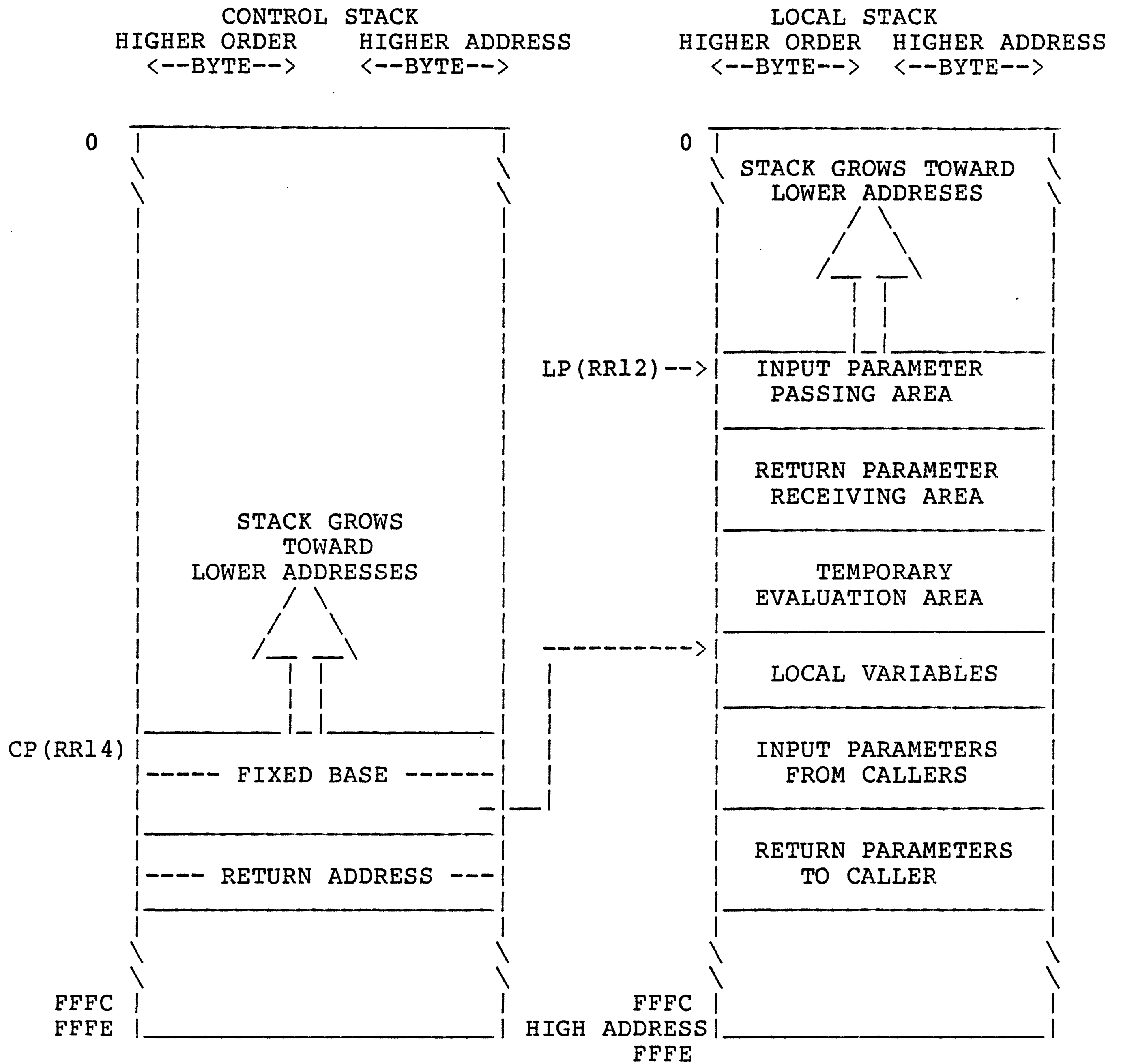


Figure 5-2. Segmented Run-Time Stacks--General Layout

parameters are evaluated and pushed on the stack, and return parameters are on the top of the stack following completion of a procedure call.

The Local Stack Pointer (LP) addresses the top word on the Local Stack and is maintained in register pair RR12. Parameters are passed by pushing them on the Local Stack; result parameters are accessed by popping them off. Local variables are accessed relative to LP, using either Based or Indexed Addressing modes. The local base address is not maintained in a register as with nonsegmented code. The displacement of local variables from LP varies during execution as temporaries or parameters are pushed and popped. However, the movement of LP is predictable during code generation and offsets to local variables can be adjusted for each reference.

The Control Stack Pointer (CP) addresses the top word of the Control Stack and is maintained in register pair RR14. This register is used by the Call and Return instructions to deposit and restore the program counter. Before execution of the procedure body, the location of the lowest-address word of local storage is pushed on the Control Stack. This address is not required for execution, but it is useful for run-time debugging since local and parameter data are difficult to locate, due to the transient nature of LP.

5.6 Register Conventions

In both segmented and nonsegmented code, two address registers are dedicated to specific purposes. These registers are used in accordance with the run-time storage management conventions outlined in Sections 5.5, 5.5.1, and 5.5.2. All other registers are available for local assignment within the body of a procedure and are subject to modification during any procedure call.

5.6.1 Nonsegmented Code

Register assignments in nonsegmented code are:

R15	Stack Pointer	(SP)
R14	Local Base	(LB)
R0-R13	Unassigned	

Procedures use registers 0 through 13 without saving their contents. The LB register, R14, is saved. Procedures remove input parameters passed to them on the stack. The SP register, R15, addresses the first return parameter after completion of value-returning procedures.

5.6.2 Segmented Code

Register assignments in segmented code are:

RR14	Control Stack Pointer	(CP)
RR12	Local Stack Pointer	(LP)
R0-R11	Unassigned	

Procedures use Registers 0 through 11 without saving their contents. CP, RR14, is saved. Procedures deallocate input parameters passed to them on the Local Stack. LP, RR12, addresses the first return parameter after completion of value-returning procedures.

5.7 Execution Preparation

To run PLZ/SYS programs on a standard Z8000, the run-time environment, assumed by the conventions specified in Section 5.6.2, must be established. The ZEUS Operating System automatically provides this function for PLZ/SYS programs, executed by System 8000. The preparations necessary for running segmented and nonsegmented code follow.

5.7.1 Nonsegmented Code

A region of memory adequate for the run-time stack must be allocated; the SP register must be set to the next highest word address. The first word allocated on the stack is at the highest memory address reserved for the stack. Any parameters for the main procedure must be passed in accordance with the calling conventions for nonsegmented code explained in Section 5.5.1. A return address is pushed on the stack and the main procedure is invoked by loading its entry address into the program counter. (This can be achieved by executing a call instruction.) The LB register does not require initialization.

5.7.2 Segmented Code

Segmented code requires the allocation of memory for two run-time stacks: the Control Stack and the Local Stack. These two stacks must not overlap. The CP register must be set to the next highest word address above the region reserved for the Control Stack. The LP register must be set to the next highest word address above the Local Stack.

On the Z8000, words must be located at an even memory address, so the contents of both LP and CP must be even. When LP and CP are properly initialized, any parameters for the main procedure must be pushed on the Local Stack in accordance with the calling conventions for segmented code outlined in Section 5.5.2. A return address must be pushed onto the Control Stack, and the main procedure must be invoked by loading its entry address into the program counter. (This can be achieved by executing a call instruction.)

As described in Section 4.4, programs containing modules processed by the code generator without the shared code option can specify the segment number of the Local Stack. For proper execution, the LP register should be initialized to address the next segment.

SECTION 6

PLZ/SYS - PLZ/ASM INTERFACE EXAMPLE

NOTE

Refer to Section 2 for applicability of these conventions to your release of PLZ/SYS and use of library functions.

6.1 Purpose

This section presents an example module written in PLZ/SYS, and shows equivalent modules written in PLZ/ASM for both segmented and nonsegmented Z8000. The PLZ/ASM equivalents conform to PLZ/SYS run-time conventions and can be substituted for the PLZ/SYS module as part of a larger program. This example can be used as a model for writing PLZ/SYS-compatible modules in PLZ/ASM.

The PLZ/SYS version is listed in Example #1. The module declares the procedure Example, whose only statement is a recursive call to itself. This example illustrates the PLZ/SYS parameter passing conventions.

EXAMPLE #1:

```
PLZSYS 3.1
   1        Example MODULE
   2
   3        ! Example PLZ/SYS module demonstrating procedure !
   4        ! calling conventions.                           !
   5
   6        GLOBAL
   7
   8        Example
   9            PROCEDURE (inl:  BYTE;  in2:  ^BYTE)
  10            RETURNS   (outl: BYTE;  out2:  WORD)
  11            LOCAL
  12                locall, local2, local3: BYTE
  13            ENTRY
  14    1           outl, out2 := Example (local2, in2)
  15    2       END Example
  16
  17        END Example

END OF COMPILATION:  0 ERROR(S)      0 WARNING(S)
    0 DATA BYTES    14 Z-CODE BYTES  SYMBOL TABLE  2% FULL
```

```
Z8000ASM  2.0
LOC     OBJ CODE    STMT SOURCE STATEMENT

                       1 Example MODULE
                       2
                       3 ! Example module written in PLZ/ASM !
                       4 ! for the nonsegmented Z8000        !
                       5
                       6 CONSTANT
                       7   ! Offsets from local base !
                       8   out2   := 14
                       9   out1   := 13
                      10   in1    := 11
                      11   in2    :=  8
                      12   local3 :=  6
                      13   local2 :=  5
                      14   local1 :=  4
                      15
                      16 GLOBAL
                      17
0000                  18 Example
                      19   PROCEDURE
                      20   ENTRY
                      21     ! --- Entry Sequence --- !
0000 97F0             22     POP    R0,@R15          ! Pop return address      !
0002 ABF3             23     DEC    R15,#4           ! Allocate local variables!
0004 93F0             24     PUSH   @R15,R0          ! Replace return address  !
0006 93FE             25     PUSH   @R15,R14         ! Save old Local Base     !
0008 A1FE             26     LD     R14,R15          ! Establish new Local Base!
                      27
                      28     ! out1, out2 := Example (local2, in2) !
000A ABF3             29     DEC    R15,#4              ! Allocate return params !
                      30
000C 30E8  0005       31     LDB    RL0,R14(#local2)
0010 93F0             32     PUSH   @R15,R0             ! Push 1st input param !
                      33
0012 53FE  0008       34     PUSH   @R15,in2(R14)       ! Push 2nd input param !
                      35
0016 D00C             36     CALR   Example
                      37
0018 57FE  000C       38     POP    out1-1(R14),@R15    ! Pop 1st return param !
                      39
001C 57FE  000E       40     POP    out2(R14),@R15      ! Pop 2nd return param !
                      41
                      42     ! --- Exit Sequence --- !
0020 97FE             43     POP    R14,@R15         ! Restore old Local Base  !
0022 97F1             44     POP    R1,@R15          ! Pop return address      !
0024 A9F7             45     INC    R15,#8           ! Pop locals & input param!
0026 1E18             46     JP     @R1              ! Resume calling procedure!
0028                  47   END Example
                      48
                      49 END Example
   0 errors
Assembly complete
```

Figure 6-1. Example 2: PLZ/ASM Module for the Nonsegmented S8000

6.2 Nonsegmented Code

An equivalent module written in PLZ/ASM for the nonsegmented Z8000 appears in Example #2 (Figure 6-1).

The entry sequence executed before the body of Example is shown in lines 22 through 26. First, the return address is popped from the stack, where it was deposited by the invoking call instruction. This produces storage for the three local variables to be allocated contiguously with the input parameters by decrementing the Stack Pointer register. The return address is then pushed back on the stack. The value of the Local Base register is preserved on the stack for restoration prior to resumption of the calling procedure. Finally, addressing of local storage and parameters is established by setting the Local Base register to the current Stack Pointer register. Lines 22 through 24 are omitted if no local variables are declared by Example.

Figure 6-2 depicts the displayed run-time stack after the entry sequence for Example has been completed. The Local Base register addresses a word containing the caller's Local Base address. The next word deeper in the stack contains the return address. The local variables begin at offset four from the local base. Although the compiler does not guarantee that local storage is allocated in the order shown, two-byte variables can be packed into one word, as demonstrated by the variables locall and local2.

Parameters passed as input to the routine reside beyond local storage at higher storage addresses. The last parameter declared, in2, is closest to the Local Base since it was pushed last. The parameter inl is padded to word length. All parameters occupy at least one word; byte parameters are extended to word length, with the upper byte undefined. Storage for the result parameters yielded by Example resides beyond the input parameters, at higher storage addresses. The first return parameter declared resides closest to the Local Base, ready to be popped from the stack after Example returns to its caller. Byte return parameters always occupy the low-order (high-address) byte of a word, with a high-order byte of undefined value.

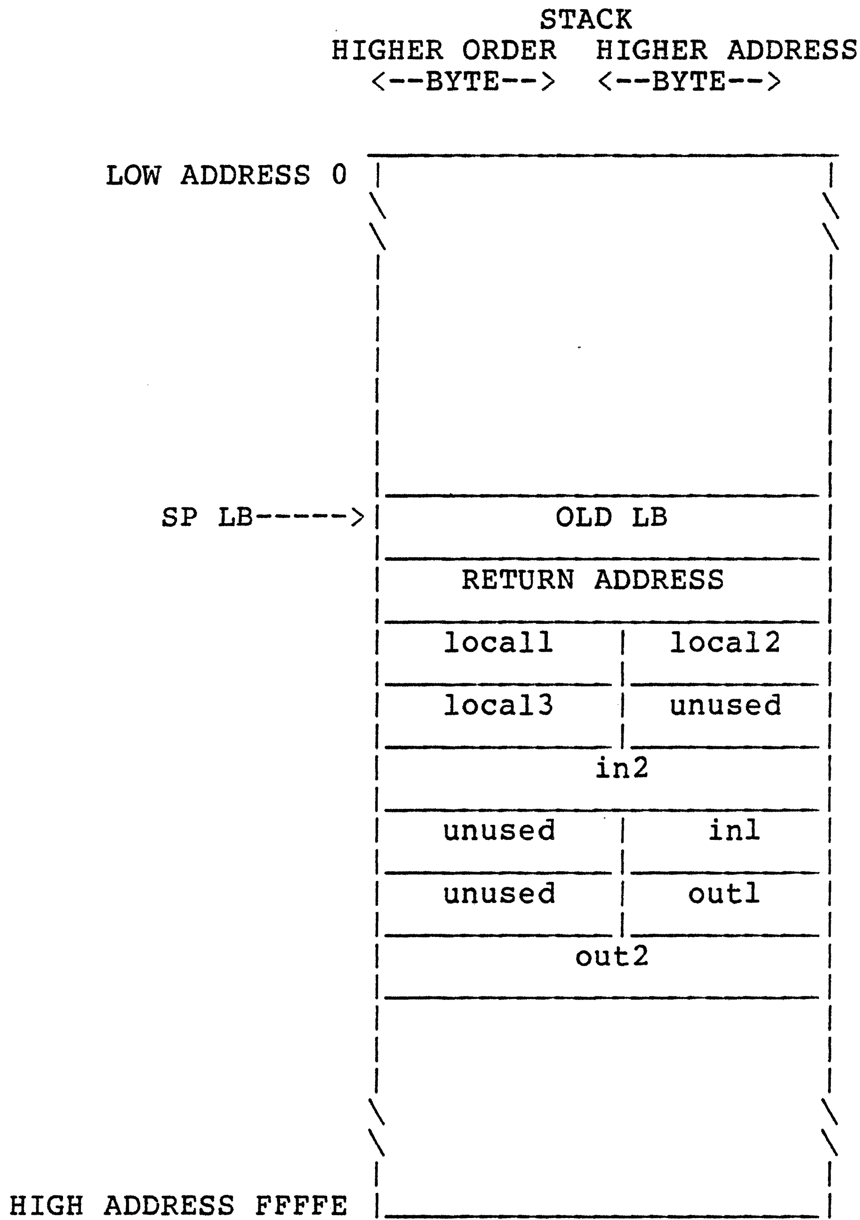


Figure 6-2. Nonsegmented Run-Time Stack Stack Detail After Entry Sequence (Before Line 29 in Example #2)

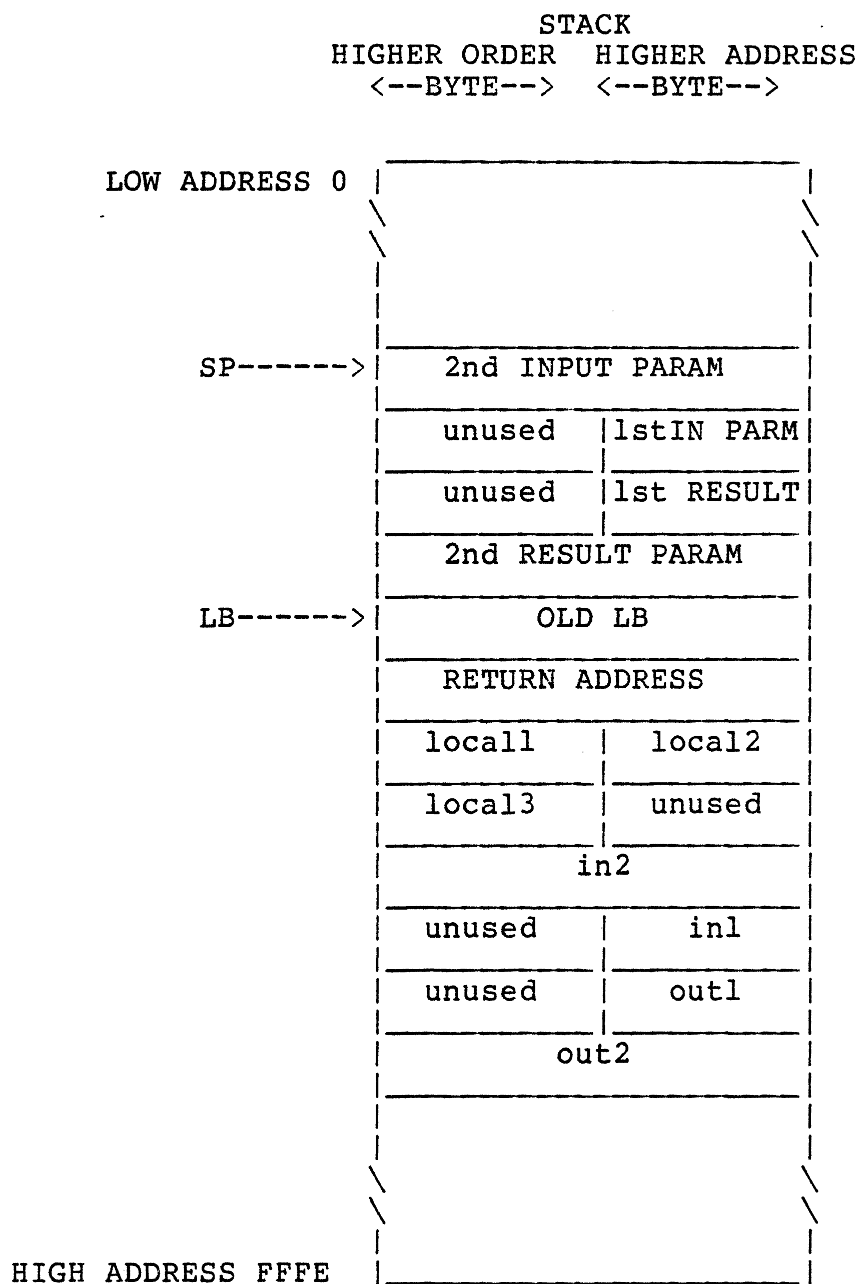


Figure 6-3. Nonsegmented Run-Time Stack Detail Before Recursive Call (Before Line 36 in Example #2)

Lines 29 through 40 demonstrate a typical call to Example. Comparable code is used for any call to Example from other modules. Storage for the two return parameters is allocated by decrementing the Stack Pointer register. Then the input parameters are evaluated and pushed onto the stack in their order of declaration. Figure 6-3 shows the visible portion of the stack prior to executing the call in line 36.

If Example returns from its recursive call, the Stack Pointer register addresses the first return parameter. Popping it from the stack exposes the second return parameter. Popping the second parameter leaves the Stack Pointer register equal to the Local Base register, as it was before line 29 was executed.

The standard procedure exit sequence for nonsegmented code appears in lines 43 through 46. Before line 43 is executed, the stack configuration is the same as it was immediately after execution of the entry sequence. The Stack Pointer and Local Base registers are equal and address the word containing the caller's Local Base address saved during the entry sequence. The Local Base of the calling procedure is popped from the stack into the Local Base register and the return address is popped into a temporary register. Next, the local storage and input parameters are deallocated by incrementing the Stack Pointer. The Stack Pointer register addresses the first return parameter. Execution of the calling procedure is resumed by jumping to the return address. If local variables or input parameters are not declared by Example, lines 44 through 46 are replaced by a return instruction.

6.3 Segmented Code

An equivalent module written in PLZ/ASM for the segmented Z8000 appears in Example #3 (Figure 6-4). In segmented code, parameters and locals are allocated on the Local Stack, while control information resides on the Control Stack. Locals and parameters are stored in the same order, but are accessed relative to the floating Local Pointer rather than to a fixed base address. This requires compensation for movement of the Local Pointer each time local or parameter data is referenced.

The entry sequence in lines 26 and 27 is executed before the body of Example. Line 26 allocates storage for the three local variables on the Local Stack adjacent to the input parameters. Line 27 saves the address of the lowest word of local storage on the Control Stack. This value is addressed by the Control Stack Pointer register during execution of the procedure body and locates the base of local storage for

the procedure. This value is not maintained in a register, since it is not needed for execution, but it is extremely helpful during debugging.

```
Z8000ASM  2.0
LOC     OBJ CODE     STMT SOURCE STATEMENT

                        1 Example MODULE
                        2
                        3 ! Example module written in PLZ/ASM !
                        4 ! for the segmented Z8000.          !
                        5
                        6 $SEGMENTED
                        7
                        8 CONSTANT
                        9   LSseg  := 0   ! Local Stack Segment  !
                       10
                       11   ! Offsets from base of locals !
                       12   out2   := 12
                       13   out1   := 11
                       14   in1    :=  9
                       15   in2    :=  4
                       16   local3 :=  2
                       17   local2 :=  1
                       18   local1 :=  0
                       19
                       20 GLOBAL
                       21
0000                   22 Example
                       23  PROCEDURE
                       24  ENTRY
                       25     ! --- Entry Sequence --- !
0000 ABD3              26     DEC    R13,#4          ! Allocate local variables
0002 91EC              27     PUSHL  @RR14,RR12      ! Save Fixed Base (optional)
                       28
                       29     ! out1, out2 := Example (local2, in2) !
0004 ABD3              30     DEC    R13,#4          ! Allocate return parameters
                       31
0006 30C8  0005        32     LDB    RL0,RR12 (#local2+4)
000A 93C0              33     PUSH   @RR12,R0        ! Push 1st input parameter
                       34
000C 35C0  000A        35     LDL    RR0,RR12(#in2+6)
0010 91C0              36     PUSHL  @RR12,RR0       ! Push 2nd input parameter
                       37
0012 D00A              38     CALR   Example
                       39
0014 57CD  00  0E      40     POP    |<<LSseg>>out1-1+4 | (R13),@RR12 ! 1st result
                       41
0018 57CD  00  0E      42     POP    |<<LSseg>>out2+2 | (R13),@RR12   ! 2nd result
                       43
                       44     ! --- Exit Sequence --- !
001C A9D9              45     INC    R13,#10         ! Pop locals & input params
001E A9F3              46     INC    R15,#4          ! Pop fixed base
0020 9E08              47     RET                    ! Resume calling procedure
0022                   48     END Example
                       49
                       50     END Example
   0 errors
Assembly complete
```

Figure 6-4. Example 3: PLZ/ASM Module for the Segmented S8000

Figure 6-5 displays the portions of the two run-time stacks visible to Example after execution of the entry sequence. Parameters and local variables reside on the Local Stack. Return addresses and Local Base addresses reside on the Control Stack. The size of parameter in2 is four bytes, since segmented addresses occupy long words.

Lines 30 through 42 demonstrate a proper call to Example. The sequence of events is identical to that of nonsegmented code. Parameters are pushed on the Local Stack. The configuration of the run-time stacks before execution of the call instruction in line 38 is shown in Figure 6-6.

If Example returns from its recursive call, the Local Stack Pointer register addresses the first return parameter. Popping it from the Local Stack exposes the second return parameter. Popping the second parameter positions the Local Stack Pointer register at the lowest-address word of local storage.

The exit sequence is shown in lines 45 through 47. Before line 45 is executed, the Control and Local Stack Pointer registers contain the same values they had after the entry sequence. The local variables and input parameters are deallocated by incrementing the Local Stack Pointer register. This leaves the Local Stack Pointer register addressing the first return parameter. The local storage base address is removed from the Control Stack, and the return instruction pops the return address from the Control Stack and resumes the calling procedure. If no local variables or input parameters are declared by Example, line 45 is omitted.

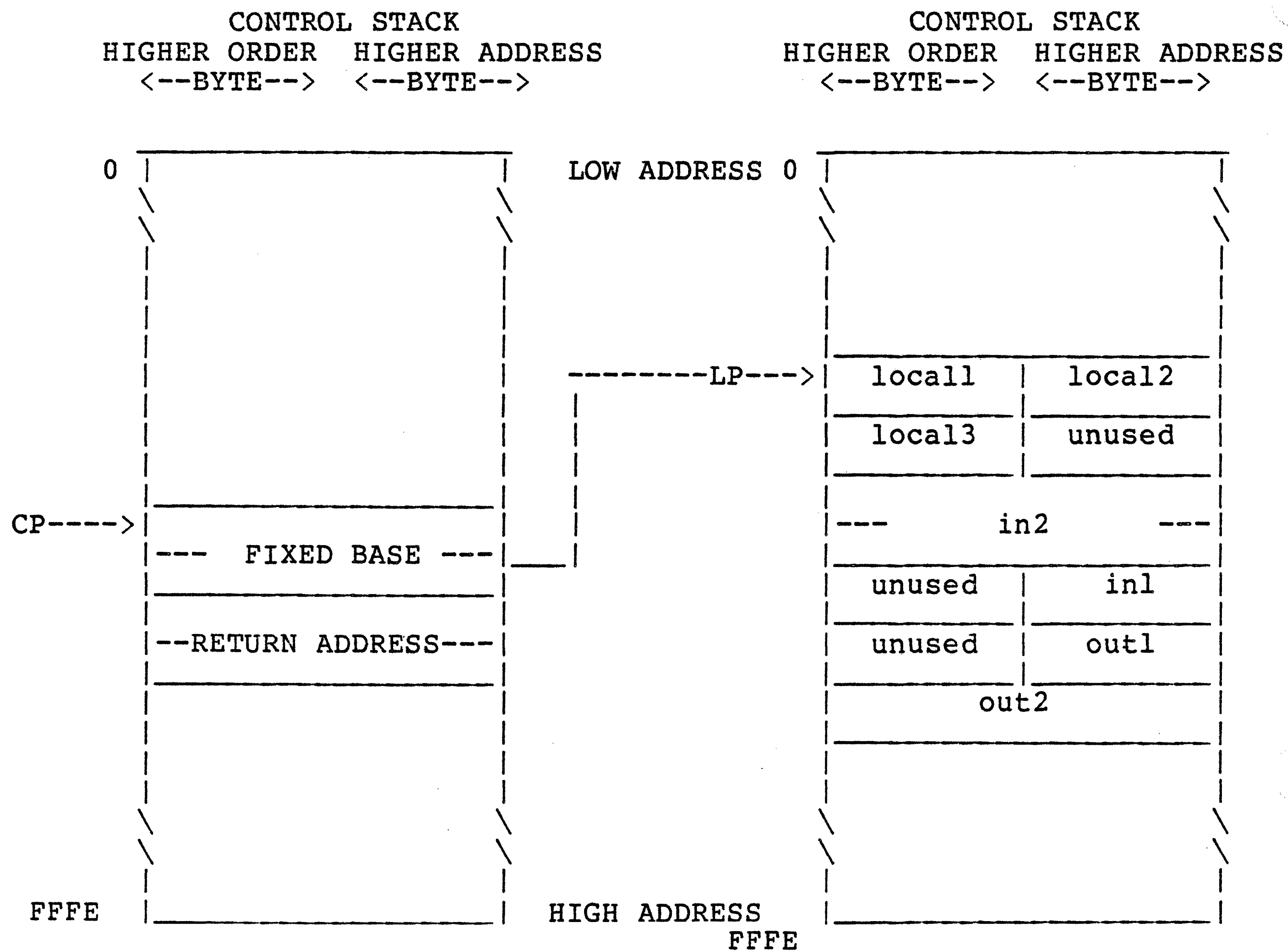


Figure 6-5. Segmented Run-Time Stack Detail After Entry Sequence (Before Line 30 in Example #3)

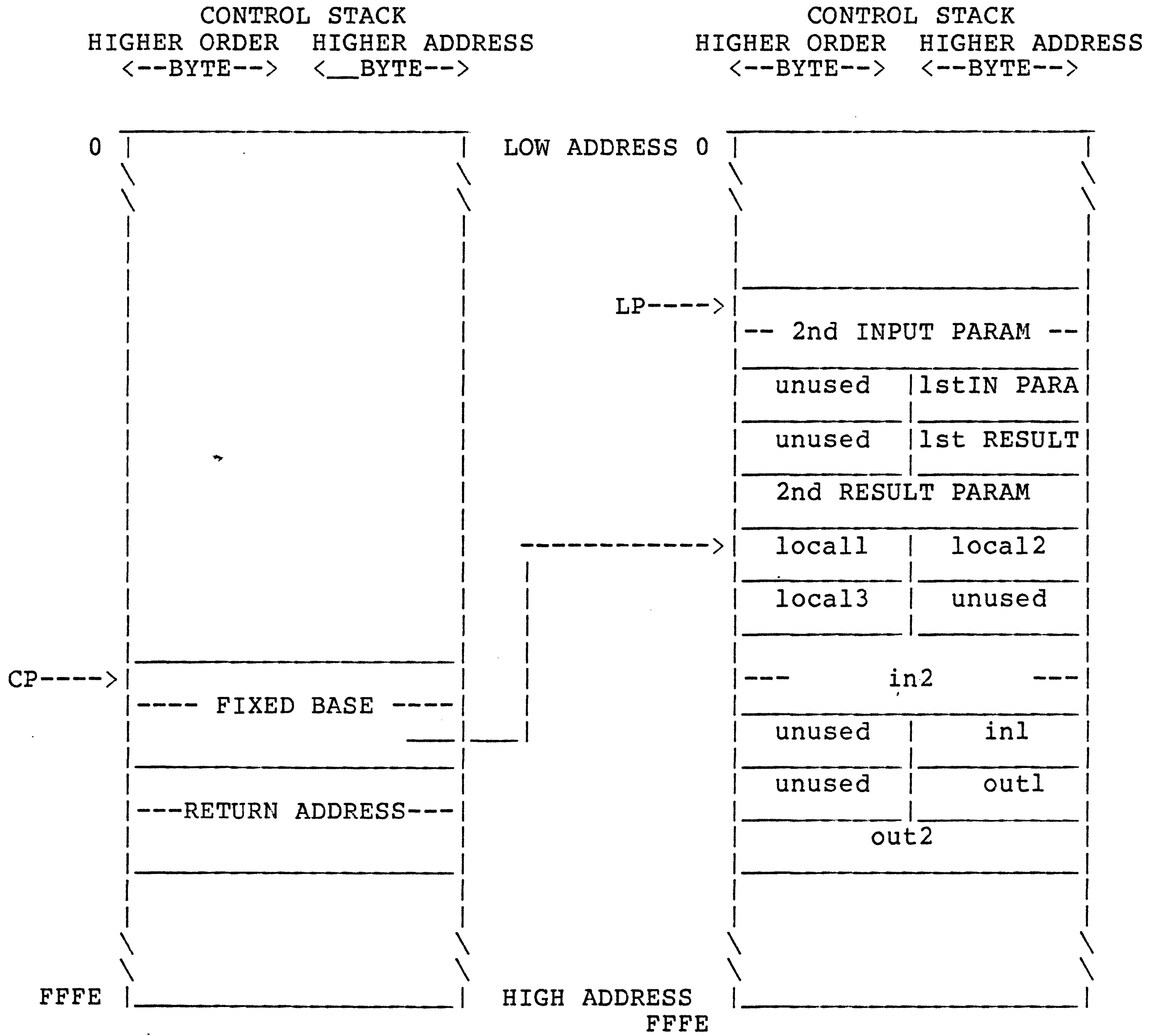


Figure 6-6. Segmented Run-Time Stack Detail Before Recursive Call (Before Line 38 in Example #3)

APPENDIX A

PLZ/SYS ERROR MESSAGES

The error messages in this appendix are shared with the PLZ/ASM assembler. For a complete list of the PLZ/ASM error messages, refer to the System 8000 PLZ/ASM User Guide (Zilog part number 03-3189).

ERROR	EXPLANATION
	Warnings
0	A minus sign (-) or a plus sign (+) treated as binary operator
1	Missing delimiter between tokens
2	Array of zero elements
3	No fields in record declaration
4	Mismatched procedure names
5	Mismatched module names
6	Constant out-of-range for type
8	Absolute address warning for System 8000
	Token Errors
10	Decimal number too large
11	Invalid operator
12	Invalid special character after prompt (%)
13	Invalid hexadecimal digit
14	Character_sequence of zero length
15	Invalid character
16	Hexadecimal number too large
	DO Loop Errors
20	Unmatched OD
21	OD expected
22	Invalid repeat statement
23	Invalid exit statement
24	Invalid FROM label
	IF Statement Errors
30	Unmatched FI
31	FI expected
32	THEN or CASE expected
33	Invalid selector record

ERROR	EXPLANATION
	Symbols Expected
40	) expected
41	(expected
42	] expected
43	[expected
44	:= expected
45	^ expected
	Undefined Names
50	Undefined identifier
51	Undefined procedure name
	Declaration Errors
60	Type identifier expected
61	Invalid module declaration
62	Invalid declaration class
63	Invalid use of array [*] declaration
64	Uninitialized array [*] declaration
65	Invalid dimension size
66	Invalid array component type
67	Invalid record field declaration
68	Invalid type used in pointer declaration
	Procedure Declaration Errors
70	Invalid procedure declaration
71	ENTRY expected
72	Procedure name expected after END
73	Formal parameter name expected
74	Invalid formal parameter type
	Initialization Errors
80	Invalid initial value
81	Too many initialization elements for declared variables
82	Invalid initialization
83	Array [*] gives single noncharacter_sequence initializer
84	Attempt to initialize an uninitialized data area

ERROR	EXPLANATION
	Special Errors
90	Invalid statement
91	Invalid instruction
92	Invalid operand
93	Operand too large
94	Relative address out of range
95	: expected
97	Duplicate record field name
98	Duplicate CASE constant
99	Multiple declaration of identifier
	Invalid Variables
100	Invalid variable
101	Invalid operand for # or SIZEOF
102	Invalid field name
103	Subscripting of nonarray variable
104	Invalid use of period (.)
105	Invalid use of ^
	Expression Errors
110	Invalid arithmetic expression
111	Invalid conditional expression
112	Invalid constant expression
113	Invalid select expression
114	Invalid index expression
115	Invalid expression in assignment
	Constant Out of Bounds
120	Constant too large for 8 bits
121	Constant too large for 16 bits
122	Constant array index out of bounds
	Procedure Call Errors
130	Invalid arithmetic expression
131	Invalid procedure call
132	Procedure call with multiple out parameters expected
133	Too few out parameters
134	Too many out parameters
135	Too few in parameters
136	Too many in parameters

ERROR	EXPLANATION
	Type Incompatibility
140	Character__sequence initializer used with array [*] declaration where component's base type is not 8 bits
141	Type incompatibility with initialization
150	Type incompatibility in arithmetic expression
151	Invalid operand type for unary operator
152	Invalid operand type for binary operator
153	Unassigned type
154	Invalid index type
156	Parameter type incompatible
157	Invalid actual parameter
158	Return parameter type incompatible
159	Return value must be address
160	Type incompatibility in assignment
161	Invalid operand type for relational operator
162	Type incompatibility in conditional expression
163	Invalid type conversion
164	Invalid relational operator for structures
	File Errors
198	EOF expected
199	Unexpected EOF encountered in source--possible unmatched ! or ' in source
	Implementation Restrictions
230	Character__sequence or identifier too long
231	Symbol table overflow
232	Procedure too large
233	Left hand side of assignment too complicated
234	Too many initialization values
235	Stack overflow
236	Too many constants in expression
237	Static data overflow
238	Program area overflow
239	Too many internal or global procedures
240	Long constants not implemented

NOTE

Errors larger than 240 can occur. If there are no other errors in the program preceding "one of these errors, contact Zilog.

APPENDIX B

PLZCG ERROR NUMBERS AND EXPLANATIONS

When the capacity of the code generator's internal tables is exceeded, the code generator aborts with an appropriate error message. This error can usually be corrected by increasing the size of the unallocated memory region, which the code generator uses for these tables. If this is not effective, the source must be modified to reduce its table requirements.

ERROR	EXPLANATION
1	Inappropriate z-code format. The z-code file was probably produced by an outdated version of the PLZ/SYS compiler. Recompile the source module using the companion PLZ/SYS compiler; specify the Z8000 as the target machine.
2	Statement too large
3	Expression too large
4	Procedure call nesting too deep
5	Too many internal and global procedures defined in module
6	Too many alternatives in select statement
7	Procedure too large

NOTE

Error numbers higher than 7 should be reported to Zilog along with any pertinent information concerning their occurrence.

SED

A Noninteractive Text Editor*

* This information is based on an article originally written by Lee E. McMahon, Bell Laboratories.

PREFACE

This document is for users of *sed*, a noninteractive context editor that runs on the ZEUS Operating System. It is assumed that the user has some familiarity with string matching and substitution features of *vi*, the interactive screen-oriented editor of ZEUS.

Section 1 provides an introduction to *sed*. The format of *sed* editing commands appears in Section 2. Section 3 gives the available *sed* commands and use of arguments.

Examples appear throughout the text. Except where otherwise noted, the examples use the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

TABLE OF CONTENTS

1.0 INTRODUCTION 4

2.0 COMMAND OPERATION 5

2.1 General Information 5
2.2 Command Line Flags 5
2.3 Flow of Edit Commands 5
2.4 Pattern Space 6

3.0 LINE SELECTION 7

3.1 Selecting Lines for Editing 7
3.2 Line Number Addresses 7
3.3 Context Addresses 7
3.4 Number of Addresses 9

4.0 FUNCTIONS 10

4.1 General Information 10
4.2 Whole Line Functions 10
4.3 Substitute Functions 12
4.4 Input/Output Functions 15
4.5 Patterns with New Line 16
4.6 Hold and Get Functions 17
4.7 Flow-of-Control Functions 18
4.8 Miscellaneous Functions 19

SECTION 1

INTRODUCTION

Sed is a noninteractive context editor designed for three cases:

1. Editing files too large for efficient interactive editing

2. Editing any size file when the sequence of editing commands is too complicated to be efficiently typed in interactive mode

3. Performing multiple "global" editing functions efficiently in one pass through the input

Sed is a descendant of the editor, *ed*. Because of the differences between interactive and noninteractive operation, considerable changes have been made between *ed* and *sed*. Even experienced users of *ed* will be surprised if they use *sed* without reading Sections 3 and 4 of this document. The most striking resemblance between the two editors is in the class of patterns or regular expressions they recognize. The code for matching patterns is copied almost verbatim from the code for *ed*, and the description of regular expressions in Section 3 is copied almost verbatim from the writeup for *ed* in the *ZEUS Programmer's Manual*.

SECTION 2

COMMAND OPERATION

2.1 General Operation

Sed copies the standard input to the standard output, and can perform one or more editing commands on each line before writing it to the output. This action can be modified by flags on the command line (Section 2.2).

The general format of an editing command is:

[address1,address2][function][arguments]

One or both addresses can be omitted. Any number of blanks or tabs can separate the addresses from the function. The function must be present. The arguments can be required or optional, according to which function is given. Tab characters and spaces at the beginning of lines are ignored.

2.2 Command Line Flags

Three flags are recognized on the command line:

-n: tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (Section 4.4)

-e: tells *sed* to take the next argument as an editing command

-f: tells *sed* to take the next argument as a file name; the file should contain one editing command to a line

2.3 Flow of Edit Commands

For more efficient execution, all the editing commands are first compiled in the order they are encountered. This is generally the order in which they are attempted at execution time. During the execution phase, the commands are applied one at a time, and the input to each command is the output of all preceding commands.

The linear order of application of editing commands can be changed by the flow-of-control commands, *t* and *b* (Section 4.7). Even when the order of application is changed by

these commands, the input line to any command is the output of any previously applied command.

2.4 Pattern Space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the N command (Section 4.5).

SECTION 3

LINE SELECTION

3.1 Selecting Lines for Editing

Lines in an input file can be selected by addresses. Addresses can be either line numbers or context addresses.

The application of a group of commands can be controlled by one address or address-pair by grouping the commands with braces ({ })(Section 4.7).

3.2 Line Number Addresses

A line number is a decimal integer. As each line is read from the input, a line number counter is incremented. A line number address matches the input line, which causes the internal counter to equal the address line number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character $ matches the last line of the last input file.

3.3 Context Addresses

A context address is a pattern "regular expression" enclosed in slashes (/). The following regular expressions are recognized by _sed_:

1. An ordinary character (not one of the special characters discussed in this section) is a regular expression, and matches itself.

2. A circumflex (^) at the beginning of a regular expression matches the null character at the beginning of a line.

3. A dollar sign ($) at the end of a regular expression matches the null character at the end of a line.

4. The characters \n match an embedded new line character, but not the new line at the end of the pattern space.

5. A period (.) matches any character except the terminal new line of the pattern space.

6. A regular expression followed by an asterisk (*) matches any number (including none) of adjacent occurrences of the regular expression it follows.

7. A string of characters in square brackets ([ ]) matches any character in the string, and no others. If the first character of the string is circumflex (^), the regular expression matches any character except the characters in the string and the terminal new line of the pattern space.

8. A concatenation of regular expressions is itself a regular expression. It matches the concatenation of strings that match the components of the regular expression.

9. A regular expression between the sequences \(and \) is identical to the regular expression, but has side-effects described in Section 4.3.

10. The expression \d means the same string of characters matched by an expression enclosed in \(and \) earlier in the same pattern. Here d is a single digit. The string specified begins with the dth occurrence of \(, counting from the left. For example, the expression ^\(.*\)\1 matches a line beginning with two repeated occurrences of the same string.

11. The null regular expression standing alone (for example, //) is equivalent to the last regular expression compiled.

To use one of the special characters:

```
^
$
.
*
[ ]
\
/
```

as a literal to match an occurrence of itself in the input, precede the special character with a backslash (\).

If a context address is to match the input, the whole pattern within the address must match some portion of the pattern space.

3.4 Number of Addresses

The commands in the next section can have zero, one, or two addresses. Two addresses are separated by a comma. Under each command, the maximum number of allowed addresses is given. It is an error for a command to have more addresses than the maximum allowed.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines that match that address.

If a command has two addresses, it is applied to the first line that matches the first address, and to all subsequent lines until and including the first subsequent line that matches the second address. An attempt is made on subsequent lines to again match the first address, and the process is repeated.

Examples:

```
/an/             matches lines 1, 3, 4 in the sample text
/an.*an/         matches line 1
/^an/            matches no lines
/./              matches all lines
/\./             matches line 5
/r*an/           matches lines 1,3, 4 (number = zero!)
/\(an\).*\1/     matches line 1
```

SECTION 4

FUNCTIONS

4.1 General Information

All functions are named by a single character. In this section, the command format shows the maximum number of allowable addresses enclosed in parentheses, the single character function name, and possible arguments enclosed in angle brackets (< >). The angle brackets around the arguments are not part of the argument and must not be typed in actual editing commands. An expanded English translation of the single character name and a description of each function also appear.

4.2 Whole Line Functions

Within the text output by these functions, leading blanks and tabs disappear. To include leading blanks and tabs in the output, precede the first desired blank or tab with a backslash. The backslash does not appear in the output.

(2)d -- delete lines

The d function deletes from the file all those lines matched by its address(es).

It also has the effect that no further commands are attempted on the deleted lines. As soon as the d function is executed, a new line is read from the input, and the list of editing commands is restarted from the beginning on the new line.

(2)n -- next line

The n function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the n command.

(1)a\
<text> -- append lines

The a function writes the argument <text> to the output after the line matched by its address. The a command is inherently multiline; a must appear at the end of a line, and <text> can contain any number of lines. The interior new lines must immediately follow a backslash

character (\). The <text> argument is terminated by the first new line not immediately preceded by a backslash.

Once an _a_ function is successfully executed, <text> is written to the output. The triggering line can be deleted entirely, but <text> is still written to the output.

The <text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line number counter.

(1)i\
<text> -- insert lines

The _i_ function behaves like the _a_ function, except that <text> is written to the output before the matched line. All other comments about the _a_ function apply to the _i_ function.

(2)c\
<text> -- change lines

The _c_ function deletes the lines selected by its address(es) and replaces them with the lines in <text>. Like _a_ and _i_, _c_ must be followed by a new line entered after a backslash. Interior new lines in <text> must follow backslashes.

The _c_ command can have two addresses, and thereby select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output. As with _a_ and _i_, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line number counter.

After a line has been deleted by a _c_ function, no further commands are attempted on it.

If text is appended after a line by _a_ or _r_ functions, and the line is subsequently changed, the text inserted by the _c_ function is placed before the text of the _a_ or _r_ functions.

NOTE

Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in sed commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

Example:

The list of editing commands:

```
n
a\
XXXX
d
```

applied to the standard input produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect is produced by either of the two following command lists:

```
n           n
i\          c\
XXXX        XXXX
d
```

4.3 Substitute Functions

A substitute function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The s function replaces the part of a line selected by <pattern> with <replacement>. It is also read:

Substitute for <pattern>, <replacement>

The <pattern> argument contains a pattern, exactly like the patterns in addresses (Section 3.3). The only

difference between <pattern> and a context address is that the context address must be delimited by slash (/) characters and <pattern> can be delimited by any character other than space or new line.

By default, only the first string matched by <pattern> is replaced.

The <replacement> argument begins immediately after the second delimiting character of <pattern> and must be followed immediately by another instance of the delimiting character. Thus, there are three instances of the delimiting character.

The <replacement> is not a pattern, and the characters that are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

& is replaced by the string matched by <pattern>

\d (where d is a single digit) is replaced by the dth substring matched by parts of <pattern> enclosed in \(and \). If nested substrings occur in <pattern>, the dth string is determined by counting opening delimiters. As in patterns, special characters can be made literal by preceding them with a backslash (\).

The <flags> argument can contain the following flags:

g -- substitute <replacement> for all nonoverlapping instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters. Characters put into the line from <replacement> are not rescanned.

p -- print the line if a successful replacement was done. The p flag prints the line to the output if a substitution was actually made by the s function. If several s functions, each followed by a p flag, successfully substitute in the same input line, multiple copies of the line are written to the output--one for each successful substitution.

w <filename> -- write the line to a file if there was a successful replacement. The w flag causes lines that are actually substituted by the s function to be written to a file named by <filename>. If <filename> exists before sed is run, it is overwritten; if not, it is created.

A single space must separate w and <filename>.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for p.

A maximum of ten different file names can be mentioned after w flags and w functions.

Examples:

Applied to the standard input, the following command,

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubhla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and, on the file "changes":

```
Through caverns measureless by man
Down by a sunless sea.
```

If the nocopy option is in effect, the command:

```
s/[.,;?:]/*P&*/gp
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

In nocopy mode, the command:

```
/X/s/an/AN/p
```

produces

```
In XANadu did Kubhla Khan
```

and the command:

```
/X/s/an/AN/gp
```

produces:

    In XANadu did Kubhla KhAN

4.4 Input/Output Functions

(2)p -- print

The print function writes the addressed lines to the standard output file. They are written at the time the p function is encountered, regardless of what subsequent editing commands do to the lines.

(2)w <filename> -- write on <filename>

The write function writes the addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands do to them.

One space must separate the w and <filename>. A maximum of ten different files can be mentioned in write functions and w flags after s functions.

(1)r <filename> -- read the contents of a file

The read function reads the contents of <filename> and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line that matched its address. If r and a functions are executed on the same line, the text from the a functions and the r functions is written to the output in the order that the functions are executed.

One space must separate the r and <filename>. If a file mentioned by an r function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

Since there is a limit to the number of files that can be opened simultaneously, take care not to mention more than ten files in w functions or flags. The number is reduced to nine if any r functions are present.

Examples:

Assume that the file notel has the following contents:

> Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

The following command:

```
/Kubla/r notel
```

produces:

```
In Xanadu did Kubla Khan
     Note: Kubla Khan (more properly Kublai Khan;
     1216-1294) was the grandson and most eminent suc-
     cessor of Genghiz (Chingiz) Khan, and founder of
     the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

4.5 Patterns with New Line

Three functions, all entered as capital letters, deal specifically with pattern spaces containing embedded new lines. They provide pattern matches across lines in the input.

(2)N -- Next line

> The next input line is appended to the current line in the pattern space and the two input lines are separated by an embedded new line. Pattern matches can extend across the embedded new line(s).

(2)D -- Delete first part of the pattern space

> Delete up to and including the first new line character in the current pattern space. If the pattern space becomes empty (the only new line is the terminal new line), read another line from the input. Begin the list of editing commands again from its beginning.

(2)P -- Print first part of the pattern space

Print up to and including the first new line in the pattern space.

The P and D functions are equivalent to their lowercase counterparts if there are no embedded new lines in the pattern space.

4.6 Hold and Get Functions

Four functions save and retrieve part of the input for later use.

(2)h -- hold pattern space

The h function copies the contents of the pattern space into a hold area, destroying the previous contents of the hold area.

(2)H -- Hold pattern space

The H function appends the contents of the pattern space to the contents of the hold area. The former and new contents are separated by a new line.

(2)g -- get contents of hold area

The g function copies the contents of the hold area into the pattern space, destroying the previous contents of the pattern space.

(2)G -- Get contents of hold area

The G function appends the contents of the hold area to the contents of the pattern space. The former and new contents are separated by a new line.

(2)x -- exchange

The exchange command interchanges the contents of the pattern space and the hold area.

Example:

The commands

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

applied to the standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

4.7 Flow-of-Control Functions

These functions control the application of functions to the lines selected by the address portion. They do no editing on the input lines.

(2)! -- Don't

The Don't command causes the next command written on the same line to be applied to input lines not selected by the address part.

(2){ -- Grouping

The grouping command, a left brace ({), causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping command can appear on the same line as the {, or on the next line.

The group of commands is terminated by a right brace (}) standing on a line by itself.

Groups can be nested.

(0):<label> -- place a label

The label function marks a place in the list of editing commands that can be referred to by b and t functions. The <label> can be any sequence of eight or fewer characters. If two different colon functions have identical labels, a compile-time diagnostic is generated, and no execution is attempted.

(2)b<label> -- branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after a colon function with the same <label> is encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile-time diagnostic is

produced, and no execution is attempted.

A _b_ function with no <label> is a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read. The list of editing commands is restarted from the beginning on the new line.

(2)t<label> -- test substitutions

The _t_ function tests whether any successful substitutions have been made on the current input line. If so, it branches to <label>; if not, it does nothing. The flag indicating that a successful substitution has been executed is reset by:

1. reading a new input line, or
2. executing a _t_ function.

4.8 Miscellaneous Functions

(1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q -- quit

The _q_ function writes the current line to the output, writes any appended or read text, and terminates execution.

An Introduction to the ZEUS Shell*

* This information is based on an article originally written by S.R. Bourne, Bell Laboratories.

PREFACE

The shell is both a command language and a programming language that provides an interface to the ZEUS Operating System. This document describes, with examples, the ZEUS shell.

This version of the shell is sometimes referred to as the "Bourne Shell" after its original author, S. R. Bourne of Bell Laboratories. There are at least four other shell programs in moderately widespread use. Users can select the shell they feel most comfortable with. Most of Zilog's internal users use the C Shell (see), which has additional features for interactive work.

The first section covers most requirements of terminal users. Some familiarity with ZEUS is an advantage when reading this section. _ZEUS for Beginners_ in this manual provides the basis for this familiarity.

Section 2 describes features of the shell primarily intended for use within shell procedures. These include control-flow primitives and string-valued variables. Knowledge of a programming language is helpful when reading this section.

The last section describes more advanced features of the shell. References of the form "_pipe_ (2)" refer to a section of the _ZEUS Reference Manual_.

TABLE OF CONTENTS

SECTION 1 BASIC TASKS 5

1.1 Introduction 5
1.2 Simple Commands 5
1.3 Background Commands 5
1.4 Input/Output Redirection 6
1.5 Pipelines and Filters 6
1.6 File Name Generation 7
1.7 Quoting 9
1.8 Prompting 9
1.9 The Shell and Login 10
1.10 Summary 10

SECTION 2 SHELL PROCEDURES 11

2.1 Introduction 11
2.2 Control Flow--For 12
2.3 Control Flow--Case 13
2.4 Here Documents............................. 15
2.5 Shell Variables 16
2.6 Test Command 19
2.7 Control Flow--While 19
2.8 Control Flow--If 20
2.9 Command Grouping 22
2.10 Debugging Shell Procedures 22
2.11 The man Command 23

SECTION 3 KEYWORD PARAMETERS 25

3.1 Introduction 25
3.2 Parameter Transmission 25
3.3 Parameter Substitution 26
3.4 Command Substitution 27
3.5 Evaluation and Quotation 28
3.6 Error Handling 31
3.7 Fault Handling 33
3.8 Command Execution 35
3.9 Invoking the Shell 37

APPENDIX A GRAMMAR 38

APPENDIX B METACHARACTERS AND RESERVED WORDS 40

LIST OF ILLUSTRATIONS

Figure

2-1 A Version of the man Command 24

3-1 ZEUS Signals 32
3-2 The touch Command 34
3-3 The scan Command 34

SECTION 1

BASIC TASKS

1.1 Introduction

The shell is a command programming language that provides an interface to the ZEUS Operating System. Its features include control-flow primitives, parameter passing, variables, and string substitution. Constructs such as while, if-then-else, case, and for are available. Two-way communication is possible between the shell and commands. String-valued parameters, typically file names or flags, can be passed to a command. A return code that is set by commands can be used to determine control flow, and the standard output from a command can be used as shell input.

The shell modifies the environment in which commands run. Input and output can be redirected to files, and processes that communicate through pipes can be invoked. Commands are found by searching directories in the file. Commands can be read either from the terminal or from a file.

1.2 Simple Commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

```
ls -l
```

prints a list of files in the current directory. The argument -l tells ls to print status information, size, and the creation date for each file.

1.3 Background Commands

To execute a command, the shell normally creates a new process and waits for it to finish. A command can also be run in the background, that is, without waiting for the process to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing & is an operator that instructs the shell to run the command in the background. To help keep track of such a process, the shell reports its process number following its creation. A list of currently active processes can be obtained using the *ps* command.

1.4 Input/Output Redirection

Most commands produce output on the standard output device (the terminal). This output can also be sent to a file by writing, for example,

```
ls -l >file
```

The notation >*file* is interpreted by the shell and is not passed as an argument to *ls*. If *file* does not exist, the shell creates it; otherwise, the original contents of *file* are replaced with the output from *ls*. Output can be appended to a file using the notation

```
ls -l >>file
```

The standard input of a command can be taken from a file instead of the terminal by entering

```
wc <file
```

The command *wc* reads its standard input (in this case redirected from *file*) and prints the number of characters, words, and lines found. If only the number of lines is required,

```
wc -l <file
```

is used.

1.5 Pipelines and Filters

The standard output of one command can be connected to the standard input of another by entering the pipe operator (|), as in,

```
ls -l | wc
```

Two commands connected in this way constitute a pipeline and the overall effect is the same as

```
ls -l >file; wc <file
```

except that no file is used. Instead, the two processes are connected by a *pipe* (2) and are run in parallel. Pipes are unidirectional. Synchronization is achieved by halting *wc* when there is nothing to read and halting *ls* when the pipe is full.

A filter is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, *grep*, selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines of the output from *ls* that contain the string *old*. Another useful filter is *sort*. For example,

```
who -| sort
```

prints an alphabetically sorted list of logged-in users.

A pipeline can consist of more than two commands. For example,

```
ls | grep old | wc -l
```

prints the number of file names in the current directory containing the string *old*.

1.6 File Name Generation

Many commands accept arguments that are file names. For example,

```
ls -l main.c
```

prints information relating to the file *main.c*.

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates, as arguments to *ls*, all file names in the current directory that end in *.c*. The character * is a pattern that matches any string including the null string. Patterns are specified as follows:

- `*` matches any string of characters including the null string
- `?` matches any single character
- `[...]` matches any one of the characters enclosed; a pair of characters separated by a minus matches any character lexically between the pair

For example,

    [a-z]*

matches all names in the current directory beginning with one of the letters _a_ through _z_.

    /usr/fred/test/?

matches all names in the directory /usr/fred/test that consist of a single character. If no file name is found that matches the pattern, the pattern is passed unchanged as an argument.

This mechanism saves typing, selects names according to some pattern, and finds files. For example,

    echo /usr/fred/*/core

finds and prints the names of all core files in subdirectories of /usr/fred. (_Echo_ is a standard ZEUS command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all subdirectories of /usr/fred.

There is one exception to the general rules given for patterns. A single period (.) at the start of a file name must be explicitly matched. For example,

    echo *

echoes all file names in the current directory not beginning with . .

    echo .*

echoes all those file names that begin with . . This avoids matching the name . (the current directory) with .. (the parent directory). The _ls_ command suppresses information for the . and .. files.

1.7 Quoting

Characters that have a special meaning to the shell, such as <, >, *, ?, |, &, and , are called metacharacters. (A complete list of metacharacters is given in Appendix B.) Any character preceded by a backslash (\) is quoted and loses its special meaning. The \ itself is not echoed, so

```
echo \?
```

echoes a single ? and

```
echo \\
```

echoes a single \. To allow long strings to be continued over more than one line, the sequence \new line is ignored.

The \ is convenient for quoting single characters, but clumsy when more than one character needs quoting. A string of characters can be quoted by enclosing the string between single quotes. For example,

```
echo xx'****'xx
```

echoes

```
xx****xx
```

The quoted string can contain new lines, which are preserved; it cannot contain a single quote. This quoting mechanism is the simplest and is recommended.

A third quoting mechanism uses double quotes (Section 3.5).

1.8 Prompting

When the shell is used from a terminal, it issues a prompt before reading a command. By default, this prompt is a dollar sign ($); it can be changed by the PS1 command. For example,

```
PS1=yesdear
```

sets the prompt to be the string <u>yesdear</u>.

If a new line is typed and further input is needed, the shell issues the > prompt. Sometimes this can be caused by mistyping a quote mark. If it is unexpected, an interrupt

returns the shell to read another command. This prompt can be changed by the PS2 command. For example,

```
PS2=more
```

1.9 The Shell and Login

Following _login_ (1), the shell is called to read and execute commands typed at the terminal. If the user's login directory contains the file .profile, it is assumed to contain commands and is read by the shell before any commands are read from the terminal.

1.10 Summary

ls
Print the names of files in the current directory.

ls >file
Put the output from _ls_ into _file_.

ls | wc -l
Print the number of files in the current directory.

ls | grep old
Print those file names containing the string _old_.

ls | grep old | wc -l
Print the number of files whose names contain the string _old_.

cc pgm.c &
Run _cc_ in the background.

SECTION 2

SHELL PROCEDURES

2.1 Introduction

The shell reads and executes commands contained in a file. For example,

```
sh file [ args... ]
```

calls the shell to read commands from _file_. Such a file is called a command procedure or shell procedure. Arguments can be supplied with the call and are referred to in _file_ using the positional parameters such as $1. For example, if the file _wg_ contains

```
who | grep $1
```

then

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

ZEUS files have three independent attributes: read, write, and execute. The ZEUS command _chmod_ (1) can be used to make a file executable. For example,

```
chmod +x wg
```

ensures that the file _wg_ has execute status. Following this, the command

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case, a new process is created to run the command.

In addition to providing names for the positional parameters, the number of positional parameters in the call is available as $#. The name of the file being executed is available as $0.

A special shell parameter, $*, substitutes for all positional parameters except $0. This provides some default arguments, as in,

```
nroff -T450 -ms $*
```

which prepends some arguments to those already given.

2.2 Control Flow--For

A frequent use of shell procedures is to loop through the arguments ($1, $2...), executing commands once for each argument.

An example of such a procedure is _tel_, which searches the file /usr/lib/telnos, which contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of _tel_ is

```
for i
do grep $i /usr/lib/telnos; done
```

The command

```
tel fred
```

prints those lines in /usr/lib/telnos that contain the string _fred_.

```
tel fred bert
```

prints those lines containing _fred_ followed by those for _bert_.

The _for_ loop notation is recognized by the shell and has the general form

```
for name in w1 w2 ...
do command-list
done
```

A _command-list_ is a sequence of one or more simple commands separated or terminated by a new line or semicolon. Reserved words like _do_ and _done_ are only recognized following a new line or semicolon. _Name_ is a shell variable that is set to the words _w1 w2_... in turn each time the _command-_

list following *do* is executed. If *in w1 w2...* is omitted, the loop is executed once for each positional parameter, that is, "in $*" is assumed.

Another example of the loop is the *create* command, for which the text is

```
for i do >$i; done
```

The command

```
create alpha beta
```

ensures that two files, *alpha* and *beta*, exist and are empty. The notation >*file* can be used on its own to create or clear the contents of a file. A semicolon (or a new line) is required before *done*.

2.3 Control Flow--Case

The case notation provides a multiple branch. For example,

```
case $# in
     1) cat >> $1 ;;
     2) cat >> $2 <$1 ;;
     *) echo 'usage: append [ from ] to' ;;
esac
```

is an *append* command. When called with one argument as

```
append file
```

$# is the string *1* and the standard input is copied onto the end of *file* using the *cat* command. The command

```
append file1 file2
```

appends the contents of *file1* to *file2*. If more than two arguments are supplied to *append*, a message is printed indicating improper usage.

The general form of the case command is

```
case word in
     pattern) command-list;;
     ...
esac
```

The shell attempts to match *word* with each *pattern* in the order in which the patterns appear. If a match is found, the associated *command-list* is executed, and execution of

the case is complete. Since * is the pattern that matches any string, it can be used for the default case.

No check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the next example, the commands following the second * are never executed.

```
case $# in
     *) ... ;;
     *) ... ;;
esac
```

Another example of the case construction is distinguishing between different forms of an argument. The following example is a fragment of a <u>cc</u> command.

```
for i
do case $i in
     -[ocs])  ... ;;
     -*)  echo 'unknown flag $i' ;;
     *.c) /lib/c0 $i ... ;;
     *)   echo 'unexpected argument $i' ;;
     esac
done
```

To allow the same commands to be associated with more than one pattern, the case command provides for alternative patterns separated by a |. For example,

```
case $i in
     -x | -y) ...
esac
```

is equivalent to

```
case $i in
     -[xy]) ...
esac
```

The usual quoting conventions apply so that

```
case $i in
     \?)  ...
```

matches the ? character.

2.4 Here Documents

The shell procedure _tel_ in Section 2.2 uses the file /usr/lib/telnos to supply the data for _grep_. An alternative includes this data within the shell procedure as a _here_ document, as in,

```
for i
do grep $i << !
     ...
     fred mh0123
     bert mh0789
     ...
!
done
```

In this example, the shell takes the lines between <<! and ! as the standard input for _grep_. The string ! is arbitrary; the document is terminated by a line that consists of the string following <<, whatever that is.

Parameters are substituted in the document before it is made available to _grep_, as illustrated by the following procedure called _edg_.

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of _string1_ in _file_ to _string2_. Substitution is prevented if \ is used to quote the special character $, as in

```
ed $3 <<+
1,\$s/$1/$2/g
w
+
```

This version of *edg* is equivalent to the first except that *ed* prints a ? if there are no occurrences of the string $1. Substitution within a *here* document is prevented entirely by quoting the terminating string. For example,

```
grep $i <<\#
...
#
```

The document is presented without modification to *grep*. If parameter substitution is not required in a *here* document, this latter form is more efficient.

2.5 Shell Variables

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. Variables can be given values with commands such as

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables user, box, and acct. A variable can be set to the null string by entering, for example,

```
null=
```

The value of a variable is substituted by preceding its name with $; for example,

```
echo $user
```

echoes *fred*.

Variables can be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv pgm $b
```

moves the file *pgm* from the current directory to the directory /usr/fred/bin. A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

directs the output of _ps_ to the file /tmp/psa, whereas,

```
ps a >$tmpa
```

substitutes the value of the variable _tmpa_.

Except for $?, the following are set initially by the shell. $? is set after executing each command.

$? The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise, a nonzero exit status is returned. Testing the value of return codes is dealt with under _if_ and _while_ commands.

$# The number of positional parameters in decimal. Used, for example, in the _append_ command to check the number of parameters.

$$ The process number of this shell in decimal. Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,

```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```

$! The process number of the last process run in the background (in decimal).

$- The current shell flags, such as -x and -v.

The following variables have a special meaning to the shell and must be avoided for general use.

$MAIL When used interactively, the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at, the shell prints the message _you have mail_ before prompting for the next command. This variable

is typically set in the file .profile, in the user's login directory. For example,

```
MAIL=/usr/mail/fred
```

$HOME The default argument for the _cd_ command. The current directory resolves file name references that do not begin with a /, and is changed using the _cd_ command. For example,

```
cd /usr/fred/bin
```

makes the current directory /usr/fred/bin.

```
cat wn
```

prints on the terminal the file _wn_ in this directory. The command _cd_ with no argument is equivalent to

```
cd $HOME
```

This variable is set in the user's login profile.

$PATH A list of directories that contain commands (the _search path_). Each time a command is executed by the shell, a list of directories is searched for an executable file. If $PATH is not set, the current directory, /bin, and /usr/bin are searched by default. Otherwise, $PATH consists of directory names separated by :. For example,

```
PATH=:/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first :) /usr/fred/bin, /bin, and /usr/bin, are to be searched, in that order. Individual users can have their own private commands that are accessible independently of the current directory. If the command name contains a /, this directory search is not used. A single attempt is made to execute the command.

$PS1 The primary shell prompt string, by default, $.

$PS2 The shell prompt when further input is needed, by default, >.

$IFS The set of characters used by blank interpretation (Section 3.5).

2.6 Test Command

The *test* command, although not part of the shell, is used by shell programs. For example,

```
test -f file
```

returns zero exit status if *file* exists and nonzero exit status otherwise. In general, *test* evaluates a predicate and returns the result as its exit status. Some of the more frequently used *test* arguments are given here. (*Test* (1) contains a complete specification.)

test s	true if the argument *s* is not the null string
test -f file	true if *file* exists
test -r file	true if *file* is readable
test -w file	true if *file* is writable
test -d file	true if *file* is a directory

2.7 Control Flow--While

The actions of the for loop and the case branch are determined by data available to the shell. A *while* or *until* loop and an *if then else* branch are also provided; their actions are determined by the exit status returned by commands. A *while* loop has the general form

```
while command-list1
do command-list2
done
```

The value tested by the while command is the exit status of the last simple command following while. Each time around the loop, *command-list* is executed. If a zero exit status is returned, *command-list* is executed; otherwise, the loop terminates. For example,

```
while test $1
do ...
     shift
done
```

is equivalent to

```
for i
do ...
done
```

Shift is a shell command that renames the positional parameters $2, $3... as $1, $2... and loses $1.

Another use for the while/until loop is to wait until some external event occurs and then run some commands. In an *until* loop, the termination condition is reversed. For example,

```
until test -f file
do sleep 300; done
commands
```

loops until *file* exists. Each time around the loop, it waits for five minutes before trying again.

2.8 Control Flow--If

Also available is a general conditional branch of the form,

```
if command-list
then command-list
else command-list
fi
```

which tests the value returned by the last simple command following if.

The *if* command can be used in conjunction with the *test* command to test for the existence of a file as in

```
if test -f file
then process file
else do something else
fi
```

A multiple test *if* command of the form

```
if ...
then ...
else if ...
     then ...
     else if ...
          ...
          fi
     fi
fi
```

can be written using an extension of the *if* notation as,

```
if ...
then ...
elif ...
then ...
elif ...
...
fi
```

The following example is the *touch* command, which changes the "last modified" time for a list of files. The command can be used in conjunction with *make* (1) to force recompilation of a list of files.

```
flag=
for i
do case $i in
     -c)  flag=N ;;
      *)  if test -f $i
          then ln $i junk$$; rm junk$$
          elif test $flag
          then echo file \'$i\' does not exist
          else >$i
          fi
     esac
done
```

The -c flag in this command forces subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable *flag* is set to some non-null string if the -c argument is encountered. The commands

```
ln ...; rm ...
```

make a link to the file and then remove it, thus causing the last modified date to be updated.

The sequence

```
if command1
then command2
fi
```

can be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes *command2* only if *command1* fails. In each case, the value returned is that of the last simple command executed.

2.9 Command Grouping

Commands can be grouped in two ways,

{ *command-list* ; }

and

(*command-list*)

In the first form, *command-list* is simply executed. The second form executes *command-list* as a separate process. For example,

```
(cd x; rm junk )
```

executes *rm junk* in the directory x without changing the current directory of the invoking shell.

The commands

```
cd x; rm junk
```

have the same effect, but leave the invoking shell in directory x.

2.10 Debugging Shell Procedures

The shell provides two tracing mechanisms to help debug shell procedures. The first is invoked within the procedure, as with

```
set -v  (v for verbose)
```

and causes lines of the procedure to be printed as they are read. This is useful to help isolate syntax errors. It can be invoked without modifying the procedure by using

```
sh -v proc ...
```

where *proc* is the name of the shell procedure. This flag can be used in conjunction with the -n flag, which prevents execution of subsequent commands. (Using *set -n* at a terminal renders the terminal useless until an end-of-file is typed.)

The command

```
set -x
```

produces an execution trace. Following parameter substitution, each command is printed as it is executed. Both flags can be turned off by entering

```
set -
```

The current setting of the shell flags is available as $-.

2.11 The *man* Command

The *man* command can be used used to print sections of a document. It is called, for example, as

```
man sh
man -t ed
man 2 fork
```

In the first line, a section of the *sh* manual is printed. Since no section is specified, Section 1 is used. The second example typesets (-t option) a section of the manual *ed*. The last prints the *fork* manual page from Section 2.

A more elaborate example of the *man* command appears in Figure 2-1.

```
cd /usr/man

: 'colon is the comment command'
: 'default is nroff ($N), section 1 ($s)'
N=n s=1

for i
do case $i in

     [1-9*]    s=$i ;;

     -t)  N=t ;;

     -n)  N=n ;;

     -*)  echo unknown flag \'$i\' ;;

      *)  if test -f man$s/$i.$s
          then ${N}roff man0/${N}aa man$s/$i.$s
          else : 'look through all manual sections'
               found=no
               for j in 1 2 3 4 5 6 7 8 9
               do if test -f man$j/$i.$j
                    then man $j $i
                         found=yes
                    fi
               done
               case $found in
                    no) echo '$i: manual page not found'
          esac
          fi
     esac
done
```

Figure 2-1. A Version of the man Command

SECTION 3

KEYWORD PARAMETERS

3.1 Introduction

Shell variables are given values by assignment or by invoking a shell procedure. An argument to a shell procedure of the form _name_=_value_ that precedes the command name causes _value_ to be assigned to _name_ before execution of the procedure begins. The value of _name_ in the invoking shell is not affected. For example,

```
user=fred command
```

executes _command_ with _user_ set to _fred_. The -k flag causes arguments of the form _name_=_value_ to be interpreted in this way anywhere in the argument list. Such names are called keyword parameters. If any arguments remain, they are available as positional parameters $1, $2, and so on.

The _set_ command can also be used to set positional parameters from within a procedure. For example,

```
set - *
```

sets $1 to the first file name in the current directory, $2 to the next, and so on. The first argument (-) ensures correct treatment when the first file name begins with a -.

3.2 Parameter Transmission

When a shell procedure is invoked, both positional and keyword parameters can be supplied with the call. Keyword parameters are implicitly available to a shell procedure by specifying in advance that such parameters are to be exported. For example, the command

```
export user box
```

marks the variables user and box for export. When a shell procedure is invoked, copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. A shell procedure cannot modify the state of its caller without explicit request on the part of the caller. Shared file descriptors are an exception to this rule.

Names whose value is intended to remain constant can be declared *readonly*. The form of this command is the same as that of the *export* command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

3.3 Parameter Substitution

If a shell parameter is not set, the null string is substituted for it. For example, if the variable d is not set

```
echo $d
```

or

```
echo ${d}
```

echoes nothing. A default string can be given as in

```
echo ${d-.}
```

which echoes the value of the variable d if it is set and "." otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo ${d-'*'}
```

echoes * if the variable d is not set. Similarly,

```
echo ${d-$1}
```

echoes the value of d if it is set and the value (if any) of $1 otherwise. A variable can be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d-.}
```

and if d was not previously set, then it is set to the string . . The notation ${...=...} is not available for positional parameters.

If there is no default, the notation

```
echo ${d?message}
```

echoes the value of the variable d if it has one; otherwise, *message* is printed by the shell, and execution of the shell procedure is abandoned. If *message* is absent, a standard message is printed. An example of a shell procedure that requires some parameters to be set starts as follows:

```
: ${user?} ${acct?} ${bin?}
...
```

Colon (:) is a command built into the shell and does nothing once its arguments have been evaluated. If any of the variables *user*, *acct*, or *bin* are not set, the shell abandons execution of the procedure.

3.4 Command Substitution

The standard output from a command can be substituted in a manner similar to parameter substitution. The command *pwd* prints on its standard output the name of the current directory. For example, if the current directory is /usr/fred/bin, the command

```
d= `pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between grave accents is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions, except that a ` must be escaped using a \. For example,

```
ls ` echo "$1"`
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs, including *here* documents, and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within shell procedures. An example of such a command is *basename*, which removes a specified suffix from a string. For example,

```
basename main.c
```

prints the string *main*. Its use is illustrated by the

following fragment from a cc command.

```
case $A in
     ...
     *.c)     B= `basename $A .c`
     ...
esac
```

Here, B is set to the part of $A with the suffix .c stripped.

Here are some composite examples:

- for i in `ls -t`; do ...
  The variable i is set to the names of files in time order, most recent first.

- set `date`; echo $6 $2 $3, $4
  prints, for example, 1981 Nov 1, 23:59:59

3.5 Evaluation and Quotation

The shell is a macroprocessor that provides parameter substitution, command substitution, and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in Appendix A. Before a command is executed, the following substitutions occur:

- parameter substitution; for example, $user

- command substitution; for example, `pwd`

  Only one evaluation occurs, so that if the value of the variable X is the string $y, then

  ```
  echo $X
  ```

  echoes $y.

- blank interpretation

  Following the above substitutions, the resulting characters are broken into nonblank words (blank interpretation). For this purpose "blanks" are the characters of the string $IFS. By default, this string consists of blank, tab, and new line. The null string is not regarded as a word unless

it is quoted. For example,

```
echo "
```

passes the null string as the first argument to *echo*, whereas

```
echo $null
```

calls *echo* with no arguments if the variable null is not set or set to the null string.

- file name generation

  Each word is then scanned for the file pattern characters comma, question mark, and [...], and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a *for* loop. Substitution occurs in the *word* used for a case branch.

In addition to the quoting mechanisms described previously, a third quoting mechanism is provided that uses double quotes. Within double quotes, parameter and command substitution occurs, but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and are quoted using \.

$	parameter substitution
`	command substitution
"	ends the quoted string
\	quotes the special characters $ ` " \

For example,

```
echo "$x"
```

passes the value of the variable x as a single argument to *echo*. Similarly,

```
echo "$*"
```

passes the positional parameters as a single argument and is equivalent to

```
echo "$1 $2 ..."
```

The notation $@ is the same as $* except when it is quoted.

The command

```
echo "$@"
```

passes the positional parameters, unevaluated, to *echo* and is equivalent to

```
echo "$1" "$2" ...
```

The following chart gives, for each quoting mechanism, the shell metacharacters that are evaluated.

metacharacter

	\	$	*	`	"	'
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

t terminator
y interpreted
n not interpreted

In cases where more than one evaluation of a string is required, the built-in command *eval* is used. For example, if the variable X has the value $*y*, and if y has the value *pqr*, then

```
eval echo $X
```

echoes the string *pqr*.

The *eval* command evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed. For example,

```
wg='eval who | grep'
$wg fred
```

is equivalent to

```
who | grep fred
```

In this example, *eval* is required since there is no interpretation of metacharacters, such as |, following substitution.

3.6 Error Handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal (as determined by gtty (2)). A shell invoked with the -i flag is also interactive.

Execution of a command (Section 3.8) can fail for any of the following reasons:

- Input/output redirection fails, for example, if a file does not exist or cannot be created
- The command itself does not exist or cannot be executed
- The command terminates abnormally, for example, with a "bus error" or "memory fault" (see Figure 3-1 for a complete list of ZEUS signals)
- The command terminates normally but returns a nonzero exit status

In all of these cases, the shell goes on to execute the next command. Except for the last case, an error message is printed by the shell. All remaining errors cause the shell to exit from a command procedure. An interactive shell returns to read another command from the terminal. Such errors include the following:

- Syntax errors; for example, if ... then ... done
- A signal such as interrupt; the shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal
- Failure of any of the built-in commands such as cd

The shell flag -e causes the shell to terminate if any error is detected.

1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT system call
7*	Unused (formerly EMT instruction)
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	Unused (formerly bus error)
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination (from _kill_ (1))
16	unassigned

Signals marked with an asterisk produce a core dump if not caught. However, the shell itself ignores _quit_, which is the only external signal that causes a dump. The signals in this list of potential interest to shell programs are 1, 2, 3, 14, and 15.

Figure 3-1. ZEUS Signals

3.7 Fault Handling

Shell procedures normally terminate when an interrupt is received from the terminal. The _trap_ command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt), and if this signal is received, executes the commands

```
rm /tmp/ps$$; exit
```

Exit is another built-in command that terminates execution of a shell procedure. The _exit_ is required; otherwise, after the trap has been taken, the shell resumes executing the procedure at the place where it was interrupted.

ZEUS signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. They can be left to cause termination of the process without any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background (Section 3.8), _trap_ commands and the signal are ignored.

The use of _trap_ is illustrated by the modified version of the _touch_ command in Figure 3-2. The cleanup action is to remove the file junk$$.

The _trap_ command appears before the creation of the temporary file; otherwise, it would be possible for the process to terminate without removing the file.

Since there is no signal 0 in ZEUS, it is used by the shell to indicate the commands executed on exit from the shell procedure.

A procedure can itself ignore signals by specifying the null string as the argument to trap. The following fragment is taken from the _nohup_ command:

```
trap '' 1 2 3 15
```

which causes _hangup_, _interrupt_, _quit_, and _kill_ to be ignored both by the procedure and by invoked commands.

```
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do case $i in
     -c)  flag=N ;;
      *)  if test -f $i
          then ln $i junk$$; rm junk$$
          elif test $flag
          then echo file \'$i\' does not exist
          else >$i
     esac
done
```

Figure 3-2. The <u>touch</u> Command

```
d=`pwd`
for i in *
do if test -d $d/$i
     then cd $d/$i
          while echo "$i:"
               trap exit 2
               read x
          do trap : 2; eval $x; done
     fi
done
```

Figure 3-3. The <u>scan</u> Command

Traps can be reset by entering

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps can be obtained by entering

```
trap
```

The procedure _scan_ (Figure 3-3) is an example of the use of _trap_ where there is no exit in the _trap_ command. _Scan_ takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end-of-file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when _scan_ is waiting for input.

Read x is a built-in command that reads one line from the standard input and places the result in the variable x. It returns a nonzero exit status if an end-of-file is read or an interrupt is received.

3.8 Command Execution

To run other than a built-in command, the shell first creates a new process using the system call _fork_. The execution environment for the command includes input, output, and the states of signals, and is established in the _child_ process before the command is executed. The built-in command _exec_, used in the rare cases when no fork is required, replaces the shell with a new command. For example, a simple version of the _nohup_ command looks like

```
trap " 1 2 3 15
exec $*
```

The _trap_ turns off the signals specified so that they are ignored by subsequently created commands, and _exec_ replaces the shell with the command specified.

In the following, _word_ is subject only to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... > *.c
```

writes its output into a file whose name is .c. Input/output specifications are evaluated left to right as they appear in the command.

> *word* — The standard output (File Descriptor 1) is sent to the file *word*, which is created if it does not already exist.

>> *word* — The standard output is sent to file *word*. If the file exists, output is appended by seeking to the end; otherwise, the file is created.

< *word* — The standard input (File Descriptor 0) is taken from the file *word*.

<< *word* — The standard input is taken from the lines of shell input that follow, up to but not including a line consisting only of *word*. If *word* is quoted, no interpretation of the document occurs. If *word* is not quoted, parameter and command substitution occur and \ is used to quote the characters \, $, and `, and the first character of *word*. In the latter case, \new line is ignored.

>& *digit* — The file descriptor *digit* is duplicated using the system call *dup* (2), and the result is used as the standard output.

<& *digit* — The standard input is duplicated from file descriptor *digit*.

<&- — The standard input is closed.

>&- — The standard output is closed.

Any of the above can be preceded by a digit to create the file descriptor specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (File Descriptor 2) directed to *file*. Also,

```
... 2>&1
```

runs a command with its standard output and message output merged. File descriptor 2 is created by duplicating file descriptor 1, but the effect is usually to merge the two streams.

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. First, the default standard input for such a command is the empty file /dev/null. This prevents two processes (the shell and the command), which are running in parallel, from trying to read the same input. For example,

```
ed file &
```

allows both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the ZEUS convention for a signal is that if it is set to 1 (ignored) then it is never changed. The shell command _trap_ has no effect for an ignored signal.

3.9 Invoking the Shell

The following flags are interpreted by the shell when it is invoked. If the first character of argument zero is a minus, commands are read from the file .profile.

-c _string_
: If the -c flag is present, commands are read from _string_.

-s
: If the -s flag is present or if no arguments remain, commands are read from the standard input. Shell output is written to File Descriptor 2.

-i
: If the -i flag is present or if the shell input and output are attached to a terminal (as told by _gtty_), this shell is _interactive_. In this case, TERMINATE is ignored so that kill 0 does not kill an interactive shell, and INTERRUPT is caught and ignored so that wait is interruptable. In all cases, QUIT is ignored by the shell.

APPENDIX A

GRAMMAR

```
item:             word
                  input-output
                  name = value

simple-command:   item
                  simple-command item

command:          simple-command
                  ( command-list )
                  { command-list }
                  for name do command-list done
                  for name in word  do command-list done
                  while command-list do command-list done
                  until command-list do command-list done
                  case word in case-part ... esac
                  if command-list then command-list else-part fi

pipeline:         command
                  pipeline || command

andor:            pipeline
                  andor && pipeline
                  andor || pipeline

command-list:     andor
                  command-list ;
                  command-list &
                  command-list ; andor
                  command-list & andor

input-output:     > file
                  < file
                  >> word
                  << word

file:             word
                  & digit
                  & -

case-part:        pattern ) command-list ;;

pattern:          word
                  pattern | word

else-part:        elif command-list then command-list else-part
                  else command-list
                  empty
```

empty:	
word:	a sequence of nonblank characters
name:	a sequence of letters, digits, or underscores starting with a letter
digit:	0 1 2 3 4 5 6 7 8 9

APPENDIX B

METACHARACTERS AND RESERVED WORDS

Syntactic

| pipe symbol

&& "andf" symbol

|| "orf" symbol

; command separator

;; case delimiter

& background commands

( ) command grouping

< input redirection

<< input from a here document

> output creation

>> output append

Patterns

* match any character(s) including none

? match any single character

[...] match any of the enclosed characters

Substitution

${...} substitute shell variable

`...` substitute command output

Quoting

\ quote the next character

'...' quote the enclosed characters except for '

"..." quote the enclosed characters except for $, `, \, or "

Reserved Words

```
if then else elif fi
case in esac
for while until do done
{  }
```

UUCP INSTALLATION*

* This information is based on an article originally written by D. A. Nowitz, Bell Laboratories.

PREFACE

This document gives the system administrator/installer a detailed description of uucp. The operation of each program in the uucp system, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system are discussed in this document.

TABLE OF CONTENTS

SECTION 1 INTRODUCTION 5

1.1 General 5
1.2 Security 5

SECTION 2 THE UUCP PROGRAMS 7

2.1 Uucp 7

2.1.1 Options 7
2.1.2 Sources and Destinations 7
2.1.3 Types of Work 8

2.2 Uux 10

2.2.1 User Line 11
2.2.2 Required File Line 11
2.2.3 Standard Input Line 11
2.2.4 Standard Output Line 11
2.2.5 Command Line 12

2.3 Uucico 12

2.3.1 Scan for Work 13
2.3.2 Call Remote System 14
2.3.3 Line Protocol Selection 15
2.3.4 Conversation Termination 16

2.4 Uuxqt 16
2.5 Uulog 16
2.6 Uuclean 17

SECTION 3 UUCP INSTALLATION 18

3.1 General 18
3.2 Files Required for Execution 18

3.2.1 Myname 18
3.2.2 L-Devices 19
3.2.3 L-Dialcodes 19

3.3 Login/System Names 19

3.3.1 Userfile 20
3.3.2 L.sys 21

TABLE OF CONTENTS (continued)

SECTION 4 UUCP ADMINISTRATION 24

4.1 General 24
4.2 Sequence Check File 24
4.3 Temporary Data Files 24
4.4 Log Entry Files 25
4.5 System Status Files 25
4.6 Lock Files 26
4.7 Shell Files 26
4.8 Login Entry 27
4.9 File Modes 27

SECTION 1

INTRODUCTION

1.1 General

Uucp is a series of programs that permits communication between ZEUS systems using either dial-up or hardwired communication lines. It is used for file transfers and remote command execution.

Each system participating in the uucp network has a spool directory that stores work to be done. There are three types of files used for the execution of work: data files, work files, and execution files. Data files contain data to be transferred to remote systems. Work files contain the directions for file transfers between systems. Execution files contain the directions for ZEUS command executions that involve the resources of one or more systems.

The uucp system consists of four primary and two secondary programs. The following are primary programs:

uucp creates work files and gathers data files in the spool directory for the transmission of files

uux creates work files, executes files, and gathers data files for the remote execution of ZEUS commands

uucico executes the work files for data transmission

uuxqt executes ZEUS execution files

The secondary programs are:

uulog updates the log file with new entries and reports on the status of uucp requests

uuclean removes old files from the spool directory

1.2 Security

The uucp system, if left unrestricted, lets anyone execute any command and copy in or out any file that is readable/writable by the uucp login user. Necessary precautions should be taken as required by the local implementation.

There are security features available other than the normal file-mode protections that must be set up by the installer of the uucp system.

- The login for uucp does not get a standard shell; the uucico program is started instead. The work can be done only through uucico.

- A path check is performed on file names that are to be sent or received. The user file supplies the information for these checks. The user file can also be set up to require call-back for certain login IDs. (See Section 3.5 for file description.)

- A conversation sequence count can be set up so that the called system can verify the caller's identity.

- The uuxqt program comes with a list of commands (cmp, diff, lpr, and mail) that it executes. A path shell statement (/bin/user/bin) is prepended to the command line by uuxqt.

- The L.sys file must be owned by uucp and have mode 0400 to protect the phone numbers and login information for remote sites. Programs uucp, uucico, uux, and uuxqt must also be owned by uucp and have the setuid bit set.

SECTION 2

THE UUCP PROGRAMS

2.1 Uucp

The uucp command is the primary interface with the system. It sets up file copying and is similar to the ZEUS copy command, cp. Uucp is invoked by the command line

```
uucp   [ option ]  ...  source ...  destination
```

where _source_ and _destination_ contain the prefix system name specifying the system on which files reside or the system on which the files will be copied.

2.1.1 Options

The following options are valid for the uucp command:

- -d — Make directories when necessary for copying the file.
- -c — Use the specified source for the transfer. Do not copy source files to the spool directory.
- -g_letter_ — Insert _letter_ as the grade in the name of the work file. This can be used to change the order of work for a specified system.
- -m — Send mail on completion of the work.

The following options are used primarily for debugging:

- -r — Queue the job, but do not start the uucico program.
- -s_dir_ — Use directory _dir_ for the spool directory.
- -x_num_ — _Num_ is the desired level of debugging output.

2.1.2 Sources and Destinations

If the destination is a directory name, the file name is taken from the last part of the source name. The source name can contain special shell characters such as ",

?, *, [, and]. If a source argument has a system-name! prefix indicating a remote system, the file name expansion is performed on the remote system.

The command

```
uucp  *.c  usg!/usr/dan
```

transfers all files with names ending in .c to the /usr/dan directory on system usg.

The source and destination names can also contain a ~user prefix to refer to the login directory on the specified system. The current directory is prepended to the file name for names with partial path names. File names with ../ are not permitted.

The command

```
uucp  usg!~dan/*.h  ~dan
```

transfers files whose names end with .h in dan's login directory on system usg to dan's local login directory.

2.1.3 Types of Work

For each source file, the uucp program checks the source and destination file names and the system-part of each to classify the work into one of five types:

1. copy source to destination on local system
2. receive files from other systems
3. send files to remote systems
4. send files from a remote system to another remote system
5. receive files from remote systems when the source contains special shell characters, such as ", ?, *, [, and].

After the work has been set up in the spool directory, the uucico program contacts the other system to execute the work unless the -r option is specified.

Type 1 A <u>cp</u> command copies source to destination on the local system. The -<u>d</u> and the -<u>m</u> options are not valid in type-1 operations.

Type 2 — A one-line work file is created for each file requested and is placed in the spool directory with the following fields, each separated by a blank. All work files and execute files use a blank as the field separator.

- R
- the full path name of the source or a ~user/pathname; the ~user part is expanded on the remote system
- the full path name of the destination file; if the ~user notation is used, it is immediately expanded to the user's login directory
- the user's login name
- a minus sign (-) followed by an option list; only the -m and -d options appear in this list

Type 3 — For each source file, a work file is created. The source file is copied into a data file in the spool directory. A -c option on the uucp command prevents the data file from being created. The file is transmitted from the indicated source. The entry fields are as follows:

- S
- the full path name of the source file
- the full path name of the destination or ~user/filename
- the user's login name
- a minus sign (-) followed by an option list
- the name of the data file in the spool directory
- the file mode bits of the source file in octal print format (mode 0666)

Types 4 and 5 — Uucp generates a uucp command and sends it to the remote machine; the remote uucico executes the uucp command.

2.2 Uux

The uux command sets up the execution of a command if the execution system and some of the files are remote. The syntax of the uux command is

```
uux [ - ] [ option ] ... command-string
```

where _command-string_ is composed of one or more arguments. All special shell characters such as <, >, |, and ^ must be quoted, either by quoting the entire command string or by quoting the character as a separate argument. Within _command-string_, the command and file names can contain a system-name! prefix. All arguments must contain an exclamation mark (!) if they are to be treated as files and to be copied to the execution system. The minus sign (-) indicates that the standard input for _command-string_ must be from the standard input of the uux command. The options, which are for debugging, are the following:

- -r — do not start uucico or uuxqt after queuing the job
- -x_num_ — _num_ is the level of debugging output desired

The command

```
pr abc | uux - usg!lpr
```

sets up the output of _pr abc_ as standard input to a line printer (lpr) command to be executed on system _usg_.

Uux generates an execute file containing the names of the files required for execution, the user's login name, the destination of the standard output, and the command to be executed. The execute file is placed in the spool directory for local execution or is sent to the remote system using a generated send command (Type 3 in Section 2.1.3).

Uux generates receive command files (Type 2) for files that are not on the execution system. These command files are placed on the execution machine and executed by the uucico program if the local system has permission to place files in the remote spool directory.

The execute file is processed by the uuxqt program on the execution system. It is composed of several lines, each containing an identification character and one or more arguments. There is no set order for the lines and not all must be present. Each line is described in the following sections.

2.2.1 User Line

The user line is as follows

U user system

where *user* and *system* are the requester's login name and system.

2.2.2 Required File Line

The required file line is

F filename realname

where *filename* is the generated name of an execution system file and *realname* is the last part of the file name, which contains no path information. Zero or more of these lines are present in the execute file. The uuxqt program checks for the existence of all required files before the command is executed.

2.2.3 Standard Input Line

The standard input line is

I filename

The standard input is either specified by a < in the command-string or obtained from the standard input of the uux command if the - option is used. If the standard input is not specified, /dev/null is used.

2.2.4 Standard Output Line

The standard output line is

O filename system-name

The standard output is specified by a > within the command string. If the standard output is not specified, /dev/null is used. The use of >> is not implemented.

2.2.5 Command Line

The command line is

    C command [arguments]
    ...

The *arguments* are specified in the command string. The standard input and standard output do not appear on this line. All required files are moved to the execution directory (a subdirectory of the spool directory) and the ZEUS command is executed using the shell specified in the *uucp.h* header file. In addition, a shell path statement is prepended to the command line as specified in the uuxqt program.

After execution, the standard output is copied or set up to be sent to the designated place.

2.3 Uucico

The copy in, copy out (uucico) program performs the following communications functions between two systems:

- scans the spool directory for work
- places a call to a remote system
- negotiates a line protocol to be used
- executes all requests from both systems
- logs work requests and work completions

Uucico can be started by a system daemon, by one of the uucp, uux, uuxqt, or uucico programs, directly by the user, or by a remote system. The uucico program must be specified as the shell field in the /etc/passwd file for the uucp logins.

When started by a remote system, the program is in SLAVE mode. When started by any other method, the program is in MASTER mode, and a connection is made to a remote system.

The MASTER mode operates in one of two ways. If a system name is specified, that system is called and work is done only for that system. If a system name is not specified, the program scans the spool directory for systems to call.

The uucico program is generally started by another program. There are several options used for execution:

-r1 Start the program in MASTER mode. This is used when uucico is started by a program or "cron" shell.

-s*sys* Do work only for system *sys*. If -*s* is specified, a call to the specified system is made even if there is no work for system *sys* in the spool directory. This program is useful for polling systems that do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

-d*dir* Use directory *dir* for the spool directory.

-x*num* *Num* is the desired level of debugging output.

The following subsections describe the major steps within the uucico program.

2.3.1 Scan for Work

The names of the work-related files in the spool directory have the format

type . system-name grade number

where *type* is an uppercase *C* (copy command file), *D* (data file), or *X* (execute file), *system-name* is the remote system, *grade* is a character, and *number* is a padded four-digit sequence number. For example, the file

C.res45n0031

is a work file for a file transfer between the local machine and the res45 machine.

The scan for work is done by looking through the spool directory for work files (files with prefix C.). A list is created for all systems to be called; uucico then calls each system and processes all work files.

2.3.2 Call Remote System

The call is made using information from several files that reside in the uucp program directory. At the beginning of the call process, a lock is set on the system being called to prevent multiple conversations between the two systems.

The system name is found in the L.sys file. The information contained for each system is the system name, the time to call the system (days-of-week and times-of-day), the device or device type to be used for the call, the line speed, the phone number if the device or device type is an automatic call unit (ACU) or the device name if the device or device type is not ACU, and the login information.

The time field is checked against the present time to see if the call should be made.

The phone number field can contain abbreviations (for example, mh, py, or boston), that get translated into dial sequences using the L-dialcodes file. The same phone number can then be sorted at every site, despite local variations in telephone services and dialing conventions. The phone number field can also contain the string "passive" to denote that this system must initiate the conversation and cannot be called. This configuration is useful for conversing over hardwired connections.

The L-devices file is scanned using the device and line speed from the L.sys file to find an available device for the call. The program tries all devices that satisfy the device types and line speed until the call is made, or until no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of uucico does not attempt to use it. If the call is complete, the login information is used to log in to the remote system. A command is then sent to the remote system to start the uucico program.

The conversation between the two uucico programs begins with a handshake started by the called (SLAVE) system. The SLAVE sends a message to let the MASTER know it is ready to receive the system identification and conversation sequence number. The response from the MASTER is verified by the SLAVE and, if acceptable, protocol selection begins. The SLAVE can also reply with a "call-back required" message and the current conversation is terminated.

2.3.3 Line Protocol Selection

The remote system sends the message

P*proto-list*

where *proto-list* is a string of characters, each representing a line protocol.

The calling program checks *proto-list* for a letter corresponding to an available line protocol and returns a use-protocol message. The use-protocol message is

U*code*

where *code* is either a one-character protocol letter or *N*, which means there is no common protocol.

The initial role (MASTER or SLAVE) for the work processing is the mode in which each program starts. The MASTER is specified by the -r1 uucico option.

There are five messages used during the work processing, each specified by the first character of the message. They are

S Send a file

R Receive a file

C Copy complete

X Execute a *uucp* command

H Hangup

The MASTER sends *R*, *S*, and *X* messages until all work from the spool directory is complete. It then sends an *H* message. The SLAVE replies with SY, SN, RY, RN, HY, HN, XY, or XN, corresponding to yes or no for each request.

The basis for the send and receive replies is the access permission for the requested file/directory obtained by using the userfile and read/write permissions of the file/directory. A copy-complete message is sent by the receiver of the file after each file is copied into the spool directory of the receiving system. The message CY is sent if the file has been successfully copied from the temporary spool file to the actual destination. Otherwise, a CN message is sent. In the case of CN, the transferred file

is in the spool directory with a name beginning with TM. The requests and results are logged on both systems.

The hangup response is determined by the SLAVE program by a work scan of the spool directory. If work for the remote system exists in the SLAVE's spool directory, an HN message is sent, and the programs switch roles. If no work exists, an HY response is sent.

2.3.4 Conversation Termination

When an HY message is received by the MASTER, it is echoed back to the SLAVE and the protocols are turned off. Each program sends a final OO message to the other. The original SLAVE program cleans up and terminates. The MASTER calls other systems and processes work, or terminates if a -*s* option is specified.

2.4 Uuxqt

The uucp command execution (uuxqt) program executes *execute files* generated by uux. The uuxqt program is started by either the uucico or uux programs. The program scans the spool directory for execute files (prefix X.). Each execute file is checked to see if all the required files are available. If so, the command line or send line is executed.

Uuxqt is initiated by executing the shell with the -c option after the appropriate standard input and standard output have been opened. If the standard output is specified, the program creates a send command or copies the output file as designated.

2.5 Uulog

The uucp programs create individual log files for each program invocation. Periodically, uulog can be executed to prepend these files to the system log file. This method of logging minimizes file locking of the log file during program execution.

The uucp log inquiry (uulog) program merges the individual log files and outputs specified log entries. The output request is specified by the following options:

-s*sys* Print entries where *sys* is the remote system name.

-u*user* Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users.

2.6 Uuclean

The uucp spool directory cleanup (uuclean) program is started by the cron process once a day. It removes files that are more than three days old from the spool directory. These are usually files for work that could not be completed.

The uuclean program should be owned by uucp with the setuid bit set (mode 4700).

The options available for uuclean are:

- -d*dir* The directory to be scanned is *dir*.
- -n*hours* Change the aging time from 72 hours to *hours* hours.
- -p*pre* Examine files with prefix *pre* for deletion. Up to ten file prefixes can be specified.
- -x*num* *Num* is the desired level of debugging output.

SECTION 3

UUCP INSTALLATION

3.1 General

Installing uucp under ZEUS requires little effort. The uucp files and directories are described here to facilitate tailoring uucp to a specific environment.

The following three directories are required for execution (default values appear within parentheses):

- program (/usr/lib/uucp) This directory contains the executable system programs and the system files.
- spool (/usr/spool/uucp) This spool directory is used during uucp execution.
- xqtdir (/usr/spool/uucp/.XQTDIR) This directory is used during execution of execute files.

The names _program_, _spool_, and _xqtdir_ are used in this section as a shorthand form to represent their corresponding directory path names.

The modes of spool and xqtdir should be mode 0777, that is, readable, writable, and executable by everyone.

3.2 Files Required for Execution

The five files required for execution must reside in the program directory. The field separator for all files is a space unless otherwise specified.

3.2.1 Myname

This file contains the name of the local system and is used by uucico and mail to identify themselves to other systems. This file should be owned by uccp and should be readable by others (mode 0644).

3.2.2 L-Devices

This file contains entries for the call-unit devices and hardwired connections that are to be used by uucp. The special device files are in the /dev directory. The format for each entry is

line call-unit speed

where _line_ is the device for the line (for example, cul0), and _call-unit_ is the automatic call unit associated with _line_ (for example, cua0). Hardwired lines have a number 0 in this field. _Speed_ is the line speed.

The line

cul0 cua0 300

is for a system that has device cul0 wired to a call-unit cua0 for use at 300 baud.

3.2.3 L-Dialcodes

This file contains entries with location abbreviations used in the L.sys file (for example, py, mh, or boston). The entry format is

abb dial-seq

where _abb_ is the abbreviation and _dial-seq_ is the dial sequence to call that location.

The line

py 165-

is set up so that entry py7777 sends 165-7777 to the dial-unit.

3.3 Login/System Names

The login name used by a remote computer to call a local computer must not be the same as the login name of a local user. However, several remote computers can employ the same login name.

Each computer has a unique system name that is transmitted at the start of each call. This name identifies the calling machine to the called machine.

3.3.1 Userfile

This file contains user accessibility information. It specifies four types of constraints:

1. which files can be accessed by a normal user of the local machine

2. which files can be accessed from a remote computer

3. which login name is used by a particular remote computer

4. whether a remote computer should be called back in order to confirm its identity

Each line in the file has the following format

login,sys [c] path name [path name] ...

where _login_ is the login name for a user or the remote computer, _sys_ is the system name for a remote computer, _c_ is the optional call-back required flag, and _path name_ is a path name prefix that is acceptable for _user_.

The constraints are implemented as follows:

1. When the program is obeying a command stored on the local machine (MASTER mode) the path names allowed are those given for the first line in the user file that has a login name matching the login name of the user who entered the command. If no such line is found, the first line with a null login name is used.

2. When the program is responding to a command from a remote machine (SLAVE mode) the path names allowed are those given for the first line in the file that has a system name matching the system name of the remote machine. If no such line is found, the first one with a null system name is used.

3. When a remote computer logs in, the login name that it uses must appear in the user file. There can be several lines with the same login name, but one of them must either have the name of the remote system or must contain a null system name.

4. If the line matched contains a _c_, the remote machine is called back before any transactions take place.

The line

```
u,m  /usr/xyz
```

allows machine *m* to log in with name *u* and request the transfer of files whose names start with /usr/xyz.

The line

```
dan,  /usr/dan
```

allows the ordinary user, *dan*, to issue commands for files whose names start with /usr/dan.

The lines

```
u,m /usr/xyz  /usr/spool
u,  /usr/spool
```

allow any remote machine to log in with name *u*,. If its system name is not *m*, it can only ask to transfer files whose names start with /usr/spool.

The lines

```
zeus,  /
,  /usr
```

allow any user to transfer files beginning with /usr. The user with login *zeus* can transfer any file.

3.3.2 L.sys

Each entry in this file represents one system that can be called by the local uucp programs. The fields are described below.

SYSTEM NAME

The name of the remote system.

TIME

This string indicates the days-of-week and times-of-day when the system is called (for example, MoTuTh0800-1730). Alternatively, the string can be "passive" to show that only the remote system can initiate a conversation. If the field is passive, the remaining fields are ignored.

The day portion can be a list containing

<u>Su</u> <u>Mo</u> <u>Tu</u> <u>We</u> <u>Th</u> <u>Fr</u> <u>Sa</u>

or it can be <u>Wk</u> for any week-day or <u>Any</u> for any day.

The time must be a range of times (for example, 0800-1230). If no time portion is specified, any time of day can be used for the call.

DEVICE

This is either ACU or the hardwired device to be used for the call. For hardwired devices, the last part of the special file name is used (for example, tty0).

SPEED

This is the line speed for the call (for example, 300).

PHONE

The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one that appears in the L-dialcodes file (for example, mh5900, boston995-9980).

For hardwired devices, this field contains the same string as the device field.

LOGIN

The login information is given as a series of fields and subfields in the format

```
expect  send  [expect  send]  ...
```

where <u>expect</u> is the string expected to be read and <u>send</u> is the string to be sent when the <u>expect</u> string is received.

The expect field is made up of subfields of the form

```
expect[-send-expect]...
```

where the <u>send</u> is sent if the prior <u>expect</u> is not successfully read and the <u>expect</u> following the <u>send</u> is the next expected string.

There are two special names available to be sent during the login sequence. The string EOT sends an EOT character and the string BREAK tries to send a BREAK character. The BREAK character is simulated using line speed changes and null characters and may not work on all devices and systems.

A typical entry in the L.sys file is

```
sys Any ACU 300  mh7654 login uucp ssword: word
```

The expect algorithm looks at the last part of the string as illustrated in the password field.

SECTION 4

UUCP ADMINISTRATION

4.1 General

This section describes some events and files that must be administered for the uucp system. Some administration can be accomplished by shell files initiated by crontab entries. Others require manual intervention. Some sample shell files are given toward the end of this section.

4.2 Sequence Check File

The Sequence Check File (SQFILE) in the program directory contains an entry for each remote system with which conversation sequence checks are to be performed. The initial entry is the system name of the remote system. The first conversation adds two items to the line: the conversation count, and the date/time of the most resent conversation. These items are updated with each conversation. If a sequence check fails, the entry must be adjusted.

4.3 Temporary Data Files

Temporary Data Files (TM) are created in the spool directory while files are being copied from a remote machine. Their names have the form

```
TM.pid.ddd
```

where _pid_ is a process-id and _ddd_ is a sequential three-digit number starting at zero for each invocation of uucico and incremented for each file received.

After the entire remote file is received, the TM file is moved or copied to the requested destination. If processing is abnormally terminated or if the move or copy fails, the file remains in the spool directory. These unused files must be removed periodically with the uuclean program. The command

```
uuclean -pTM
```

removes all TM files more than three days old.

4.4 Log Entry Files

During execution of programs, individual Log Entry Files (LOG files) are created in the spool directory with information about queued requests, calls to remote systems, execution of uux commands, and file copy results. These files must be combined into the LOGFILE by using the uulog program. The command

uulog

puts the new LOG files at the beginning of the existing LOGFILE. Options are available to print some or all the log entries after the files are merged. The LOGFILE must be removed periodically since it is copied each time new log entries are put into the file.

The log files are created with mode 0222. If the program that creates the file terminates normally, it changes the mode to 0666. Aborted runs can leave the files with mode 0222 and the uulog program does not read or remove them. To remove them, use either rm or uuclean, or change the mode to 0666 and let uulog merge them with the logfile.

4.5 System Status Files

System Status Files (STST) are created in the spool directory by the uucico program. They contain information of failures such as login, dialup, or sequence check. They contain a TALKING status when two machines are conversing. The form of the file name is

STST.sys

where _sys_ is the remote system name.

For ordinary failures, such as dialup and login, the file prevents repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it contains a talking status. In this case, the file must be removed before a conversation is attempted.

4.6 Lock Files

Lock files (LCK) are created for each device in use, for example, the automatic calling unit and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

```
LCK..str
```

where str is either a device or system name. The files can be left in the spool directory if runs abort. They are ignored (reused) after 24 hours. When runs abort and calls are desired before the time limit, the lock files must be removed.

4.7 Shell Files

The uucp program spools work and attempts to start the uucico program, but the starting of uucico sometimes fails. Therefore, the uucico program must occasionally be started. The command to start uucico can be put in a shell file with a command to merge log files and started by a crontab entry on an hourly basis. The file contains commands such as

```
program /uulog
program /uucico   -rl
```

The "-rl" option is required to start the uucico program in MASTER mode.

Another shell file can be set up on a daily basis to remove TM, ST, and LCK files, and C. or D. files for work that cannot be accomplished. Use a shell file containing commands such as

```
program /uuclean   -pTM -pC. -pD.
program /uuclean   -pST -pLCK -nl2
```

The -nl2 option causes the ST and LCK files older than 12 hours to be deleted. If there is no -n option, a three-day limit is used.

A daily or weekly shell must also be created to remove or save old logfiles. Use a shell such as

```
cp spool /LOGFILE    spool /o.LOGFILE
rm spool /LOGFILE
```

4.8 Login Entry

One or more logins must be set up for uucp. Each of the "/etc/passwd" entries must have program/uucico as the shell to be executed. The login directory is not used, but if the system has a special directory for use as a sending or receiving file, it must be the login entry. The various logins are used in conjunction with the user file to restrict file access. Specifying the shell argument limits the login to the use of uucp (uucico) only.

4.9 File Modes

The owner and file modes of various programs and files are to be set as follows.

The programs uucp, uux, uucico, and uuxqt must be owned by uucp with the setuid bit set and execute only permissions (mode 04111). This prevents outsiders from modifying the programs to get at a standard shell from the uucp login.

The L.sys, SQFILE, and the user file that are in the program directory must be owned by uucp and set with mode 0400.

Introduction to Display Editing with vi*

* This information is based on an article written by William Joy and revised by Mark Horton.

TABLE OF CONTENTS

SECTION 1 INTRODUCTION 1

1.1 General 1
1.2 Command Notation 1
1.3 Special Characters 2
1.4 Invoking vi 3
1.5 Operating Modes 5
1.6 Escape to the Shell 6
1.7 Leaving vi 6
1.8 vi and ex 7
1.9 Using vi on Hardcopy Terminals and "Glass TTYs" 7
1.10 Uppercase Terminals 9
1.11 Slow Terminals 9
1.12 Abbreviations 9
1.13 Line Numbers 9
1.14 Line Representation in the Display 10
1.15 End of File Indicators 10
1.16 Counts 10

SECTION 2 vi DISPLAY CONTROL 13

2.1 Scroll Control 13
2.2 Page Control 13
2.3 String Searches 13
2.4 Cursor Position Control 15
2.5 Tags 18
2.6 File Status 19
2.7 Clearing the Display 19
2.8 Window Size 19

SECTION 3 EDIT COMMANDS 23

3.1 General 23
3.2 Insert Text 24
3.3 Delete and Insert Characters 26
3.4 Delete Operator 27
3.5 Undo Operator 30
3.6 Program Editing Features 31
3.7 Erase and Line Kill Characters 32

SECTION 4 REARRANGING AND DUPLICATING TEXT 33

4.1 General 33
4.2 Buffers 34
4.3 Text Manipulation 34

TABLE OF CONTENTS (continued)

SECTION 5 FILE MANIPULATION 39

5.1 Writing, Quitting and Editing New Files 39
5.2 File Manipulation Commands 40

SECTION 6 OPTIONS 45

6.1 General 45
6.2 Editing on Slow Terminals 47
6.3 Ignore case 47
6.4 Magic Characters 48
6.5 Autoindent and Shiftwidth 49
6.6 Continuous Text Input 50
6.7 LISP Editing Options and Commands 50
6.8 Line Numbers 51
6.9 Tabs and End of Line Indicators 51
6.10 Automatic Writing of Files 51
6.11 Defining Paragraphs and Sections 52
6.12 Terminal Type 52
6.13 Scroll 53
6.14 Terse 53
6.15 Window 53
6.16 Wrapping Around the End of Files 53

SECTION 7 RECOVERING LOST INPUT 55

7.1 Lost Lines 55
7.2 Lost Files 55

SECTION 8 MISCELLANEOUS 57

8.1 Filtering Portions of the Buffer 57
8.2 Typing Non-Printing Characters 57

APPENDIX A SPECIFYING TERMINAL TYPE 59

APPENDIX B vi CORRECTION CHARACTERS 61

APPENDIX C vi SYMBOL DICTIONARY 63

APPENDIX D vi QUICK REFERENCE 77

LIST OF TABLES

Table

5-1 File Manipulation Commands 40

6-1 Frequently Used Options 45
6-2 Magic Option Extended Operators 48

B-1 Operators Used for Corrections and Changes 61

SECTION 1

INTRODUCTION

1.1 General

Vi (Visual) is a display oriented interactive text editor in which the display acts as a window into the file being edited. Changes are reflected in the display, and this simplifies modifications. The regularity and the mnemonic assignment of commands makes the editor command set easy to remember and use. The full command set of the more traditional, line-oriented editor ex is available with vi, and it is easy to switch between the two editing modes.

Vi can be used on a wide variety of display terminals. New terminals are easily driven after editing a terminal description file. While it is advantageous to have an "intelligent" terminal that can insert and delete lines and characters from the local display, the editor functions well on "dumb" terminals with low-bandwidth telephone lines. The editor optimizes response time by using a smaller window and a different display updating algorithm. The command set of vi can be used as a one-line-window editor on hard-copy terminals, storage tubes, and "glass TTYs."

This document was written on the assumption that the system being used is a Zilog S8000, that the system console is a Lear Siegler ADM-31, and that the system software is Zilog ZEUS, a super-set of UNIX.

1.2 Command Notation

In this document, the following notation is used in command descriptions.

< >	Angle brackets enclose descriptive names for the data or item to be entered. For example, <filename>.
[ ]	Square brackets enclose optional data.
|	Bar denotes an OR function.
ESC	denotes the escape key (ALT on some keyboards)
RUB	denotes the delete key (DEL on some keyboards)

CTRL denotes the CONTROL key. On certain terminals, CTRL is echoed as the circumflex symbol (^). Do not confuse the echo with the symbol used in this document for the up arrow (shown below).

↑ denotes an up arrow

1.3 Special Characters

ESC key: this key cancels partially entered commands. It also terminates text mode operations. If the terminal is already quiescent this key may also trigger a bell or audio annunciator.

RETURN key: this key initiates execution of most commands. It also initiates the csh (C Shell) commands.

RUB key: this key interrupts and stops the editor.

Interrupting the editor while it is redrawing or otherwise updating large portions of the display, might cause a confused display. If this occurs, it is still possible to continue editing by:

1. Entering the command

    CTRL-z

   redraws the display.

2. Ignoring the state of the display and either moving or searching again.

For the purposes of this document, the use of RUB is equivalent to an interrupt.

Slash (/): This symbol specifies a string for a search. When this key is pressed, the cursor moves to the bottom line of the display, where it acts as a prompt. To return the cursor to the current position, press RUB. Backspacing over the slash will also cancel the search.

The line kill and erase characters are user programmable. They can be changed with the program stty (refer to STTY(1) in the ZEUS Programmer's Reference Manual).

Line kill: the line kill character is usually the character

    @

Character erase: the erase character is usually

CTRL-h

1.4 Invoking vi

When the system is up and running, set the terminal type, as shown:

```
%setenv TERM <code> (RETURN)
```

where: % is the system prompt
setenv is the command for setting the environment
TERM is a required keyword
<code> is the terminal type code to be entered
For the Lear Siegler terminal, <code> is adm31
(RETURN) is the RETURN key

For other terminals, and additional information relevant to setting the terminal type, refer to Appendix A.

After the terminal type has been set, invoke vi. The command format is

```
% vi [-t <tag>] [-r[<filename>]] [+[<command>]]
   [+<n>] [+/<string>] [-l] [<filename>] (RETURN)
```

where: % is the system prompt

vi is the command to invoke the visual display editor

-t <tag> is the option to edit the file containing the tag <tag> at <tag> (see Section 5.2)

-r [<filename>] is the option used to recover a file after an editor or system crash (see Section 7.2)

+[<command>] executes the ex command <command> prior to entering visual mode. Without <command> the visual editor starts at the end of file (see also Section 5.2)

+<n> starts the visual editor at line number <n>

+/<string> causes vi to search for and start at the string <string>

-l is the option to set editing options for LISP (see Section 6)

<filename> is the name of the file to be edited.

NOTE

Do not include the square brackets ([]) and the angle brackets (<>) in the command.

Examples:

1. The simplest vi command is to invoke vi for editing a single file:

```
%vi <filename> <RETURN>
```

where: <filename> is the name of the file to be edited.

NOTE

All entries must be terminated by a RETURN. For the remainder of this document, neither RETURN nor the system prompt is shown in the system commands; however, it is assumed that each command is terminated by RETURN.

2. To start vi at line number n, use the form

```
vi [+<n>] <filename>
```

3. To start vi at some string <string>, use the form

```
vi [+/<string>] <filename>
```

   vi searches for <string> and, if found, starts at <string>. For additional details, see Section 5.2.

After the command is entered, the file name is echoed on the screen. The editor does not directly modify the file being edited. Rather, the editor copies the file in a buffer, and then remembers the file name. The contents of the file are not affected until the changes are written back to the original file.

After a file has been copied, vi edits that file. The display clears, and the text of the file appears on the display. If it does not

1. Check for the correct terminal type code. An incorrect type code entry produces an unusable display. To check

exit vi, enter

:q

Following the RETURN, control returns to the shell (command interpreter). To verify the correct terminal code, enter:

printenv TERM

and reenter it as described above.

2. Check for the correct filename. An incorrect filename can result in the display of an error diagnostic. If this occurs, return to the shell (as shown in 1. above) and restart.

3. If the editor does not respond, interrupt it with the delete key DEL (or RUB). Then, return to the shell (as shown in 1. above).

1.5 Operating Modes

Vi has four operating modes:

1. Command mode. This is the initial state and the normal operating mode. The other modes return to this mode. The escape key (ESC) cancels any partially entered command.

2. Text mode. This mode is entered by one of the following operators:

   a A i I o O c C s S R

   Any desired text can be entered in this mode. Text entry is normally terminated with the ESC. Text entry can also be terminated (abnormally) with RUB.

3. Last line mode. This mode is initiated by the following operators:

   : / ? !

   Commands or string searches are executed after a RETURN or ESC. Commands are canceled with DEL (or RUB).

   When the editor is in this mode, commands are echoed on the last line. If the cursor is in the first position of the last line, the editor is performing a computation such as computing a new position in the file after a search, or running a command to reformat part of the

buffer. While this is happening, it is possible to stop (interrupt) the editor with RUB. On some systems, when the cursor is on the bottom line, and the editor has been interrupted, the operator cannot type ahead.

4. Open mode. This is described in Section 1.9.

1.6 Escape to the Shell

To execute a shell command, while in vi, use a command of the form

```
:!<command>
```

where <command> is the shell command. The system runs the <command> and returns to vi when the command is completed. The operator is prompted

```
Hit RETURN to continue
```

After RETURN is entered, the editor clears and redraws the display; vi resumes control, and editing can continue. However, if another : command is entered prior to the RETURN, the display is not redrawn.

To execute more than one command in the shell, enter the command

```
:sh
```

When all necessary shell commands are completed, return to vi by entering

```
CTRL-d
```

Vi clears the display and editing can continue.

1.7 Leaving vi

To leave vi and return to the shell, use the command

```
ZZ
```

If changes have been made to the text, the contents of the vi buffer are written back into the original file, and the editor exits. If no changes have been made, the editor exits.

It is also possible to write the changes to the file without leaving vi by using the command

:w

To exit vi (quit) without writing the changes, use the command

:q!

This discards all text changes. This command is convenient when changes have been made to the contents of the buffer and the original file must remain unchanged. Do not use this command for changes that must be saved.

1.8 vi and ex

Vi is one mode of editing within the line-oriented editor ex. Some operations are easier in ex than in vi, such as systematic changes in line-oriented material. Experienced users often mix vi and ex commands to facilitate their work.

When vi is running, it is possible to escape to ex with the command

Q

Ex prompts with a colon (:). The vi commands prefaced with a colon (:) that are described in this document are available in ex. Similarly, most of the ex commands are available in vi when prefaced with a colon.

In rare instances, an internal error may occur in vi. In this case, a diagnostic is displayed, vi exits, and control returns to the command mode of ex. It is then possible to either:

1. Save the work in progress and quit by entering the command

   x

   or,

2. Re-enter vi with the command

   vi

1.9 Using vi on Hardcopy Terminals and "Glass TTYs"

It is possible to use vi on a hardcopy terminal, or a terminal with a cursor that cannot move from the bottom line. On these terminals, vi runs in "open" mode. In this mode, when

a vi command is entered, the editor states that it is in open mode. This name comes from the open command in ex, which invokes the open mode. With a "dumb" terminal, vi automatically enters open mode.

To invoke open mode manually, enter ex, and then, from ex, enter the command

    open

to return to ex from open mode, enter the command

    Q

To return to vi from ex, enter

    vi

The differences between visual and open mode are:

1. The way the text is displayed. In open mode, the editor uses a single-line window into the file. Moving backward and forward in the file displays new lines, which are always below the current line.

2. The command

        z

   takes no parameters, but draws a window of context around the current line and returns to the current line.

3. On a hardcopy terminal, the command

        CTRL-R

   retypes the current line. On these terminals, the editor usually uses two lines to represent the current line. The first line is a copy of the original line, and the second line is the work line; that is, it shows any editorial changes. When characters are deleted, the editor displays a number of backslashes (\) to show what characters were deleted. The editor also reprints the current line soon after such changes so that they are visible.

1.10 Uppercase Terminals

Vi can be used on uppercase-only terminals by using the normal terminal typing conventions. All characters are converted to lowercase characters. However, each upper case character must be preceded with a backslash (\). The combination "\character" does not echo until the backslash is followed by the second character.

The following characters are not available on uppercase-only-terminals:

{ } | ~ `

These characters can be entered as shown below:

```
For { use \(
For } use \)
For ~ use \^
For | use \!
For ` use \'
```

1.11 Slow Terminals

The vi editor minimizes the delay time required for display updates by limiting the output to the display. For slow and for "dumb" terminals, vi optimizes screen updates during text mode, and it replaces deleted lines with the symbol "@".

On slow terminals that can support vi in the full screen mode, it is useful to use "open" mode.

Vi has an operating option (slowopen) that is convenient when a slow terminal is being used. For additional information, see Section 6.2.

1.12 Abbreviations

Vi has a number of short commands that abbreviate longer commands that have been introduced above. These commands are listed on the quick reference card.

1.13 Line Numbers

The vi editor, if desired, can number each line. Use the editor option, "number" (line number option) which is described in Section 6.8.

1.14 Line Representation in the Display

The vi editor folds long logical lines into shorter physical lines on the display. Commands that advance lines also advance logical lines. Hence they skip over all segments of a line in one motion. The command

    |

moves the cursor to a specific column, and it can be useful for getting near the middle of a long line to split it. (This command is a vertical bar, not a numeral one or a lowercase l). For example, the command

    80|

places the cursor on the 80th column in a long sentence.

On a "dumb" terminal, the editor puts only full lines on the display; if there is not enough room on the display to fit a logical line, the editor leaves the physical line empty and places an @ on the line as a place indicator. When lines are deleted, the editor often just clears each text line and displays an "@" to save time, rather than rewriting the entire display. To maximize the information on the display enter:

    CTRL-R

1.15 End of File Indicators

When the end of the file is displayed, and the last line is not at the bottom of the display, the vi editor displays the tilde (~) at the left end of each remaining line. This indicates that the 1st line of the file is shown in the display, and that those lines with the tilde are past the end of the file.

1.16 Counts

A count is an argument that affects the number of times the command is executed, or the number of lines affected. Several vi commands use a preceding count that affects the operation of the command. Some of the most common are the following:

1. For the following commands, a preceding count affects the amount of scroll:

   CTRL-d   CTRL-u

2. For the following commands, the count affects the line or column number:

   z   G |(vertical bar)

3. For most vi commands, a preceding count affects the number of times the command is repeated. For example, the command

   5RETURN

   advances 5 words. The command

   5dw

   deletes 5 words

   3.

   deletes 3 more words.

VI

12

SECTION 2

vi DISPLAY CONTROL

2.1 Scroll Control

Use the following commands to scroll the display:

[<n>]CTRL-u to scroll up n lines.
[<n>]CTRL-d to scroll down n lines.

If n is omitted the default is half the window size.

NOTE

Certain "dumb" terminals cannot scroll up. In this case, CTRL-U clears the display and refreshes it with a line that is farther back in the file (towards the top).

2.2 Page Control

The functions CTRL-F and CTRL-B move the viewing window forward and backward one page, respectively. Both commands retain a few lines of text from the previous page for continuity. It is possible to read through a file using the page commands rather than the scroll commands. The primary difference is that the scroll commands move the text smoothly and leave more of the previous text, whereas the page commands change a page at a time, leaving only a few lines of text for continuity.

2.3 String Searches

The search function also positions the display within a file. This function searches the text file for a particular string of characters and positions the cursor at the next occurrence of the specified string. The search command is:

/<string>

To search backwards from the location of the cursor, use the command

:<string>

To repeat the forward or backward string search to the next occurrence of <string>, use the command

n

To repeat the string search in the reverse direction enter

N

If <string> is not present in the text file, vi prints the message "Pattern not found" on the last line of the screen, and returns the cursor to its original position. String searches normally wrap around the end of the file, and to find the string even if it is not in the direction originally specified in the command (provided the string is indeed in the file). The wraparound function can be disabled by the editor option "nowrapscan" (or nows). The nowrapscan option is one of the options described briefly in Section 6. Refer to the "Ex Reference Manual" by William Joy; July 20, 1979.

If the search is to match a string at the beginning of a line, then precede the search string with an up arrow (↑). To match only at the end of a line, end the search string with $.

Examples:

/^search

searches for the word "search" at the beginning of a line, and

/last$

searches for the word "last" at the end of a line.

If the search string contains a slash (/), it must be preceded by a backslash (\). This is also true if the editor option, "magic," is set (see Section 6).

At the end of the string search, vi places the cursor at the next or the previous occurrence of the string, as appropriate.

Whole lines of text can be affected up to the line prior to the line containing the string. To do so, use a search command with the form

/<string>/-<n>

where: <string> is part of the search command, and

<n> is the number of lines preceding the line containing the string.

A "+" can be substituted for the "-". The result is that the search locates the string <n> lines after the line containing <string>. If no line offset is included, the editor affects characters up to the point of the string match, rather than whole lines. Thus, use "+0" to affect the line that matches.

The editor, if commanded, ignores the case of words in the string search. This is briefly described in the ignore case option in Section 6.

String searches can also be used in conjunction with the operators "d" and "c" (see Section 3.4), and "y" (see Section 4.3).

2.4 Cursor Position Control

To position the cursor at any particular line, where the lines are identified by number, use the command

```
[<n>]G
```

where n is a line number. Thus, 1G moves the cursor to the first line in the file. If <n> is omitted, the default is the last line of the file.

The cursor can be moved up, down, forward and back by the following keys:

up: k, CTRL-p, or CTRL-k
down: j, CTRL-n, or CTRL-j

back: h, CTRL-h, or backspace
forward: space bar, or l

Some terminals have arrow keys (four or five keys with arrows going in various directions) that have the same functions. (On the HP 2621 the function keys must be shifted.)

To advance the cursor to the first non-white position of the next line in the file, strike RETURN or "+" key. Similarly, strike "-" to move the cursor back to the first non-white position on the preceding line. These keys can also be used to scroll when the cursor is at the top or bottom of the display, as appropriate.

Vi also has commands to position the cursor at the top, middle,or the bottom of the display. For the top, strike

the H key. Striking

<n>H

moves the cursor n lines down from the top of the display. The <n> is optional; the default position is the top of the display. Similarly, the command "M" positions the cursor in the middle of the display. The command

<n>L

positions the cursor either on the last line of the display, or the nth line from the bottom. If the <n> is omitted, the default is the bottom of the display.

The cursor can also be moved within a line with any of the following commands. To position the cursor on some word other than the first word, use the command

[<n>]w

which moves the cursor right to the beginning of the nth word on the line. The default is one word. The command

[<n>]b

moves the cursor back n words. The default is one word. The command

[<n>]e

advances the cursor right to the end of the nth word, rather than the beginning of the word. The default is one word.

The commands "b", "w" and "e" stop at punctuation marks. To move the cursor forward or backward without stopping at punctuation, use the characters "W", "B" or "E", respectively. The word keys wrap around the end of the line, and continue to the next line.

After the cursor has been moved for any reason, it can be returned to its previous position with the command `` (two back single quotation marks). The command '' (two forward single quotation marks) moves the cursor to the first non-white character of the line containing the previous position mark ('').

This is often more convenient than the command G because it requires no line count or other preparation.

To move the cursor to the first non-white position on the current line of text, use either "0" or the up arrow (↑).

To move the cursor to the end of the current line, use "$."

The command

[<n>]f<c>

moves the cursor to the nth subsequent occurrence of the character <c>. The default is the next occurrence. Repeat by using the semicolon (;). The inverse command is

[<n>]F<c>

This performs the same function, but moves the cursor backward (into the preceding text). Repeat with a semicolon.

To move the cursor to the character preceding the nth occurrence of the character <c>, enter:

[<n>]t<c>

To move the cursor backwards to the character following the nth occurrence of the character <c>, enter:

[<n>]T<c>

The commands (f, F, t, and T) can be repeated with the semicolon, or the direction can be reversed with the comma.

To move the cursor to the matching parenthesis in a pair, place the cursor at either an opening or closing parenthesis and strike the percent (%) key. This feature also works for braces ({}) and square brackets ([]).

To advance the cursor to the beginning of the nth sentence following, use the command:

[<n>])

where the default for n is one. Similarly, to move the cursor back to the beginning of a sentence, use the command

[<n>](

where the default for n is one. A sentence is defined as ending with a period, a question mark or an exclamation point, followed either by two spaces or by an end of line. Sentences also begin at paragraph and section boundaries. For example, the command

2)

advances the cursor one sentence beyond the end of the current sentence.

To move the cursor forward to the beginning of the next paragraph, use the closing brace (}); similarly, to move the cursor back to the beginning of the preceding paragraph, use the opening brace ({). To move the cursor additional paragraphs, precede the brace with a count, n. For example, the command

    3}

advances three paragraphs. A paragraph begins after an empty line or at a section boundary.

Finally, to move the cursor to the beginning of the next section, use a double closing square bracket:

    ]]

Use a double opening square bracket:

    [[

to move the cursor back to the previous section boundary.

2.5 Tags

It is possible to mark a position in the editor file with a single letter tag, and then to return to any particular tag. To tag a position in text, use the command

    m<tag>

where the tag is any letter of the alphabet.

To return to the tag, use the command

    `<tag>

When using operators (such as the delete operator) with a tagged line, it may be convenient to operate on entire lines (for example, to delete entire lines), rather than to the exact position of the tag. In this case, use the form

    '<tag>

rather than the form

    `<tag>

For example, the command

d`<tag>

deletes entire lines from the position of the cursor to the line with the tag.

2.6 File Status

To find out the file status, enter the command

CTRL-g

The editor displays the name of the file being edited, the number of the current line, the number of lines in the buffer, and the relative position in the buffer as a percentage.

2.7 Clearing the Display

If, for any reason, the terminal display is garbled, it is often possible to obtain a correct display by using the command:

CTRL-l

or

CTRL-z

depending on the terminal. On a "dumb" terminal, when one or more lines have been deleted, it is possible to eliminate the "@" symbols with the command

CTRL-R

or

CTRL-r

This redraws the display and closes the deleted line(s).

2.8 Window Size

The window size is the number of lines written on the display. Vi maintains the current or default window size. On terminals that run at speeds greater than 1200 baud, the editor uses the full terminal display. On slower terminals (most dialup lines are in this group) the editor uses eight

lines as the default window size. On terminals that run at 1200 baud, the default window size is 16 lines.

The appropriate window size is used when the editor clears and refills the display after a search or other motion that moves beyond the edge of the current window. Commands that take a new window size as count (see Section 1.16) often cause the display to be redrawn. With some of these commands, a smaller window size may be equally convenient, and it may be expedient to specify a smaller window size with the appropriate command. In any case, the number of lines displayed increases when:

1. Commands such as "-" are used; these move the window up.

2. commands such as "+", RETURN or CTRL-d are used; these move the window down.

The scroll commands CTRL-d and CTRL-u "remember" the amount of scroll last specified. The default is half the window size.

The editor makes editing easier at low speeds by starting with a small window and expanding as the editing progresses. The editor can expand the window easily when inserts are placed in the middle of the display on intelligent terminals.

The window can be enlarged or reduced, and the current line, or any desired line, can be placed anywhere in the window with the command

<m>z<n><suffix>

where: <m> is the line number. The default is the current line

z is the command operator

<n> is the number of lines in the window

<suffix> controls the position of the desired line within the window, and is any of the following:

<RETURN> places the line at the top
. places the line at the center
- places the line at the bottom

For example, the command

z5.

redraws the display with the current line in the center of a five line window, while the command

```
5z5.
```

places line five in the center of a five line window.

SECTION 3

EDIT COMMANDS

3.1 General

In general, the edit commands use text mode. Text mode is initiated by entering of one of the various insert commands. Following the entry of the insert command, all subsequent keystrokes become text insertions. The text insert mode is always terminated by striking the (ESC) key.

Many related editor commands are invoked by the same alpha key and differ only in that one is given by a lowercase key, and the other is given by an uppercase key. The uppercase key usually differs from the lowercase key only in the sense of direction: the uppercase key operates backward and/or up and the lowercase key operates forward and/or down.

Using any of the text mode commands, it is possible to insert one letter, or many lines of text. To insert more than one line of text, strike the RETURN key in the middle of the input. A new line is then created for text and the insertion can continue. For slow or "dumb" terminals, the editor may wait to redraw the tail of the screen. In this case, the new text overwrites existing lines on the display. This avoids delays that occur if the editor attempts to keep the tail of the display up to date. The display is updated correctly when text mode is terminated.

Those characters normally used at the system command level for character or line deletion can also be used in text mode (e.g., CTRL-h or #; and @, CTRL-x or CTRL-u, as appropriate). CTRL-H always erases the last input character, regardless of the erase character.

Backspacing (while in text mode) does not erase characters. The cursor moves backwards, but the characters remain on the display. This is useful for entering similar text. The display is updated after the escape. To correct the display immediately, use the ESC, and reenter text mode.

It is not possible to backspace around the end of a line. To back up for a correction on a previous line, use ESC and then move the cursor back to the previous line. Make the correction, return and then reenter the appropriate text command.

NOTES

The character CTRL-W erases a whole word and leaves a space after the previous word. This is useful for backing up quickly for an insert.

It is not possible to erase characters with CTRL-W unless these characters were entered in text mode.

3.2 Insert Text

The general form of the text mode command is

<n><command><string>ESC

where: <n> is a preceding count; the default is one
<command> is one of the insert mode commands listed below
<string> is the inserted text string
ESC is the escape key

The effect of the preceding count is to repeat the inserted string n times. All one of the following command operators can be used to enter insert mode:

a A i I o O c C s S R

These commands and their variations are described below.

To insert text in the file, use one of the insert mode commands. For example,

i

Following the "i" (or other insert mode operator), all subsequent string of characters or text entered on the terminal are inserted in the file, until insert mode is terminated. To terminate insert mode, strike ESC (escape). On certain "dumb" terminals, when text is inserted, the display appears to overwrite the original text. When insert mode is terminated, all inserted and previous text is displayed properly.

A variation of the "i" command is

^i

which inserts text at the beginning of a line. The command

I

is equivalent.

In general, most of the insert commands can have a preceding count. For example, the command

    5iapple

repeats the word "apple" five times:

    appleappleappleappleapple

In the following description the preceding count is not always shown.

The command

    a

also enters the vi text mode. The difference between the two commands is that with the command "i," text is inserted before the cursor (to the left), whereas with "a," text is inserted after the cursor (to the right). The command "a" is sometimes convenient for appending one or more letters to a word. The append operation is also terminated with the ESC key.

A variation of the command "a" uses the dollar sign,

    $a

to move the cursor to the end of the current line and append text. An equivalent command is

    A

Another way to add one or more lines of text to the file is to use the command

    o

This opens the existing text and adds new text below the current line. Similarly, the command

    O

opens and adds new text above the current line. Both commands are terminated with ESC. A preceding count opens n lines.

It is also possible to insert non-printing characters in the text. Refer to Section 8.2.

3.3 Delete And Insert Characters

To delete a character or characters, place the cursor on the character to be deleted. Use the following command

    [<n>]x

where: <n> is the number of characters and spaces to be deleted; the default is one.

x is the character delete command

To delete a character or characters preceding the cursor, use the command

    [<n>]X

where: <n> is the number of characters and spaces to be deleted; the default is one.

X is the character delete command

To replace (change) one or more characters, use the command

    [<n>]r<c>

where: <n> is the number of characters to be changed,

r is the replace command,

<c> is limited to one character which is repeated n times in place of n deleted characters.

To replace (change) one or more characters with a string, use the command

    [<n>]R<string>

where: <n> is the number of times the replacement is performed

R is the replace command

<string> is the string used for replacement. The string can be any length.

To replace a number of characters with more than one character, use the command:

    [<n>]s<string>

where: <n> is the number of characters to be replaced,

s is the substitute command,

<string> is the string that is substituted for the deleted characters. The string can be any length.

Use ESC to terminate string input.

3.4 Delete Operator

The command

```
d
```

acts as the delete operator. To delete n words, position the cursor, and then enter

```
[<n>]dw
```

or

```
d[<n>]w
```

The default is one word.

To delete a word backwards (to the left of the cursor), enter

```
[<n>]db
```

or

```
d[<n>]b
```

The default is one word.

To delete n single characters, position the cursor on the appropriate starting character, and enter the command

```
[<n>]d<space>
```

This is equivalent to the x command. The default is one space.

A variation of the "d" command is

```
d$
```

which deletes the rest of the text on the current line. An equivalent command is

D

The operator "c" changes entire words. To change n words, enter the command

[<n>]cw

When the commmand is entered, the end of the text to be changed is marked with the symbol "$". Enter the replacement text, and terminate text entry with ESC. The default is one word.

A variation of the "c" command is

c$

which changes the rest of the text on the current line. An equivalent command is

C

When operating on a line of text, it is often desirable to delete the characters up to the first instance of a character. To do so, use the command

[<n>]df<x>

where f<x> locates the nth occurrence of the character <x> following the cursor. The default is the first occurrence of <x>. This command deletes the text up to--and including--the character <x>. A variant is the command

[<n>]dt<x>

where the operator f is replaced by the t. In this instance, the text is deleted up to--but not including--the character <x>. The command

T

is similar, but it operates in the reverse of the t operator--that is, it operates in the preceding text.

To delete n entire lines, use the delete operator twice:

[<n>]dd

The default is one line.

On a "dumb" terminal, the editor may sometimes erase the entire line on the screen and replace it the symbol "@" at the far left. This does not correspond to any line in the

file, but is a place indicator; it helps avoid a lengthy redraw of the display, which would be required in order to close up the deleted lines.

The operator

    [<n>]cc

is similar to the command "dd", but it leaves vi in text mode, whereas dd does not. The command "cc" is convenient for changing an entire line. Position the cursor as appropriate, enter the command, and then enter the replacement text. Terminate text mode operation with ESC. The command

    [<n>]S

is synonymous to the command "cc", and it is analogous to the command "s". Think of the "s" as a character substitute and the "S" as a line substitute.

There are several other variations on the line delete commands. The command

    d<n>L

deletes all of the lines from the cursor down to the nth line from the botttom of the display. The default is all lines to the bottom of the display.

It is also possible to use a string search with the delete operator:

    d/<string>

This command deletes characters from the cursor position to the point of the string match. Similarly, the command

    d/<string>/-n

deletes characters from the cursor position to the nth line preceding the string match. The command

    d/<string>/+n

deletes characters from the cursor to the nth line following the string match. Similar commands can be used to change entire lines in relation to a string:

    c/<string>/-n

and

c/<string>/+n

In editing a document, it is usually easiest to edit in terms of sentences, paragraphs and sections. The operators "(" and ")" can be used with the delete operator. For example, the command

[<n>]d)

deletes the rest of n sentences. The default is from the cursor position to the end of the current sentence. Similarly,

[<n>]d(

performs one of two deletions:

1. With the cursor at the beginning of a sentence, the command deletes the previous n sentences, or

2. When the cursor is not at the beginning of a sentence, the command deletes the text from the cursor back to the beginning of n sentences. The default is the beginning of the current sentence. The editor displays the extent of the change; it also indicates when a change will affect text that is not shown on the display.

To repeat the command more than once, use the period (.) key.

3.5 The Undo Operator

Vi has an undo operator

u

that reverses the last change made. The undo command can undo the preceding undo command--that is, the first undo command can return the text to its original state, and the second command can reinsert the change, but it can involve several lines. The undo command reverses only a single change. However, after having made more than one change to a line, the line can be restored to its original state with the command U.

Deleted text can be recovered even when the undo operator does not recover it. Recovering lost text is discussed in a separate section.

3.6 Program Editing Features

The editor has a number of commands for program editing. One of the most convenient is the autoindent option, which helps generate correctly indented programs. Another is the shiftwidth option, which is used to reset the backtab value. Both are discussed in Section 6.5.

The operators "<" and ">" are used to shift individual lines left or right, respectively, by one shiftwidth. To shift a line, use the double operators, as shown:

> [<n>]<< shifts the line to the left one shiftwidth, and
> [<n>]>> shifts the line to the right one shiftwidth.

Where n specifies a number of lines; the default is one line.

It is also possible to shift all lines from the cursor to the bottom of the display, either to the left or to the right, respectively. Use the command

    <L

or

    >L

respectively.

Another feature is useful for matching the opening and closing parenthesis in complicated expressions. To see the matching parenthesis, place the cursor at either an opening or a closing parenthesis and strike the percent key (%). This feature also works for braces ({}) and brackets ([]).

For editing programs in C, the double brackets ([[and]]) advance and retreat, respectively, to a line starting with a brace ({)--that is, one function declaration at a time. When the closing double brackets ( ]]) are used with an operator, it stops after a after a line that starts with a brace ({). This is sometimes useful with the command "y", as shown:

    y]]

where the y operator yanks a line, and stores it in a buffer.

3.7 Erase and Line Kill Characters

The most common way to correct input text is to strike CTRL-H to delete an incorrect character, or to strike CTRL-W to delete incorrect words. If the normal system uses the crosshatch as the character erase (#), it works like CTRL-H in vi.

The line kill character is normally one of the following:

@

CTRL-X

CTRL-U

which erases all input on the current line. In general, the kill character does not erase back around an end of line, nor will it erase characters that were not inserted with the current text mode command. To make corrections on the previous line--after a new line has been started--use the following procedure:

1. Strike ESC to terminate input mode.

2. Move the cursor as appropriate to make the correction.

3. Return and continue in input mode. When continuing, the operator "A" is often convenient for appending the current line.

SECTION 4

REARRANGING AND DUPLICATING TEXT

4.1 General

By definition, a sentence ends with a period (.), an exclamation point (!), or a question mark (?); and is followed by either the end of a line, or two spaces. Any number of closing parens, brackets, or quotation marks may appear after the closing punctuation marks, but before the spaces or new line.

The operators (and) move the cursor to the beginning and the end of the previous and next sentences, respectively. Similarly, the operators { and }, and the operators [[and]] move over paragraphs and sections, respectively. The square bracket operators require a double operator entry because they can move the cursor an appreciable distance. While it is easy to return with the back quotation marks '' these commands could still be frustrating if they were easy to execute accidentally.

By definition, a paragraph begins after each empty line, and also at each of a set of paragraph macros. (Refer to the NROFF and TROFF documentation in the ZEUS Programmer's Manual.) The paragraph macros can be changed or extended by assigning a different string to the the paragraphs option in EXINIT. The sentence and paragraph commands can be given counts to operate over groups of sentences and paragraphs. Sections in the editor begin after each macro in the sections option. Section boundaries are always line and paragraph boundaries.

It is possible to look through a large document by using the section commands. It is also possible to use a preceding count with each of the section and paragraph commands. The section commands interpret a preceding count as a different window size in which to redraw the screen display at the new location. This window size is the base size for newly drawn windows until another size is specified. This is useful when looking for a particular section on a slow terminal. It is possible to give the first section command a small count, and then see each successive section heading in a small window.

4.2 Buffers

Vi has the following buffers:

1. A single, unnamed buffer, where the last delete or changed text is saved, and

2. A set of named buffers--a through z--that can be used to save or move text, either within a file, or between files.

The buffers are used by the "yank" and "put" operators described in section 4.3.

4.3 Text Manipulation

The operator (for "yank") is used to place text into the unnamed buffer, or any of the named buffers. The command syntax is

"[<buffer>][<n>]yw

where: " indicates that the following character is a buffer, and not a command
<buffer> is a buffername a through z; default is the unnamed buffer
<n> is the number of words to yank; default is one word
y is the yank operator
w is the word operator

This command does not delete the yanked text. Punctuation marks are counted as words. To yank a complete word, the cursor must be on the first letter of the word. If the cursor is not on the beginning of the word then all characters from the cursor position to next white space (at the end of the word) are yanked.

The operator "yy" is eqivalent to "Y"; the command

"[<buffer>][<n>]Y

yanks the entire line on which the cursor rests, and places it in a buffer, as described above. The count <n> preceding the Y operator yanks n lines of text. The default is one line.

Examples:

The command

yw

yanks the word on which the cursor is located. The command

4yw

yanks the word on which the cursor is located, and the following three words into the unnamed buffer. The command

"a12yw

yanks 12 words into buffer a.

An ordinary delete command saves the text in the unnamed buffer, so that an ordinary put command (p or P, described below) can move it elsewhere. However, the unnamed buffer contents are lost when files are changed; therefore, to change text from one file to another, be sure to use a named buffer.

Text that has been yanked can be reinserted (put) in the text with the operators p or P, where the command syntax is

"[<buffer>]p

where quotation marks and <buffer> indicate the buffername, where the yanked text was stored. The operator "p" reinserts the yanked text after or below the cursor, and the operator "P" reinserts the text before or above the cursor. Command syntax is identical for both P and p operators. If a buffer is not specified, the default is the unnamed buffer.

The text being yanked can be part of a line, or an object such as a sentence that spans more than one line. In this case, when the text is replaced, it is replaced after (or before) the cursor, depending on the command. If the text forms whole lines, then it is returned in whole lines, without changing the current line.

The command

[<n>]YP

yanks a copy of n lines, and then reinserts the same text immediately prior to the current line. The result is that there are two identical text lines and the cursor moves to the top line. The command

[<n>]Yp

is similar, but it copies n lines and places them after (below) the current line, so that there are two identical lines. For example, the command 3YP repeats the line of text three times. The default is one line of text.

The yank command, like the delete and change commands, can be used with a string search. The command

y/<string>/-<n>

yanks the characters from the cursor position to the nth line preceding the string match. Similarly, the command

y/<string>/+<n>

yanks characters from the cursor to the nth line following the string.

The same buffers can be used with the delete operators to move blocks of text within the file or to another file. Moving a block of text requires three operations:

1. delete & store n lines

2. move cursor to the new location

3. "put" the text.

Example:

Delete five lines of text and temporarily store them in buffer a:

"a5dd

The quotation marks indicate a buffername, not the "a" command. Next, move the cursor to the new text location, and enter the command

"ap

or the command

"aP

to insert the text in the new location.

To switch to another file for editing before restoring the yanked text use a command of the form

:e <filename>

where <filename> is the other file to be edited. (These commands are described in a later section.)

NOTE

If the contents of the current editor buffer have been changed, they must be either written back or discarded prior to switching to the other file.

SECTION 5

FILE MANIPULATION

5.1 Writing, Quitting and Editing New Files

The basic write and quit commands are described in section 1.7.

If the text has been changed, but the changes are not to be written to the file, the quit command (:q!) discards the changes. To re-edit the same file (starting over) enter the command:

    :e!

This command is seldom used, because the changes cannot be made after they have been discarded.

To edit a different file without leaving the vi editor enter

    :e <filename>

If the changes have not been written to the file (prior to this command), vi displays the message

    No write since last change (:edit! overrides)

and delays editing the other file. Respond by entering the command

    :w

to save the changes in the first file. After the changes are written, repeat the command ":e <filename>" or use the command

    :e!

to discard the changes in the first file and call the second file. To save changes automatically set the autowrite option. When autowrite is set, use the command

    :n

rather than

    :e

5.2 File Manipulation Commands

Table 5-1 contains the vi file manipulation commands. These commands are followed by a carriage return (RETURN) or an escape (ESC). Most of the commands are self explanatory; however, the following describes how to use these commands.

Table 5-1. File Manipulation Commands

Command	Function
:w	Write changes back to file
:wq	Write changes back and quit
:x	Write, if necessary, and quit
:e<name>	Edit file <name>
:e!	Discard changes and re-edit
:e+<name>	Edit file <name>, starting at end
:e+<n><name>	Edit file <name> starting at line n or with command n
:e#	Edit alternate file, which is designated by the last filename typed before the current filename.
:e%	Edit current file
:w <name>	Write file <name>
:w! <name>	Overwrite file <name>
:<x>,<y>w <name>	Write lines <x> through <y> to <name>
:r <name>	Read file <name> into buffer
:r!<cmd>	Read output of <cmd> into buffer
:n	Edit next file in argument list
:n!	Discard changes to current file, and edit next file
:n <arglist>	Specify new list of arguments <arglist>
:ta <tag>	Edit the file containing the tag <tag>, at <tag>

The basic write command is

```
:w
```

which writes changes to the file. When editing is completed for a single file, write the changes back and terminate vi with the command

    ZZ

For editing long text, it is convenient to write back the changes more frequently with the command ":w" and terminate with the command "ZZ".

When editing more than one file, write back the changes with the command ":w" and start editing a new file with an ":e" command. Another way is to set the autowrite option (see section 6) and use the command

    :n <file>

to fetch the next file for editing. This command is inoperative unless the changes to the current file have been written back.

Whenever changes have been made to the editor's copy of a file, but they are not to be written back, then the exclamation point (!) is added to the command being used. The result is that the editor discards any changes that have been made. For best results, use this command carefully.

The various ":e" commands can be given arguments. The argument "+" starts editing at the end of the file, and the argument

    +<n>

starts the editor at line n. Moreover, n can also be any editor command not containing a space, such as a scan like

    +/<string>

or

    +?<string>

where the editor searches for <string>.

Other arguments for ":e" include the character "%", which, when used in the command, is interpreted as the current file name. Another argument is "#", which is interpreted as an alternate filename, where the alternate filename is the last filename typed other than the current filename. For example, suppose the command

    :e

has been entered, and a diagnostic is returned indicating that the file has not been written. One possibility is to enter the command

:w

which writes the file, and then the command

:e#

to redo the previous ":e". The command

CTRL-↑

performs the same function.

To write a part of a buffer to a file, first determine the line numbers that bound the portion to be written. Use the command

CTRL-g

to display the line numberwhere the cursor is located or set the option number. Then enter the command

:<x>,<y>w <name>

where: <x>,<y> specify the top and bottom line numbers
<name> is the file name of the destination file.

If the destination file does not exist, it will be created; otherwise vi prints the diagnostic message

"<name>" File exists - use "w! <name>" to overwrite

command. Then, instead of line numbers, use the address marks in the command. For example, the command

ma

marks the first line in register a, and

mb

marks the last line in register b. The command

'a,'bw <name>

writes these lines to the file <name>.

It is possible to read another file into the buffer after the current line. Use the command

:r <name>

To edit a set of files in succession, first enter all of the filenames as arguments in the command

:n <name1> <name2> .... <namex>

then edit each one, in turn, using the command

:n

It is also possible to use the command ":n" and specify a pattern to be expanded, such as with an asterisk (*) or a set of characters to match. This can also be done with the initial vi command.

The command

:ta

is very useful for editing large programs. It uses a data base of function names and their locations (which can be created by the program ctags(1).
See the ZEUS Programmer's Reference Manual) for finding a function with a name.
If the ":ta" command requires the editor to switch files, any current work must be written to a file or abandoned prior to switching files.
To relocate a tag, repeat this command without any arguements.

To read in the output from a shell command, use an exclamation point with a shell command <cmd>, as shown:

:!<cmd>

SECTION 6

OPTIONS

6.1 General

As noted previously, the options in the editor ex are also available and easy to use with Vi. The most useful ones are listed in Table 6-1 below.

Table 6-1. Frequently Used Options

Option	Default	Function
autoindent	noai	Automatic indentation
autowrite	noaw	Automatic write before :n, :ta, CTRL-↑, and !
ignorecase	noic	Ignore case in searching
lisp	nolisp	({)} commands deal with S-expressions
list	nolist	Tabs print as CTRL-I; end of lines are marked with $
magic	magic	The characters . [and * are special in scans
number	nonu	Lines are displayed prefixed with line numbers
paragraphs	para= IPLPPPQPP LIbp	Macro names that start paragraphs
redraw	nore	Simulate a smart terminal on a dumb one
scroll	1/2	Number of lines scrolled
sections	sect=NHSHH HU	Macro names that start new sections
shiftwidth	sw=8	Shift distance for <, > and input CTRL-d and CTRL-t
showmatch	nosm	Show matching (or { as) or } is typed
slowopen	noslow	Postpone display updates during inserts
term	adm31	The type of terminal being used
terse	noterse	Shorter error diagnostics
window	speed dependent	Number of lines in display window
wrapmargin	wm=0	Bring right margin in from the right
wrapscan	ws	Wrapping around end-of-file

In general, there are three kinds of options: numeric options, string options, and toggle options. Numeric and string options are set by commands of the form:

```
set <opname>=<val>
```

where: <opname> is the name of the option
<val> is the appropriate string or numeric value for the option

Toggle options can be set or reset, respectively, with the following commands:

```
set <opname>
set no<opname>
```

These options can be entered while in vi by preceding the set command with a colon, and the command can be abbreviated as shown:

```
:se <opname>=<value>
```

or

```
:se <opname>
```

To display a list of those options that have been set, enter the set command without any option name, as shown:

```
:set
```

To display the value of a single option enter the command:

```
:set <opname>?
```

Similarly, to display a list of all possible options and their current values, enter the command

```
:set all
```

Note that the above commands can also be abbreviated, and that multiple options can be placed set using only one option command:

```
:se ai as nu
```

The options that are set during an editing session last only until the editor is exited. However, it may be convenient to have a list of options that are set whenever the editor is used. This can be accomplished by creating a list of ex commands--that is, commands used by the text editor ex--that are to be run every time the programs ex, edit, or vi are

invoked. (Note that all commands that start with a colon are ex commands.) It is good practice to list these commands on a single line.

It is possible to put any number of the option commands in the environment variable EXINIT. When options are set in the environment, then they are automatically set at each entry to vi. For example, to set autoindent, autowrite and terse, the command would be (using csh):

```
setenv EXINIT 'set ai aw terse'
```

6.2 Editing on Slow Terminals

The slow terminal text mode is controlled by the slowopen option. This option is set by the command

```
:se slow
```

On slow systems this option limits the output to the terminal. It is also possible to force the editer to use this option even on faster terminals by using this option. To disable the slowopen option, use the command

```
:se noslow
```

It is also possible to simulate an intelligent terminals with the redraw option. This simulation generates a great deal of output, and is generally tolerable only on lightly loaded systems and fast terminals. This option is set with the command

```
:se redraw
```

and it is cancelled with the command

```
:se noredraw
```

6.3 Ignore Case

The editor will, if commanded, ignore the case of words in the string search. The appropriate command is:

```
:se ic
```

To turn off the ignore case option, use the command

```
:se noic
```

6.4 Magic Characters

Strings used in a string search can contain characters that have "magic" meanings to vi. If this capability is not desired, then reset the magic option with the command

```
:se nomagic
```

With nomagic, only the characters "^" and "$" are special in patterns. The character "\" is also special (as it is almost everywhere in the system), and may be used for an extended pattern matching capability.

With either magic or nomagic, it is necessary to use a "\" (backslash) before a "/" in a forward string search or a "?" in a backward string search. That is, if the string search is for either a "/" (forward) or a "?" (backward), then the character must be preceded by a backslash. Table 6-2 lists the extended forms that are used when the magic option is set.

Table 6-2. Magic Option Extended Operators

Operator	Function
^	At the beginning of a pattern, matches the beginning of a line
$	At the end of a pattern,matches the end of a line
.	Matches any character
\<str	Matches string str at the beginning of a word
str\>	Matches string str at the end of a word
[str]	Matches any single character in the string str
[^str]	Matches any single character not in the string str
[x-y]	Matches any character between x and y, where x and y are alpha-numeric characters
*	Matches any number of the preceding pattern

Note that in the nomagic mode the primitives

. [and *

are used with a preceding "\".

6.5 Autoindent and Shiftwidth

The autoindent option is convenient for generating correctly indented programs. To set the autoindent option, use the command

```
:se ai
```

To demonstrate the operation of the option, open a new line with the letter "o", enter a few tabs, type some characters, and then start another line. The editor supplies white space at the start of the new line, so that it is lined up with the previous line of text. Note that it is not possible to backspace over the automatic indentation.

When the autoindent option is being used, it is sometimes convenient to return to the margin--for example, to place a label at the margin. To defeat the autoindent, use the command

```
CTRL-d
```

which then backspaces over the automatic indent. Each time this command is entered, the cursor backs up one shiftwidth. If the shiftwidth is set to eight, the cursor backs up eight columns. Note that this only works immediately after the supplied autoindent.

To stop all indent, including the next line, strike:

```
0CTRL-d
```

An easy way to place a label at the left margin is to strike the up-arrow (^) and then CTRL-D. The editor moves the cursor to the left margin for one line, and then restores the indent on the next line.

There is normally an eight column left boundary. To reset this boundary, use the shiftwidth option, which is entered by the command

```
:se sw=<n>
```

where <n> is the number of columns that sets the width of the boundary.

6.6 Continuous Text Input

When large amounts of text are being entered, it is often convenient to have lines broken near the right margin automatically. To have the text broken n columns from the right margin, use the command

    :se wm=<n>

If the editor breaks an input line, it can be rejoined with the command

    [<n>]J

where n is the number of lines to be joined. The default is to move the following line to the end of the current line. The editor supplies white space, as appropriate, at the juncture of the joined lines, and leaves the cursor at this white space. To delete the white space use the command "x".

6.7 LISP Editing Options and Commands

The vi editor has some convenient options for editing programs in LISP. The first is the lisp option which is set with the command

    :se lisp

This option changes the parenthesis commands "(" and ")" so that they move backward and forward over s-expressions. The braces-- "{" and "}"--are like the parenthesis commands, but they do not stop at atoms. These commands can be used to skip quickly through a comment, or to the next list.

The autoindent option works differently for LISP. It supplies indent to align at the first argument to the last open list. If there is no such argument, then the indent is two spaces more than the last level

The showmatch option is convenient for typing in LISP. Providing that the opening parenthesis is showing on the display, if a closing parenthesis is typed, the cursor then briefly moves to the position of the opening parenthesis. To set this option, use the command

    :se sm

The vi editor also uses the operator

    =

which realigns existing lines as though they had been typed with the lisp and the autoindent options set. For example, the command

    =%

at the beginning of a function realigns all the lines of the function declaration.

Finally, when editing LISP, the double brackets "[[" and "]]" cause the cursor to advance or retreat, respectively, to lines beginning with an opening parenthesis. This is useful for dealing with entire function definitions.

6.8 Line Numbers

If desired, the editor can place line numbers before each line of text on the display. Use the command

    :se nu

To disable the line number option, use the command

    :se nonu

6.9 Tabs and End of Line Indicators

It is possible to have the display represent tabs as CTRL-I and represent the ends of lines with the symbol "$" by using the list option. Give the command

    :se list

This option can be disabled with the command

    :se nolist

6.10 Automatic Writing of Files

When a file has not been written out prior to changing to a new file, vi prints the diagnostic

    "No write since last change (edit! overrides)".

To have the editor automatically save changes, set the "autowrite" option

    :se aw

To change files, use the command

```
:n
```

instead of

```
:e
```

To disable this option use the command

```
:se noaw
```

6.11 Defining Paragraphs and Sections

There are editor options available to define a paragraph and/or section for NROFF macros (see Section 7 of the Zeus Programmer's Manual). A paragraph normally begins after each empty line; these paragraph boundaries are used by the operators "{" and "}" (see Section 2.4). By setting the "paragraph" option

```
set para=<macro name>
```

where <macro name> is an NROFF macros(s) that defines the start of a paragraph. Similarly, sections can be redefined by using

```
set sections=<macro name>
```

By definition, a section begins after each line with a formfeed CTRL-L in the first column; section boundaries are also line and paragraph boundaries. These boundaries are used by the operators "[[" and "]]" (see Section 2.4).

6.12 Terminal Type

The terminal type is determined from the environment when

```
% setenv TERM <type>
```

was executed (see Section 1.4). This option

```
:se term
```

simply outputs the terminal type.

6.13 Scroll

The amount of scroll when using the CTRL-d, CTRL-u and "z" commands can be altered by issuing

```
:se scroll=<val>
```

where: <val> is the amount of scroll (number of lines)

6.14 Terse

The error diagnostics can be shortened with the command

```
:se terse
```

and lengthened again with

```
:se noterse
```

This is desirable for the more experienced user.

6.15 Window

The number of lines in a text window can be altered with this command

```
:se window=<val>
```

For slow terminals (600 baud or less), the window size is 8; for medium terminals (1200 baud), the size is 16; and for high speed terminals, the full screen size minus 1 is assigned.

6.16 Wrapping Around the End of Files

String searches normally proceed through a file and then continue to search at the beginning. This capacity can be disabled with

```
:se nows
```

 VI

 54

SECTION 7

RECOVERING LOST INPUT

7.1 Lost Lines

The editor saves the last nine blocks of deleted text in a set of registers numbered 1 through 9. Register 1 contails the most recently deleted text. To access any block of deleted text in any register, use the command

"<bufferid>p

or

"<bufferid>P

where: " indicates that a register name is to follow,
<bufferid> is a buffer number in the range 1-9

If, for any reason, this command is not successful, use the undo command, followed by a period:

u.

This repeats the put command.

In general, the period (.) is a command to repeat the last change made. As a special case, when the last change refers to a numbered text register, the period command increments the register number before repeating the command. Thus, a sequence of the form

"1pu.u.u

when the command ".u" is repeated restores to the display each of the deleted block of text stored in the nine buffers. To display all of the deleted text, omit the undo commands, and repeat the period command until the desired text is displayed. It is possible to stop after any period command and keep the text recovered to that point.

7.2 Lost Files

If the system should crash, most of the work in progress can be saved. Following a crash, to access the lost files, first change to the directory in use at the time of the crash. Then invoke vi with the command

vi -r <filename>

where <filename> was the file being edited. See Section 1.4. This usually recovers most of the text up to the point where work was discontinued. In rare cases, some of the lines in the file may be lost. The editor lists the numbers of these lines and the text of the lost lines is replaced by the string "LOST." These lines are almost always among the last few changed. At this point, either discard the changes or replace the few lost lines manually.

To list the files saved, use the command

```
vi -r
```

Current files are saved if the system crashes. If there is more than one crash, there will be one copy saved for each crash. Files are recovered on a last-in, first-out basis. Therefore, to recover an earlier copy of a file, first recover the later copies. The invocation "vi -r" does not always list all saved files, but they can be recovered even if they are not listed.

SECTION 8

MISCELLANEOUS

8.1 Filtering Portions of the Buffer

System commands can be run over portions of the buffer with the operator

    !

For example, it can be used to sort lines in the buffer. As an example, type a list of random words, one per line, and end with a blank line. Then return the cursor to the beginning of the list, and enter the command:

    !}sort

This command sorts the material in the following paragraph alphabetically. The blank line ends the paragraph.

8.2 Typing Non-Printing Characters

In general, to enter non-printing characters (such as control characters) in the file, precede them with

    CTRL-V

This command echoes as an up-arrow (^) on which the cursor rests. This indicates that the next character entered will be a control character. In fact, any character can be inserted in the text except:

1. The null character, CTRL-@
2. The linefeed, CTRL-J, which is used to separate lines in the file.

After vi echoes the up-arrow, any character is treated as a request to insert the corresponding control character. This is the only way to type CTRL-S or CTRL-Q, which are used by the system to suspend and resume output (respectively). These characters are not processed by the editor.

More specifically, to enter the erase or kill characters (for example, the "#" or "@"), simply precede the character with a backslash (\).

APPENDIX A

SPECIFYING TERMINAL TYPE

Before calling vi, the correct terminal type must be entered. The following is an incomplete list of terminals and terminal type numbers that can be entered in vi, as appropriate. Unless indicated by an asterisk (*), the terminals listed here are all intelligent.

Terminal	Code
Hewlett-Packard 2621A/P	2621
Hewlett-Packard 264x	2645
Microterm ACT-IV	act4 *
Microterm ACT-V	act5 *
Lear Siegler ADM-3a	adm3a *
Lear Siegler ADM-31	adm31
Human Design Concept 100	c100
Datamedia 1520	dm1520 *
Datamedia 2500	dm2500
Datamedia 3025	dm3025
Perkin-Elmer Fox	fox *
Hazeltine 1500	h1500
Heathkit h19	h19
Infoton 100	i100
Teleray 1061	t1061
Dec VT-52	vt52 *

To enter the type of terminal, use the command

```
setenv TERM <code>
```

where <code> is the terminal type code listed above.

Example:

The terminal normally supplied with the Zilog System 8000 is the Lear Siegler ADM-31. Use the following command:

```
setenv TERM adm31
```

 VI

 60

APPENDIX B

vi CORRECTION CHARACTERS

Vi correction characters are listed in Table B-1.

Table B-1. Operators Used for Corrections and Changes

Operator	Function
CTRL-H	Deletes the last character input
CTRL-W	Deletes the last word input, as defined by the operator "b"
erase	your system erase character; same as CTRL-H
kill	your system line delete character
\	Escapes a following erase, kill or CTRL-H
ESC	Escape key; terminates text mode
DEL	Delete key; interrupts an text mode operation, terminating it abnormally
RETURN	Carriage return, or more simply, RETURN; starts a new line.
CTRL-D	Backspaces over autoindent
0CTRL-D	Kills all the autoindent
^CTRL-D	Same as above, but restores indent next line
CTRL-V	Quotes the next non-printing character into the file.

APPENDIX C
vi SYMBOL DICTIONARY

This appendix gives the uses the editor makes of each character. The characters are presented in their order in the ASCII character set: Control characters come first, then most special characters, then the digits, upper and then lowercase characters.

The information for each character includes the meaning it has as a command, and any meaning it has during an insert. If it has only meaning as a command, then only this is discussed.

CTRL-@ — Not a command character. If typed as the first character of an insertion it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted the mechanism is not available. A ^@ cannot be part of the file due to the editor implementation.

CTRL-A — Unused.

CTRL-B — Backward window. A count specifies repetition. Two lines of continuity are kept if possible.

CTRL-C — Unused.

CTRL-D — As a command, scrolls down a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future CTRL-D and CTRL-U commands. During an insert, backtabs over _autoindent_ white space at the beginning of a line; this white space cannot be backspaced over.

CTRL-E — Unused.

CTRL-F — Forward window. A count specifies repetition. Two lines of continuity are kept if possible.

CTRL-G — Equivalent to :fCR, printing the current file name, whether it has been modified, the current line number and the number of lines in the file, and the percentage of the way through the file that you are.

CTRL-H (BS) Same as left arrow. (See h.) During an insert, eliminates the last input character, backing over it but not erasing it; it remains so you can see what you typed if you wish to type something only slightly different.

CTRL-I (TAB) Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab character it rests at the last of the spaces which represent the tab. The spacing of tabstops is controlled by the tabstop option.

CTRL-J (LF) Same as down arrow (see j).

CTRL-K Same as up arrow (see k).

CTRL-L Same as right arrow. The ASCII formfeed character, this causes the screen to be cleared and redrawn on dumb terminals. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt.

CTRL-M (RETURN)
A carriage RETURN advances to the next line, at the first non-white position in the line. Given a count, it advances that many lines. During an insert, a RETURN causes the insert to continue onto another line.

CTRL-N Same as down arrow (see j).

CTRL-O Unused.

CTRL-P Same as up arrow (see k).

CTRL-Q Not a command character. In input mode, CTRL-Q quotes the next character, the same as ^V, except that some teletype drivers eat the CTRL-Q so that the editor never sees it.

CTRL-R Same as replacement operator (see r). On hardcopy terminals in _open_ mode, retypes the current line.

CTRL-S Unused. Some teletype drivers use CTRL-S to suspend output until CTRL-Q is typed.

CTRL-T — Not a command character. During an insert, with _autoindent_ set and at the beginning of the line, inserts _shiftwidth_ white space.

CTRL-U — Scrolls the screen up, inverting CTRL-D, which scrolls down. Counts work as they do for CTRL-D, and the previous scroll amount is common to both. On a dumb terminal, CTRL-U will often necessitate clearing and redrawing the screen further back in the file.

CTRL-V — Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file.

CTRL-W — Not a command character. During an insert, backs up as b would in command mode; the deleted characters remain on the display (see CTRL-h).

CTRL-X — Unused.

CTRL-Y — Unused.

CTRL-Z — Redraws the screen.

CTRL-[(ESC) — Cancels a partially formed command, such as a z when no following character has yet been given; terminates inputs on the last line (read by commands such as : / and ?); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. Thus, ESC can be used to stop any function and reenter command mode. The flash or ring indicates that all functions have been stopped, and vi has returned to command mode. Prior to entering insert mode, if there is any doubt about what mode is currently in effect, then press ESC, followed by an insert mode command, such as a. the result is that vi enters insert mode, regardless of the previous mode.

CTRL-\ — Goes to ex.

CTRL-] — Searches for the word which is after the cursor as a tag. Equivalent to typing :ta, this word, and then a RETURN. Mnemonically, this command is "go right to"

CTRL-↑ — Equivalent to :e #. Display returns to the previous position in the last edited file. To edit a file that was specified by this command, and the system response was the diagnostic "No write since last change", enter the command :w. This allows CTRL-↑ to operate. To change files without writing the current underscore file, use the command :e! # instead.

CTRL-_ — Unused. Reserved as the command character for the Tektronix 4025 and 4027 terminal.

SPACE — Same as right arrow (see l).

! — An operator that processes lines from the buffer with reformatting commands. Follow ! with the object to be processed, and then the command name terminated by RETURN. Doubling ! and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the !. Thus 2!}sort sorts the next two paragraphs by running them through the program sort. To read a file or the output of a command into the buffer use :r. To simply execute a command use :!.

— In input mode, if this is the erase character, it deletes the last character typed in input mode. It must be preceded with a \ to insert it, since it normally backs over the last preceding input character.

$ — Moves the cursor to the end of the current line. With a count <n>, the cursor advances to the nth end of line following. For example, 2$ advances the cursor to the end of the following line. With the list option, the end of each line is indicated by a $.

% — Moves to the parentheses or brace {} which balances the parentheses or brace at the current cursor position.

& — Same as :& RETURN; repeats a previous substitution.

" — Precedes a named buffer specification. There are named buffers 1-9 that save deleted text, and named buffers a-z that store "yanked" text.

'

The ' can be used the following ways:

a. When followed by another ', the cursor returns to its previous position, but at the beginning of the line. The previous position is set whenever the cursor is moved from the current line.

b. When the ' is followed by a letter a-z, the cursor returns to the line that was marked with this letter (by the m command),at the first non-white character in the line.

c. When ' is used with a second ' and an operator such as d, the operation takes place over complete lines. Example: d'' deletes the lines between the appropriate marks. Similarly, when used with a `, the operation takes place from the exact marked place to the current cursor position within the line.

(

Retreats to the beginning of a previous sentence, or to the beginning of a LISP s-expression if the lisp option is set. Any number of closing)] " and ' characters may appear after the . ! or ?, and before the spaces or end of line. A count <n> advances n sentences.

)

Advances to the beginning of the next sentence. A count repeats the effect. See (above for the definition of a sentence.

*

Unused.

+

Same as RETURN when used as a command.

,

Reverse of the last f F t or T command, looking the other way in the current line. Especially useful after hitting too many ; characters. A count repeats the search.

-

Retreats to the previous line at the first non-white character. This is the inverse of + and RETURN. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn if this is not possible. If a large amount of scrolling would be required the screen is also cleared and redrawn, with the current line at the center.

. Repeats the last command which changed the buffer. Especially useful when deleting words or lines; use "." to delete more and more words or lines. A count is passed on to the command being repeated. Thus, after 2dw, 3. deletes three words.

/ Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during the input of the bottom line; an (ESC) returns to command state without searching. The search begins with the RETURN which terminates the pattern. The cursor moves to the beginning of the last line to indicate that the search is in progress; the search may then be terminated with a DEL or RUB, or by backspacing when at the beginning of the bottom line. The cursor returns to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer.

When used with an operator the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern, whole lines are affected. To do this, give a pattern with a closing / and then an offset +n or -n.

To include the character / in the search string, escape it with a preceding \. A ↑ at the beginning of the pattern forces the match to occur at the beginning of a line only; this speeds the search. A $ at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available. Unless nomagic is set in the .exrc file, the characters ., [, *, and ~ in the search pattern must be preceded with a \ to get them to work as expected.

0 Moves to the first character of the current line. Also used to form numbers after an initial 1-9.

1-9 Used to form numeric arguments to commands

: A prefix for the commands for file and option manipulation, and for escapes to the system. Input is given on the bottom line and terminated with a RETURN; and the command

then executed. If the colon (:) is hit accidentally, return by hitting DEL (or RUB).

<   Shifts lines left one shiftwidth (normally 8 spaces). Like all operators, affects lines when repeated, as in <<. Counts are passed through to the basic object, thus 3<< shifts three lines.

=   Reindents line for LISP, as though they were typed in with lisp and autoindent set.

>   Shifts lines right one shiftwidth (normally 8 spaces). Affects lines when repeated as in >>. Counts repeat the basic object.

?   Scans backwards; the opposite of /. For details see the / description above.

@   If this is the kill character, escape it with a \ to type it in during input mode, as it normally backs over input on the current line.

A   Appends at the end of line, a synonym for $a

B   Backs up a word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the effect.

C   Changes the rest of the text on the current line; a synonym for c$.

D   Deletes the rest of the text on the current line; a synonym for d$.

E   Moves forward to the end of a word, defined as blanks and non-blanks, like B and W. A count repeats the effect.

F   Finds a single following character, backwards in the current line. A count repeats this search that many times.

G   Goes to the line number given as preceding argument, or the end of the file if no preceding count is given. The screen is redrawn with the new current line in the center if necessary.

H — Home arrow. Homes the cursor to the top line of the screen. If a count <n> is given, then the cursor moves to the nth line of the screen. In any case, the cursor moves to the first non-white character on the line. If used as the target of an operator, full lines are affected.

I — Inserts at the beginning of a line; a synonym for ↑ i.

J — Joins together lines, supplying appropriate white space; one space between words, two spaces after a ., and no spaces at all if the first character of the joined line is). A count causes that many lines to be joined rather than the default two.

K — Unused.

L — Moves the cursor to the first non-white character of the last line on the screen. With a count <n> to the first non-white character on nth line from the bottom. Operators affect whole lines when used with L.

M — Moves the cursor to the middle line on the screen, at the first non-white character on the line.

N — Scans for the next match of the last pattern given to / or ?, but in the reverse direction; this is the reverse of n.

O — Opens a new line above the current line and inputs text there. Terminate with (ESC). A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the *slowopen* option works better.

P — Puts the last deleted text back before/after the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise, the text is inserted between the characters before and at the cursor. May be preceded by a named buffer specification "x to retrieve the contents of the buffer; buffers 1-9 contain deleted material, buffers a-z are available for general use.

Q — Quits from vi to ex command mode. In this mode, whole lines form commands, ending with a RETURN. For all commands; the editor ex prompts with the colon.

R — Replaces characters on the screen with characters you type (overlay fashion). Terminate with (ESC).

S — Changes whole lines, a synonym for cc. A count substitutes for that many lines. The lines are saved in the numeric buffers, and erased on the screen before the substitution begins.

T — Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count <n> repeats the effect n times. Most useful with operators such as d.

U — Restores the current line to its state before you started changing it.

V — Unused.

W — Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/non-blank characters. A count <n> repeats the effect n times.

X — Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.

Y — Yanks a copy of the current line into the unnamed buffer, to be put back by a later p or P; a very useful synonym for yy. Count <n> yanks n lines. May be preceded by a buffer name to put lines in that buffer.

ZZ — Exits the editor (Same as :xRETURN). If any changes have been made, the buffer is written out to the current file. Then the editor quits.

[[— Backs up to the previous section boundary. A section begins at each macro in the sections option, normally a ".NH" or ".SH" and also at lines which start with a formfeed ^L. Lines beginning with { also stop [[; this makes it

useful for looking backwards, a function at a time, in C programs. If the option lisp is set, stops at each (at the beginning of a line, and is thus useful for moving backwards at the top level LISP objects.

\
Unused

]]
Forward to a section boundary; see [[for a definition.

↑
Moves to the first non-white position on the current line.

_
Unused.

`
When the ` is followed by another ` the cursor returns to the previous context. The previous context is set when the cursor is moved from the line. When followed by a letter a-z, returns to the position which was marked with this letter by the m command. When used with an operator such as d, the operation takes place from the exact marked place to the current position within the line. When ' is used, the operation takes place over complete lines. See forward quote (').

a
Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates with (ESC).

b
Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count <n> repeats the effect n times.

c
An operator that changes the following object and replaces it with following input text. The c command must take an object, such as the operator w. Terminated by (ESC). If more than one line is affected, then the previous text is saved in the numeric named buffers. If only part of the current line is affected, then the last character to be changed is marked by $. A count <n> affects

n objects. For example, both 3c) and c3) change the following three sentences.

d — An operator d deletes the following object; an object is an operator like w. If more than part of a line is affected, the text is saved in the numeric buffers. A count <n> affects n objects. Thus 3dw is the sames as d3w.

e — Advances to the end of the next word, defined as for b and w. A count <n> repeats the effect n times.

f — Finds the first instance of the next character following the cursor on the current line. A count <n> repeats n times.

g — Unused.

Arrow keys h, j, k, l, and H.

h — Left arrow. Moves the cursor one character to the left. h and CTRL-H have the same effect. On terminals that send escape sequences (such as vt52, c100, or hp), the arrow keys cannot be used. A count repeats the effect.

i — Inserts text before the cursor; otherwise like a.

j — Down arrow. Moves the cursor one line down in the same column. If the position does not exist, _vi_ comes as close as possible to the same column. synonyms include ^J (linefeed) and ^N.

k — Up arrow. Moves the cursor one line up. ^P is a synonym.

l — Right arrow. Moves the cursor one character to the right. SPACE is a synonym.

m — Marks the current position of the cursor in the mark register, which is specified by the next character a-z. Return to this position or use with an operator using ` or '.

n — Repeats the last string search command.

o Opens new lines below the current line; otherwise like O.

p Puts texts after/below the cursor; otherwise like P.

q Unused.

r Replaces the single character at the cursor with another single character. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see R, which is usually more useful.

s Changes the single character under the cursor to the text that is inserted. Terminate with (ESC). With a count, the count characters on the current line are changed. the last character to be changed is marked with $ (as in c).

t Advances the cursor up to the character before the next character typed. Most useful with operators such as d and c to delete the characters up to a following character. Use . to delete more.

u Undoes the last change made to the current buffer. If repeated, will alternate between these two states. When used after an insert which inserted text on more than one line, the lines are saved in the numeric named buffers.

v Unused.

w Advances to the beginning of the next word, as defined by b.

x Deletes the single character under the cursor. With a count deletes that many characters forward from the cursor position, but only on the current line.

y An operator, yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, "x, the text is placed in that buffer also. Text can be recovered by a later p or P.

z	Redraws the screen with the current line placed as specified by the following character; RETURN specifies the top of the screen, . the center of the screen, and - at the bottom of the screen. A count may be given after the z and before the following character to specify the new screen size for the redraw. A count before the z gives the number of the line to place in the center of the screen instead of the default current line.
{	Retreats to the beginning of the preceding paragraph. A paragraph begins at each macro in the *paragraphs* option, normally, `.IP', `.LP,' `.PP,' `.QP,' and `.bp.' A paragraph also begins after a completely empty line, and each section boundary (see [[above).
|	Places the cursor on the character in the column specified by the count.
}	Advances to the beginning of the next paragraph.
~	Unused.
CTRL-? (DEL)	Interrupts the editor, returning it to command accepting state.

APPENDIX D

vi QUICK REFERENCE

Interrupting and Canceling

File manipulation

```
:w            write back changes
:wq           write and quit
:q            quit
:q!           quit; discard changes
:e name       edit file name
:e!           reedit, discard changes
:e #          edit alternate file (also CTRL-^)
:w name       write file name
:w!           overwrite file name
:!cmd         run cmd, then return
:n            edit next file in arglist
:f            show current file and line number (also CTRL-g)
:sh           escape to shell (CTRL-d) for return)
```

Cursor Positioning within file

```
CTRL-f        forward screenful
CTRL-b        backward screenful
CTRL-d        scroll down half screen
CTRL-u        scroll up half screen
G             goto line (end default)
/<string>     next line matching <string>
?<string>     previous line matching <string>
n             repeat last / or ?
N             reverse last / or ?
/<string>/+n  n`th line after <string>
?<string>?-n  n`th line before <string>
]]            next section/function
[[            previous section/function
%             find matching parenthesis or brace
```

Marking and returning

```
``            return to previous position in text
''            cursor moves to first non-white character
              on the line at the previous position
mx            mark position with letter x
`x            to mark x at position within line
'x            to mark x at first non-white character in line
```

Line positioning

```
H           home window line
L           last window line
M           middle window line
+           next line, at first non-white
-           previous line, at first non-white
RETURN      same as carriage return; moves cursor to
              beginning of next line
j           next line, same column
k           previous line, same column
```

Cursor positioning within line

```
↑           first non white
0           beginning of line
$           end of line
h or ->     forward
l or <-     backwards
CTRL-H      same as <-
space       same as ->
fx          find x forward
Fx          f backward
tx          upto x forward
Tx          back upto x
;           repeat last f F t or T
,           inverse of ;
|           to specified column
```

Words, sentences, paragraphs

```
w           word forward
b           back word
e           end of word
)           beginning of next sentence
}           beginning of next paragraph
(           beginning of previous sentence
{           beginning of previous paragraph
W           blank delimited word
B           back W
E           to end of W
```

Corrections during insert

CTRL-H	erase last character
CTRL-W	erases last word
erase	your erase; same as CTRL-h
kill	your kill; erase input this line
\	escapes CTRL-h; your erase and kill
ESC	ends insertions, back to command
CTRL-?	interrupt, terminates insert
CTRL-D	backtab over *autoindent*
^CTRL-D	kill *autoindent*, for one line only
0CTRL-D	kills all *autoindent*
CTRL-V	quote non-printing character

Insert and replace

a	append after cursor
i	insert before cursor
A	append at end of line
I	insert before first non-blank
o	open line below
O	Open above
rx	replace single character with x
R	replace characters

Operators (double to affect lines)

d	delete
c	change
<	left shift
>	right shift
!	filter through command
=	indent for LISP
y	yank lines to buffer

Miscellaneous operations

C	change rest of line
D	delete rest of line
s	substitute chars
S	substitute lines
J	join lines
x	delete character at cursor
X	delete character before cursor
Y	yank lines

Yank and Put

p	put back line(s) after current line
P	put back line(s) before current line
xp	put from buffer x
"xd	delete into buffer x
"xy	yank to buffer x

Undo, redo, retrieve

u	undo last change
U	restore current line
.	repeat last change
"dp	retrieve d th last delete

Entering/leaving vi

%vi name	edit name at top
ZZ	exit from vi, saving changes

The display

Last line	Error messages, echoing input to :/ ? and !, feed back about i/o and large changes.
@lines	On screen only, not in file (on dumb terminals)
~lines	Lines past end of file
CTRL-x	Control characters, CTRL-? is delete
tabs	expand to spaces, cursor at last

Simple Commands

dw	delete a word
de	delte a word, leave punctuation
dd	delete a line
3dd	delete 3 lines
itextESC	insert text text
cwnewESC	change word to new
easESC	pluralize word
xp	transpose characters

YACC*

YET ANOTHER COMPILER-COMPILER

* This information is based on an article originally written by Stephen C. Johnson, Bell Laboratories.

PREFACE

This document describes the basic process of preparing a Yacc specification. Section 1 gives an introduction to Yacc, Section 2 describes the preparation of grammar rules, Section 3 the preparation of the user-supplied actions associated with these rules, and Section 4 the preparation of lexical analyzers. Section 5 describes the operation of the parser. Section 6 discusses various reasons why Yacc may be unable to produce a parser from a specification, and how to solve them. Section 7 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 8 discusses error detection and recovery. Section 9 discusses the operating environment and special features of the parsers Yacc produces. Section 10 gives some suggestions that should improve the style, Section 11 discusses some advanced topics, and Section 12 gives acknowledgments.

Appendix A gives a summary of the Yacc input syntax and Appendix B has a brief example. Appendix C gives an example using some of the more advanced features of Yacc. Appendix D describes mechanisms and syntax no longer actively supported, but is provided for historical continuity with older versions of Yacc.

TABLE OF CONTENTS

SECTION 1 INTRODUCTION 5

SECTION 2 BASIC SPECIFICATIONS 8

SECTION 3 ACTIONS 11

SECTION 4 LEXICAL ANALYSIS 14

SECTION 5 HOW THE PARSER WORKS 16

SECTION 6 AMBIGUITY AND CONFLICTS 21

SECTION 7 PRECEDENCE 26

SECTION 8 ERROR HANDLING 29

SECTION 9 THE YACC ENVIRONMENT 32

SECTION 10 HINTS FOR PREPARING SPECIFICATIONS 34

10.1 General 34
10.2 Input Style 34
10.3 Left Recursion 34
10.4 Lexical Tie-Ins 35
10.5 Reserved Words 36

SECTION 11 ADVANCED TOPICS 37

11.1 Simulating Error and Accept in Actions 37
11.2 Accessing Values in Enclosing Rules ... 37
11.3 Support for Arbitrary Value Types 38

APPENDIX A YACC INPUT SYNTAX 40

APPENDIX B A SIMPLE EXAMPLE 43

TABLE OF CONTENTS (continued)

APPENDIX C AN ADVANCED EXAMPLE 46

APPENDIX D OLD FEATURES 52

SECTION 1

INTRODUCTION

Yacc is a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process. This includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (tokens) from the input stream. These tokens are organized according to the input structure rules (grammar rules). When one of these rules has been recognized, user code supplied for this rule (an action) is invoked. Actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of the C Programming Language, and the actions and output subroutine are in C as well. Many of the syntactic conventions of Yacc also follow C.

The input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule is

```
date  :  month_name  day  ','  year   ;
```

Here, _date_, _month name_, _day_, and _year_ represent structures of interest in the input process; _month name_, _day_, and _year_ are defined elsewhere. The comma (,) is enclosed in single quotes to imply that it is to appear in the input. The colon and semicolon serve as punctuation in the rule and have no significance in controlling the input. Thus, with proper definitions, the input

```
July  4, 1776
```

is matched by this rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the low-level structures, and communicates these tokens to the parser. A structure recognized by the lexical analyzer is called a terminal symbol, or token, and the structure recognized by the parser is called a nonterminal symbol.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name  :  'J' 'a' 'n'   ;
month_name  :  'F' 'e' 'b'   ;

          . . .

month_name  :  'D' 'e' 'c'   ;
```

can be used in the previous example. The lexical analyzer recognizes only individual letters, and month name is a nonterminal symbol. Such low-level rules waste time and space, and can complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer recognizes the month names, and returns an indication that a month name was seen; in this case, month name is a token.

Literal characters such as , must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the previous example the rule

```
date  :  month '/' day '/' year   ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule can be inserted into a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. The resulting input errors are detected early with a left-to-right scan. Thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data is usually found quickly. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The

former cases represent design errors; the latter cases are often corrected by making the lexical analyzer more powerful or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems. The constructions that are difficult for Yacc to handle are also frequently difficult for human beings to handle. The discipline of formulating valid Yacc specifications often reveals errors of design early in the program development.

SECTION 2

BASIC SPECIFICATIONS

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, it is useful to include the lexical analyzer and other programs as part of the specification file. Thus, every specification file consists of three sections: the _declarations_, (_grammar_) _rules_, and _programs_. The sections are separated by double percent (%%) marks. (The single percent (%) is generally used in Yacc specifications as an escape character.) In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section can be empty. If the programs section is omitted, the second %% mark is omitted. Thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and new lines are ignored except that they cannot appear in names or multicharacter reserved symbols. Comments can appear wherever a name is legal, and are enclosed in /* . . . */, as in C language and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A  :  BODY  ;
```

_A_ represents a nonterminal name, and _BODY_ represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names have arbitrary length, and can be made up of letters, dot (.), underscore (_), and noninitial digits. Upper and lowercase letters are distinct. The names used in the body of a grammar rule can represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes (' '). As in C, the backslash (\) is an escape character within literals, and all the C escapes are recognized.

```
'\n' new line
'\r' return
'\'' single quote (')
'\\' backslash (\)
'\t' tab
'\b' backspace
'\f' form feed
'\xxx'    "xxx" in octal
```

The NUL character ('\0' or 0) is never used in grammar rules.

If there are several grammar rules with the same left side, the vertical bar (|) can be used to avoid rewriting the left side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A    :    B  C  D   ;
A    :    E  F   ;
A    :    G   ;
```

can be given to Yacc as

```
A    :    B  C  D
     |    E  F
     |    G
     ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated as:

```
empty :   ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token   name1  name2 . . .
```

in the declarations (Sections 4, 6, and 7). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, the start symbol has particular importance because it is recognized by the parser. This symbol represents the largest, most general structure described by the grammar rules. By default, the start

symbol is the left side of the first grammar rule in the rules section. It is recommended to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start   symbol
```

The end of the input to the parser is signaled by a special token called the endmarker. If the tokens up to, but not including, the endmarker form a structure that matches the start symbol, the parser function returns to its caller after the endmarker is seen and accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate (Section 4). Usually, the endmarker represents some I/O status, such as "end-of-file" or "end-of-record."

SECTION 3

ACTIONS

With each grammar rule, there are associated actions to be performed each time the rule is recognized in the input process. These actions can return values and can obtain the values returned by previous actions. The lexical analyzer can also return values for tokens.

An action is an arbitrary C statement and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements enclosed in braces ({ and }). For example,

```
A    :    '('  B  ')'
               {    hello( 1, "abc" );  }
```

and

```
XXX  :    YYY  ZZZ
               {    printf("a message\n");
                    flag = 25;    }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are slightly altered. The symbol dollar sign ($) is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudovariable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action uses the pseudovariables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule. Thus, if the rule is

```
A    :    B  C  D   ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As another example, consider the rule:

```
expr :    '('  expr  ')'   ;
```

The value returned by this rule is the value of the expr in parentheses. This can be indicated by

```
expr :     '('  expr  ')'          {  $$ = $2 ;  }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A    :    B    ;
```

frequently do not need to have an explicit action.

In the previous examples, all the actions come at the end of their rules. Sometimes it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule returns a value, accessible through the usual mechanism by the actions to the right of it. In turn, it can access the values returned by the symbols to its left. Thus, in the rule

```
A    :    B
               {  $$ = 1;  }
          C
               {  x = $2;    y = $3;  }
     ;
```

the effect is to set x to 1, and set y to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered by recognizing this added rule. Yacc treats the previous example as if it had been written:

```
$ACT :    /* empty */
               {  $$ = 1;  }
     ;

A    :    B  $ACT  C
               {  x = $2;    y = $3;  }
     ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function node, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, then returns the index of the newly created node. The parse tree is built by supplying actions such as:

```
expr :    expr  '+'  expr
                {  $$ = node( '+', $1, $3 );  }
```

in the specification.

Other variables can be defined for the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks %{ and %}. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   int variable = 0;   %}
```

can be placed in the declarations section, making _variable_ accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; such names must be avoided.

In these examples, all the values are integers. A discussion of other value types is found in Section 11.

SECTION 4

LEXICAL ANALYSIS

A lexical analyzer must be supplied to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The function returns an integer (the token number), representing the kind of token read. If there is a value associated with that token, it must be assigned to the external variable yylval.

The parser and the lexical analyzer must agree on these token numbers for communication between them to take place. The numbers are chosen by Yacc or by the user. In either case, the "# define" mechanism of C allows the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer looks like the following code:

```
yylex(){
     extern int yylval;
     int c;
     . . .
     c = getchar();
     . . .
     switch( c ) {
          . . .
     case '0':
     case '1':
       . . .
     case '9':
          yylval = c-'0';
          return( DIGIT );
          . . .
          }
     . . .
```

The intent is to return a token number of DIGIT and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT is defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers. In the grammar, avoid using any token names that are reserved or significant in C or the parser. For example, the use of token names <u>if</u> or <u>while</u> causes severe difficulties when the lexical analyzer is compiled. The token

name *error* is reserved for error handling and must not be used (Section 8).

In the default situation, the token numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), follow the first appearance of the token name or literal in the declarations section with a nonnegative integer. This integer is the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. All token numbers must be distinct.

The endmarker must have token number 0 or a negative number. This token number cannot be redefined by the user; thus, all lexical analyzers must be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *lex* program. These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules.

SECTION 5

HOW THE PARSER WORKS

Yacc turns the specification file into a C program that parses the input according to the specification given. The parser itself is relatively simple, and understanding how it works makes treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and retaining the next input token, called the lookahead token. The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it; they are called shift, reduce, accept, and error. Movement of the parser is done as follows:

1. Based on its current state, the parser determines whether it needs a lookahead token to determine what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser determines its next action and carries it out. This results in states being pushed onto the stack or popped off the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there is an action:

```
IF   shift 34
```

which means in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are taken when the parser has seen the right side of a grammar rule and is prepared to announce that it has seen an instance of the rule, replacing the

right side with the left side. It may be necessary to consult the lookahead token to decide whether to reduce. This is not usually the case, since the default action (represented by a .) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, which can lead to some confusion. The action

```
    .    reduce 18
```

refers to grammar rule 18, while the action

```
    IF   shift 34
```

refers to state 34.

Suppose the rule being reduced is

```
    A    :    x  y  z    ;
```

The reduce action depends on the left symbol (A in this case), and the number of symbols on the right side (three in this case). To reduce, first pop off the top three states from the stack. (The number of states popped equals the number of symbols on the right side of the rule.) After popping these states, a state is uncovered that is the state the parser was in before beginning to process the rule. Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left symbol (called a goto action) and an ordinary shift of a token. The lookahead token is cleared by a shift, and is not affected by a goto. The uncovered state contains an entry such as:

```
    A    goto 20
```

causing state 20 to be pushed onto the stack and become the current state.

In effect, the reduce action "turns back the clock" in the parser, popping the states off the stack to go back to the state where the right side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right side of the rule is empty, no states are popped off the stack; the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced,

the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable yylval is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable yyval is copied onto the value stack. The pseudovariables $1, $2, etc. refer to the value stack.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that results in a legal input. The parser reports an error, and attempts to recover and resume parsing. The error recovery (as opposed to the detection of error) is discussed in Section 8.

Consider the specification

```
%token  DING  DONG  DELL
%%
rhyme     :    sound  place
     ;
sound     :    DING  DONG
     ;
place     :    DELL
     ;
```

When Yacc is invoked with the -v option, a file called y.output is produced, with a human-readable description of the parser. The y.output file corresponding to this grammar (with some statistics stripped off the end) is:

```
state 0
     $accept  :  _rhyme  $end

     DING  shift 3
     .  error

     rhyme  goto 1
     sound  goto 2

state 1
     $accept  :   rhyme_$end
```

```
        $end  accept
        .  error

    state 2
        rhyme   :   sound_place

        DELL  shift 5
        .  error

        place   goto 4

    state 3
        sound    :   DING_DONG

        DONG  shift 6
        .  error

    state 4
        rhyme   :   sound  place_    (1)

        .   reduce  1

    state 5
        place  :   DELL_    (3)

        .   reduce  3

    state 6
        sound    :   DING  DONG_    (2)

        .   reduce  2
```

In addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character indicates what has been seen and what is yet to come in each rule. Suppose the input is

```
    DING  DONG  DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser refers to the input to select among the actions available in state 0, so the first token (DING) is read, becoming the lookahead token. The action in state 0 on DING is "shift 3," so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, DONG, is read, becoming the lookahead token. The action in state 3 on the token DONG is "shift 6," so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even

consulting the lookahead token, the parser reduces by rule 2.

```
sound  :  DING  DONG
```

This rule has two symbols on the right side, so two states, 6 and 3, are popped off the stack; uncovering state 0. Consulting the description of state 0, looking for a goto on _sound_,

```
sound    goto 2
```

is obtained; thus, state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, DELL, must be read. The action is "shift 5," so state 5 is pushed onto the stack (which now has 0, 2, and 5 on it) and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right side, so one state (5) is popped off, and state 2 is uncovered. The goto in state 2 on _place_, the left side of rule 3, is state 4. Now the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on _rhyme_ causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the _y.output_ file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

Consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, etc. A few minutes spent with this and other simple examples will be repaid when problems arise in more complicated contexts.

SECTION 6

AMBIGUITY AND CONFLICTS

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr :    expr  '-'  expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr  -  expr  -  expr
```

the rule allows this input to be structured as either

```
(  expr  -  expr  )  -  expr
```

or as

```
expr  -  (  expr  -  expr  )
```

The first is called left association, and the second is called right association.

Yacc detects such ambiguities when it is attempting to build the parser. Consider the problem that confronts the parser when it is given an input such as

```
expr  -  expr  -  expr
```

When the parser has read the second expr, the input that it has seen:

```
expr  -  expr
```

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying the rule, the input is reduced to <u>expr</u> (the left side of the rule). The parser then reads the final part of the input:

```
-  expr
```

and again reduces. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr - expr
```

it defers the immediate application of the rule, and continues reading the input until it had seen

```
expr - expr - expr
```

It then applies the rule to the rightmost three symbols, reducing them to _expr_ and leaving

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a shift/reduce conflict. It may also happen that the parser has a choice of two legal reductions; this is called a reduce/reduce conflict. There are never any shift/shift conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.

2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule in the input sequence.

Rule 1 implies that reductions are deferred in favor of shifts whenever there is a choice. Rule 2 gives rather crude control over the behavior of the parser, but reduce/reduce conflicts should be avoided.

Conflicts arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc constructs. The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of

disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

Whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. This rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc produces parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
stat :    IF  '('  cond  ')'  stat
     |    IF  '('  cond  ')'  stat  ELSE  stat
     ;
```

In these rules, _IF_ and _ELSE_ are tokens, _cond_ is a nonterminal symbol describing conditional (logical) expressions, and _stat_ is a nonterminal symbol describing statements. The first rule is called the _simple_-_if_ rule, and the second is called the _if_-_else_ rule.

These two rules form an ambiguous construction, since input of the form

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be structured according to these rules in two ways:

```
IF  (  C1  )  {
     IF  (  C2  )  S1
     }
ELSE  S2
```

or

```
IF  (  C1  )  {
     IF  (  C2  )  S1
     ELSE  S2
     }
```

The second interpretation is the one given in most programming languages having this construct. Each _ELSE_ is associated with the last preceding "un-_ELSE_'d" _IF_. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the ELSE. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of groupings of the input.

On the other hand, the ELSE can be shifted, S2 read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

is reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which is reduced by the simple-if rule. This leads to the second of the groupings of the input, which is usually desired.

The parser can do two valid things--there is a shift/reduce conflict. The application of Disambiguating Rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

There can be many conflicts, each associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (-v) option output file. For example, the output corresponding to the conflict state is:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

        stat  :  IF  (  cond  )  stat_          (18)
        stat  :  IF  (  cond  )  stat_ELSE  stat

       ELSE      shift 45
       .         reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules that has been seen. Here, in state 23, the parser has seen input corresponding to

```
IF  (  cond  )  stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is ELSE, it shifts into state 45. State 45 has as part of its description the line

```
stat  :  IF  (  cond  )  stat  ELSE_stat
```

since the ELSE has been shifted in this state. Back in state 23, the alternative action (described by .) is to be done if the input symbol is not mentioned explicitly in the above actions. In this case, if the input symbol is not ELSE, the parser reduces by Grammar Rule 18:

```
stat  :  IF  '('  cond  ')'  stat
```

The numbers following shift commands refer to other states, while the numbers following reduce commands refer to grammar rule numbers. In the y.output file, the rule numbers are printed after those rules that can be reduced. In most states, there is at most one reduce action possible in the state, and this is the default command. The user who encounters unexpected shift/reduce conflicts should look at the verbose output to decide whether the default actions are appropriate.

SECTION 7

PRECEDENCE

The rules given for resolving conflicts are not sufficient in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. Ambiguous grammars with appropriate disambiguating rules can create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. Writing grammar rules of the form

```
expr  :  expr  OP  expr
```

and

```
expr  :  UNARY  expr
```

for all binary and unary operators creates a very ambiguous grammar with many parsing conflicts. The precedence, or binding strength, of all the operators, and the associativity of the binary operators must be specified as disambiguating rules. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules and to construct a parser that recognizes the desired precedence levels and associative properties.

The precedence levels and associative properties are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword (%left, %right, or %nonassoc) followed by a list of tokens. All the tokens on the same line are assumed to have the same precedence level and associativity. The lines are listed in order of increasing precedence or binding strength. Thus, the statements

```
%left  '+'  '-'
%left  '*'  '/'
```

describe the precedence level and associative properties of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators that cannot associate with themselves.

As an example of the behavior of these declarations, the description

```
%right  '='
%left  '+'  '-'
%left  '*'  '/'

%%

expr :    expr  '='  expr
     |    expr  '+'  expr
     |    expr  '-'  expr
     |    expr  '*'  expr
     |    expr  '/'  expr
     |    NAME
     ;
```

is used to structure the input

```
a  =  b  =  c*d  -  e  -  f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must be given a precedence level. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary minus (-); unary minus is given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication, use the following statements:

```
%left  '+'  '-'
%left  '*'  '/'
%%
expr :    expr  '+'  expr
     |    expr  '-'  expr
     |    expr  '*'  expr
     |    expr  '/'  expr
     |    '-'  expr      %prec  '*'
     |    NAME
     ;
```

A token declared by %left, %right, and %nonassoc need not be, but can be, declared by %token as well.

The precedence level and associativity are used by Yacc to resolve parsing conflicts and to give rise to disambiguating rules. Formally, the rules work as follows:

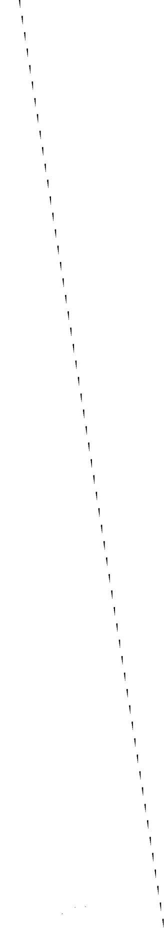
1. The precedence level and associativity properties are recorded for those tokens and literals that have them.

2. Some grammar rules have no precedence and associativity associated with them. In this is the case, the precedence and associativity of the last token or literal in the body of the rule are associated with the grammar rule by default. If the %prec construction is used, it overrides this default.

3. When there is a reduce/reduce conflict or a shift/reduce conflict, and either the input symbol or the grammar rule has no precedence level and associativity, the two disambiguating rules given the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence level and associativity connected to them, the conflict is resolved in favor of the action (shift or reduce) related to the higher precedence level. If the precedence levels are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence levels are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences can disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in "cookbook" fashion until some experience has been gained. Also, the y.output file is very useful in deciding whether the parser is actually doing what was intended.



SECTION 8

ERROR HANDLING

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it is often necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser restarted after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow some control over this process, Yacc provides a simple, but reasonably general feature: the token name "error." This name is reserved for error handling and can be used in grammar rules. It suggests places where errors are expected and recovery can take place. The parser pops the stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is deleted.

As an example, a rule of the form

```
stat :    error
```

means that on a syntax error, the parser skips over the statement in which the error was seen. More precisely, the parser scans ahead, looking for three tokens that legally follow a statement, and starts processing at the first of these. If the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement and report a second error where there is no error.

Actions can be used with these special error rules. These actions attempt to do such things as reinitialize tables and reclaim symbol table space.

Such error rules are very general but difficult to control. An easier error form is:

```
stat :   error
```

When there is an error, the parser skips over the statement, but does so by skipping to the next ; character. All tokens after the error and before the next ; cannot be shifted, and they are discarded.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule is

```
input  :  error  '\n'
          {  printf("Reenter last line:");  }  input
             {  $$  =  $4;                        }
```

The problem with this approach is that the parser must correctly process three input tokens before it correctly resynchronizes after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message; this is unacceptable. For this reason, there is a mechanism that forces the parser to function as though full error recovery has taken place. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input     :    error  '\n'
               {    yyerrok;
                    printf( "Reenter last line: " );    }
          input
               {    $$  =  $4;  }
     ;
```

The token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action produces this effect. For example, suppose the action after error is to call some sophisticated resynchronization routine, supplied by the user, that attempts to advance the input to the beginning of the next valid

statement. After this routine is called, the next token returned by yylex is, presumably, the first token in a legal statement. The old, illegal token must be discarded, and the error state reset. This is done by a rule like

```
stat :    error
               {    resynch();
                    yyerrok ;
                    yyclearin ;   }
        ;
```

These mechanisms allow for a simple, fairly effective recovery of the parser from many errors. The error actions required by other portions of the program can also be controlled.

SECTION 9

THE YACC ENVIRONMENT

When the user inputs a specification to Yacc, the output is a file of C programs called y.tab.c on most systems (the names can differ from installation to installation). The integer-valued function produced by Yacc is called yyparse. When it is called, it in turn repeatedly calls yylex, the lexical analyzer supplied by the user (Section 4) to obtain input tokens. Eventually, an error is detected and, if no error recovery is possible, yyparse returns the value 1. Otherwise, the lexical analyzer returns the endmarker token, and yyparse returns the value 0.

A certain amount of environment for this parser must be provided to obtain a working program. For example, as with every C program, a program called main must be defined, which eventually calls yyparse. In addition, a routine called yyerror prints a message when a syntax error is detected.

These two routines must be supplied by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of main and yyerror. The name of this library is system dependent; on many systems the library is accessed by a -ly argument to the loader. The source for these default programs is given here:

```
main(){
     return( yyparse() );
     }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
     fprintf( stderr, "%s\n", s );
     }
```

The argument to yyerror is a string containing an error message, usually the string "syntax error." The program must keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable yychar contains the lookahead token number at the time the error was detected; this gives better diagnostics. Since the main program is probably supplied by the user (to read arguments, etc.), the Yacc library is useful only in small projects or in the earliest stages of larger ones.

The external integer variable yydebug is normally set to 0. If it is set to a nonzero value, the parser outputs a verbose description of its actions, including a discussion of which input symbols have been read and what the parser actions are.

SECTION 10

HINTS FOR PREPARING SPECIFICATIONS

10.1 General

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are independent.

10.2 Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following are some hints:

1. Use all capital letters for token names, all lowercase letters for nonterminal names. This helps isolate the source of problems.

2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

3. Put together all rules with the same left side. Put the left side in only once, and let all following rules begin with a vertical bar.

4. Put a semicolon only after the last rule with a given left side, and put the semicolon on a separate line. This allows new rules to be easily added.

5. Indent rule bodies by two tab stops and action bodies by three tab stops.

The example in Appendix B is written following this style, as are the examples in the text of this document. The user must decide how to make the rules visible through the bulk of action code.

10.3 Left Recursion

The algorithm used by the Yacc parser encourages "left recursive" grammar rules of the form

```
name :    name  rest_of_rule  ;
```

These rules frequently arise when specifications of sequences and lists are written:

```
list :   item
     |   list  ','  item
     ;
```

and

```
seq  :   item
     |   seq  item
     ;
```

In each case, the first rule is reduced for the first item only, the second rule is reduced for the second and all succeeding items.

With right recursive rules such as

```
seq  :   item
     |   item  seq
     ;
```

the parser is a bit bigger, and the items are seen and reduced from right to left. An internal stack in the parser is in danger of overflowing if a very long sequence is read. Thus, left recursion must be used.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq  :   /* empty */
     |   seq  item
     ;
```

Once again, the first rule is always reduced once before the first item is read, then the second rule is reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts arise if Yacc is asked to decide which empty sequence it has seen when it has not seen enough to know.

10.4 Lexical Tie-Ins

Some lexical decisions depend on context. For example, the lexical analyzer deletes blanks normally, but not within quoted strings. Names can be entered in a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer and set by actions. For example, suppose a program consists of zero or more declarations followed by zero or more statements.

Consider the statements:

```
%{
        int dflag;
%}
  ...  other declarations ...

%%

prog :    decls  stats
     ;

decls      :    /* empty */
                {    dflag = 1;  }
     |    decls  declaration
     ;

stats      :    /* empty */
                {    dflag = 0;  }
     |    stats  statement
     ;

     ...  other rules ...
```

The flag dflag is now 0 when reading statements, and 1 when reading declarations, except for the first token in the first statement. The parser must see this token before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

10.5 Reserved Words

Some programming languages permit the use of words normally reversed as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of `if' is a keyword, and that instance is a variable." Therefore, do not use keywords.

SECTION 11

ADVANCED TOPICS

11.1 Simulating Error and Accept in Actions

The parsing actions of error and accept are simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes yyparse to return the value 0. YYERROR causes the parser to behave as if the current input symbol had been a syntax error; yyerror is called, and error recovery takes place. These mechanisms are used to simulate parsers with multiple endmarker or context-sensitive syntax checking.

11.2 Accessing Values in Enclosing Rules

An action can refer to values returned by actions to the left of the current rule. The mechanism is the same as with ordinary actions: a dollar sign followed by a digit, but in this case the digit can be zero or negative. Consider the commands

```
sent :    adj  noun  verb  adj  noun
               {  look at the sentence . . .  }
     ;

adj  :    THE         {     $$ = THE;  }
     |    YOUNG       {     $$ = YOUNG;  }
     . . .
     ;

noun :    DOG
               {     $$ = DOG;  }
     |    CRONE
               {     if( $0 == YOUNG ){
                          printf( "what?\n" );
                          }
                     $$ = CRONE;
                     }
     ;
     . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. This is only possible when a great deal is known about what might precede the symbol noun in the input. The mechanism saves a great deal of trouble, especially when a few combinations are excluded from an otherwise regular structure.

11.3 Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc also supports values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser is strictly type checked. The Yacc value stack is declared to be a union of the various types of values desired, and union member names are associated with each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc automatically inserts the appropriate union name so that no unwanted conversions take place. In addition, type-checking commands such as Lint are more silent.

There are three mechanisms to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs (notably the lexical analyzer) must be informed of the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc cannot easily determine the type.

To declare the union, the following lines must be included in the declaration section:

```
%union {
     body of union ...
     }
```

This declares the Yacc value stack and the external variables yylval and yyval to have type equal to this union. If Yacc was invoked with the -d option, the union declaration is copied onto the y.tab.h file. Alternatively, the union can be declared in a header file, and a typedef can be used to define the variable YYSTYPE to represent this union. Thus, the header file can also contain:

```
typedef union {
     body of union ...
     } YYSTYPE;
```

The header file must be included in the declarations section by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

indicates a union member name. If this follows one of the keywords %token, %left, %right, or %nonassoc, the union member name is associated with the tokens listed. Thus, entering

```
%left  <optype>  '+'  '-'
```

causes any reference to values returned by these two tokens to be tagged with the union member name _optype_. Another keyword, %type, is used similarly to associate union member names with nonterminals, as with

```
%type  <nodetype>  expr  stat
```

There are several cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no "a priori" type. Similarly, reference to left context values (such as $0 in the previous subsection) leaves Yacc with no way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between < and >, immediately after the first $. An example of this usage is

```
rule :    aaa  {  $<intval>$  =  3;  } bbb
               {    fun( $<intval>2, $<other>0 );  }
     ;
```

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type turns on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold _int_'_s_.

APPENDIX A

YACC INPUT SYNTAX

This Appendix has a description of the Yacc input syntax as a Yacc specification. Such items as context dependencies are not considered. The Yacc input specification language is most naturally specified as an LR grammar; the cumbersome part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule that has an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and determines whether the next token is a colon. If so, it returns the token C_IDENTIFIER; otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
                /*  grammar for the input to Yacc   */

        /*  basic entities   */
%token  IDENTIFIER
        /*   includes identifiers and literals   */
%token  C_IDENTIFIER
        /*    identifier (but not literal) */
        /*    followed by colon   */
%token  NUMBER
        /*    [0-9]+    */

     /*     reserved words   */
        /*    %type => TYPE, %left => LEFT, etc.   */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token    MARK /*  the  %%  mark   */
%token    LCURL     /*  the  %{  mark   */
%token    RCURL     /*  the  %}  mark   */

     /*  ascii character literals stand for themselves   */

%start    spec

%%

spec :    defs  MARK  rules  tail
     ;

tail :    MARK {In this action, eat up the rest of the file}
     |    /*  empty:  the second MARK is optional   */
```

```
        ;

defs :    /*  empty  */
     |    defs  def
     ;

def  :    START  IDENTIFIER
     |    UNION  {  Copy union definition to output  }
     |    LCURL  {  Copy C code to output file  }  RCURL
     |    ndefs  rword tag nlist
     ;

rword     :    TOKEN
     |    LEFT
     |    RIGHT
     |    NONASSOC
     |    TYPE
     ;

tag  :    /*  empty:  union tag is optional  */
     |    '<'  IDENTIFIER  '>'
     ;

nlist     :    nmno
     |    nlist  nmno
     |    nlist  ','  nmno
     ;

nmno      :         IDENTIFIER
                    /*  NOTE: literal illegal with %type  */
          |         IDENTIFIER NUMBER
                    /*  NOTE: illegal with %type  */
     ;

     /*  rules  section  */

rules     :    C_IDENTIFIER  rbody  prec
     |    rules  rule
     ;

rule :    C_IDENTIFIER  rbody  prec
     |    '|'  rbody  prec
     ;

rbody     :    /*  empty  */
     |    rbody  IDENTIFIER
     |    rbody  act
     ;

act  :    '{' { Copy action, translate $$, etc. }  '}'
     ;
```

```
prec :    /*  empty  */
     |    PREC  IDENTIFIER
     |    PREC  IDENTIFIER  act
     |    prec  ';'
     ;
```

APPENDIX B

A SIMPLE EXAMPLE

This example gives the complete Yacc specification for a small desk calculator. The desk calculator has 26 registers, labeled "a" through "z," and accepts arithmetic expressions made up of the operators +, -, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in the C language, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and of demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. The way that decimal and octal integers are read in by the grammar rules is primitive; this job is better done by the lexical analyzer.

```
%{
#  include  <stdio.h>
#  include  <ctype.h>

int  regs[26];
int  base;

%}

%start  list

%token  DIGIT  LETTER

%left  '|'
%left  '&'
%left  '+'  '-'
%left  '*'  '/'  '%'
%left  UMINUS
       /*  supplies  precedence  for  unary  minus  */

%%      /*  beginning  of  rules  section  */

list :    /*  empty  */
     |    list  stat  '\n'
     |    list  error  '\n'
               {    yyerrok;  }
```

```
        ;

stat :      expr
                {     printf( "%d\n", $1 );   }
        |   LETTER  '='  expr
                {     regs[$1]   =   $3;   }
        ;

expr :      '('   expr   ')'
                {     $$   =   $2;   }
        |   expr   '+'   expr
                {     $$   =   $1   +   $3;   }
        |   expr   '-'   expr
                {     $$   =   $1   -   $3;   }
        |   expr   '*'   expr
                {     $$   =   $1   *   $3;   }
        |   expr   '/'   expr
                {     $$   =   $1   /   $3;   }
        |   expr   '%'   expr
                {     $$   =   $1   %   $3;   }
        |   expr   '&'   expr
                {     $$   =   $1   &   $3;   }
        |   expr   '|'   expr
                {     $$   =   $1   |   $3;   }
        |   '-'   expr          %prec   UMINUS
                {     $$   =   -   $2;   }
        |   LETTER
                {     $$   =   regs[$1];   }
        |   number
        ;

number     :    DIGIT
        {   $$ = $1;    base   =   ($1==0)   ?   8   :   10;   }
        |   number  DIGIT
                {     $$   =   base * $1   +   $2;   }
        ;

%%      /*  start  of  programs  */

yylex() {  /*  lexical analysis  routine  */
           /*  returns LETTER for a lower case letter */
           /*  yylval = 0  through  25  */
           /*  return  DIGIT  for  a  digit */
           /*  yylval = 0  through  9  */
           /*  all other characters  */
           /*  are returned immediately  */

        int  c;

        while( (c=getchar()) == ' ' ) { /* skip blanks */ }

        /*  c  is  now  nonblank  */
```

```
        if(  islower(  c  )  )  {
             yylval  =  c  -  'a';
             return  (  LETTER  );
             }
        if(  isdigit(  c  )  )  {
             yylval  =  c  -  '0';
             return(  DIGIT  );
             }
        return(  c  );
        }
```

APPENDIX C

AN ADVANCED EXAMPLE

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 11. The desk calculator example in Appendix B is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, -, *, /, unary -, and = (assignment), and has 26 floating point variables, "a" through "z." Moreover, it also understands intervals, written as

```
( x , y )
```

where x is less than or equal to y. There are 26 interval valued variables "A" through "Z" that can also be used. The usage is similar to that in Appendix B; assignments return no value and print nothing, while expressions print the floating or interval value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure consisting of the left and right endpoint values, stored as doubles. This structure is given a type name, INTERVAL, by using typedef. The Yacc value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). This entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures.

Observe the use of YYERROR to handle two error conditions: division by an interval containing zero, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc ignores the rest of the offending line.

In addition to mixing types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (scalar or interval) of intermediate expressions. A scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

```
2.5 + ( 3.5 - 4. )
```

and

```
2.5 + ( 3.5 , 4. )
```

The 2.5 is used in an interval valued expression in the second example, but this fact is not known until the , is read; by this time, 2.5 is finished, and the parser cannot go back. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion is applied automatically. Despite this evasion, there are still many cases where the conversion can be applied or not, leading to conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts are resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there are many kinds of expression types instead of just two, the number of rules needed increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

The only unusual feature in lexical analysis is the treatment of floating point constants. The C library routine _atof_ is used to do the actual conversion from a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and error recovery.

```
%{

#  include  <stdio.h>
#  include  <ctype.h>

typedef  struct  interval  {
     double  lo,  hi;
     }  INTERVAL;

INTERVAL  vmul(),  vdiv();

double  atof();

double  dreg[ 26 ];
```

```
INTERVAL  vreg[ 26 ];

%}

%start    lines

%union    {
     int  ival;
     double  dval;
     INTERVAL  vval;
     }

%token  <ival>  DREG  VREG
     /*  indices into dreg, vreg arrays  */

%token  <dval>  CONST       /*  floating point constant */

%type  <dval>  dexp         /*  expression  */

%type  <vval>  vexp         /*  interval expression  */

     /*  precedence information about the operators  */

%left     '+'  '-'
%left     '*'  '/'
%left     UMINUS              /*  precedence for unary minus  */

%%

lines     :    /*  empty  */
     |    lines  line
     ;

line :    dexp  '\n'
               {  printf( "%15.8f\n", $1 );  }
     |    vexp  '\n'
               {  printf( "(%15.8f, %15.8f )\n",
                  $1.lo, $1.hi);  }
     |    DREG  '='  dexp  '\n'
               {  dreg[$1]  =  $3;  }
     |    VREG  '='  vexp  '\n'
               {  vreg[$1]  =  $3;  }
     |    error  '\n'
               {  yyerrok;  }
     ;

dexp :    CONST
     |    DREG
               {  $$  =  dreg[$1];  }
     |    dexp  '+'  dexp
               {  $$  =  $1  +  $3;  }
     |    dexp  '-'  dexp
```

```
                  {  $$  =  $1  -  $3;  }
          |    dexp  '*'  dexp
                  {  $$  =  $1  *  $3;  }
          |    dexp  '/'  dexp
                  {  $$  =  $1  /  $3;  }
          |    '-'  dexp %prec  UMINUS
                  {  $$  =  - $2;  }
          |    '('  dexp  ')'
                  {  $$  =  $2;  }
          ;

     vexp :    dexp
                  {  $$.hi  =  $$.lo  =  $1;  }
          |    '('  dexp  ','  dexp  ')'
                  {
                  $$.lo  =  $2;
                  $$.hi  =  $4;
                  if(  $$.lo  >  $$.hi  ){
                    printf( "interval out of order\n" );
                       YYERROR;
                       }
                  }
          |    VREG
                  {  $$  =  vreg[$1];    }
          |    vexp  '+'  vexp
                  {  $$.hi  =  $1.hi  +  $3.hi;
                     $$.lo  =  $1.lo  +  $3.lo;    }
          |    dexp  '+'  vexp
                  {  $$.hi  =  $1  +  $3.hi;
                     $$.lo  =  $1  +  $3.lo;    }
          |    vexp  '-'  vexp
                  {  $$.hi  =  $1.hi  -  $3.lo;
                     $$.lo  =  $1.lo  -  $3.hi;    }
          |    dexp  '-'  vexp
                  {  $$.hi  =  $1  -  $3.lo;
                     $$.lo  =  $1  -  $3.hi;    }
          |    vexp  '*'  vexp
                  {  $$  =  vmul(  $1.lo,  $1.hi,  $3  );  }
          |    dexp  '*'  vexp
                  {  $$  =  vmul(  $1,  $1,  $3  );  }
          |    vexp  '/'  vexp
                  {  if(  dcheck(  $3  )  )  YYERROR;
                     $$  =  vdiv(  $1.lo,  $1.hi,  $3  );  }
          |    dexp  '/'  vexp
                  {  if(  dcheck(  $3  )  )  YYERROR;
                     $$  =  vdiv(  $1,  $1,  $3  );  }
          |    '-'  vexp %prec  UMINUS
                  {  $$.hi  =  -$2.lo;    $$.lo  =  -$2.hi; }
          |    '('  vexp  ')'
                  {  $$  =  $2;  }
          ;
```

```
%%

#  define  BSZ  50
/*  buffer size for floating point numbers  */

     /*  lexical  analysis  */

yylex(){
     register  c;

     while(  (c=getchar())  ==  ' '  )
          {  /*  skip  over  blanks  */  }

     if(  isupper(  c  )  ){
          yylval.ival  =  c  -  'A';
          return(  VREG  );
          }
     if(  islower(  c  )  ){
          yylval.ival  =  c  -  'a';
          return(  DREG  );
          }

     if(  isdigit(  c  )  ||  c=='.'  ){
          /*  gobble  up  digits,  points,  exponents  */

          char  buf[BSZ+1],  *cp  =  buf;
          int  dot  =  0,  exp  =  0;

          for(  ;  (cp-buf)<BSZ  ;  ++cp,c=getchar()  ){

               *cp  =  c;
               if(  isdigit(  c  )  )  continue;
               if(  c  ==  '.'  ){
                    if(  dot++  ||  exp  )  return(  '.'  );
               /*  will  cause  syntax  error  */
                    continue;
                    }

               if(  c  ==  'e'  ){
                    if(  exp++  )  return(  'e'  );
               /*  will  cause  syntax  error  */
                    continue;
                    }

               /*  end  of  number  */
               break;
               }
          *cp  =  '\0';
          if(  (cp-buf)  >=  BSZ  )
            printf(  "constant  too  long:  truncated\n"  );
          else  ungetc(  c,  stdin  );
            /*  push  back  last  char  read  */
```

```
            yylval.dval = atof( buf );
            return( CONST );
            }
        return( c );
        }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
        /* returns the smallest interval */
        /* containing a, b, c, and d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a;  v.lo = b; }
        else { v.hi = b;  v.lo = a; }

        if( c>d ) {
            if( c>v.hi ) v.hi = c;
            if( d<v.lo ) v.lo = d;
            }
        else {
            if( d>v.hi ) v.hi = d;
            if( c<v.lo ) v.lo = c;
            }
        return( v );
        }

INTERVAL vmul( a, b, v ) double a, b; INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }

dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
            printf( "divisor interval contains 0.\n" );
            return( 1 );
            }
        return( 0 );
        }

INTERVAL vdiv( a, b, v ) double a, b; INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

APPENDIX D

OLD FEATURES

This Appendix mentions synonyms and features that are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals can also be delimited by double quotes (").

2. Literals can be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multicharacter literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job that must be actually done by the lexical analyzer.

3. Most places where % is legal, backslash (\) can be used (\\ is the same as %%, \left the same as %left).

4. There are a number of other synonyms:

   ```
   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec
   ```

5. Actions can also have the form

   ```
   ={ . . . }
   ```

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

ZEUS FOR BEGINNERS*

* This information is based on an article originally authored by Brian W. Kernighan, Bell Laboratories.

PREFACE

This manual introduces the ZEUS operating system. It includes the basic procedures and commands needed for day-to-day use of the system. The major formatting programs and macro packages used for document preparation and hints on preparing documents are discussed. Descriptions of supporting software and ZEUS programming are also included.

This manual is divided into four sections. Section 1 describes how to log in, how to enter data, what to do about typing errors, and how to log out. Some of this information is dependent on the system and terminal that are being used, so this section must be supplemented by local information. Information required for day-to-day use of the system (such as commonly used commands) is found in Section 2. Section 3 describes some of the formatting tools used in preparing manuscripts. Some of the tools used for developing programs are described in Section 4.

For further information, refer to the ZEUS Reference Manual and the ZEUS Utilities Manual.

TABLE OF CONTENTS

SECTION 1 GETTING STARTED

1.1 Logging In 4
1.2 Typing Commands 4
1.3 Unusual Terminal Behavior 5
1.4 Typing Errors 5
1.5 Read-Ahead 6
1.6 Stopping a Program 6
1.7 Logging Out 6
1.8 Mail 6
1.9 Writing to Other Users 7
1.10 On-Line Manual 8
1.11 Computer-Aided Instruction 8

SECTION 2 DAY-TO-DAY USE

2.1 The Editor 9
2.2 The List Command 10
2.3 Displaying Files 11
2.4 Rearranging Files 12
2.5 File Names 13

2.5.1 Directories and Path Names 15
2.5.2 Current Directory 17
2.5.3 Subdirectories 17

2.6 Using Files Instead of Terminal Input and Output ... 18
2.7 Pipes 19
2.8 The Shell 20

SECTION 3 DOCUMENT PREPARATION

3.1 Introduction 23
3.2 Formatting Programs 23
3.3 Supporting Tools 24
3.4 Hints for Preparing Documents 25

SECTION 4 PROGRAMMING

4.1 Introduction 26
4.2 Programming the Shell 27
4.3 Programming in C 27

SECTION 1

GETTING STARTED

1.1 Logging In

Terminals are connected to the system by a high-speed asynchronous line. Log in when the message login: appears on the terminal. If this message is not on the screen, press the RETURN key. If the message still does not appear, contact the System Administrator for assistance.

When login: is displayed, enter the login name in lowercase, followed by a RETURN. For terminals that have only uppercase, it is possible to type commands in uppercase. If the login name is typed in uppercase, the entire terminal session must be performed in uppercase. The system does not respond until a RETURN is entered. If a password is required, the message Password: appears. Enter the password, followed by a RETURN. The password, which protects files from unauthorized access, is not echoed on the screen.

When a prompt character appears on the screen, the system is ready to accept commands. The prompt character is usually a dollar sign ($) or a percent sign (%). (Messages of the day or notifications that mail is being held can appear on the screen before the prompt character.)

1.2 Typing Commands

Once the prompt appears, commands (requests that the system do something) can be entered. Type the command

```
date
```

followed by a RETURN. A response similar to

```
Mon Jan 16 14:17:10 EST 1978
```

is displayed.

Always press RETURN after every command line; the system does not respond unless RETURN is pressed.

The command who specifies everyone who is currently logged in to the system. Entering

```
who
```

causes a response similar to the following:

```
ski  tty05    Jan 16    09:33
gam  tty11    Jan 16    13:07
```

The time specifies when the user logged in; ttyxx indicates the terminal being used.

If a typing mistake is made when a command is entered, thereby referencing a nonexistent command, the system responds with an error message. For example, typing

```
whom
```

results in the response

```
whom: not found
```

If the name of some other command is inadvertently typed, that command is run.

If the terminal does not have tabs, type the command

```
stty -tabs
```

The system then converts each tab into the correct number of spaces when printing. If the terminal does have computer-settable tabs, the command _tabs_ sets the stops. Refer to _stty_(_1_) in the _ZEUS Reference Manual_. (The notation _stty_(_1_) refers to the command _stty_ in Section 1 of the _ZEUS Reference Manual_.)

1.3 Unusual Terminal Behavior

Sometimes the terminal functions incorrectly. For example, each letter may be typed twice, or RETURN may not cause a line feed or a return to the left margin. Logging out and logging back in may correct this.

1.4 Typing Errors

A typing error that is discovered before RETURN is typed can be corrected in one of two ways. Control-h (hitting "h" while holding down the control key) erases the last character typed. Control-h can be repeated to erase characters back to the beginning of the line (but not beyond).

Control-x erases the current input line. If a line of text has several errors, type control-x and then retype the line. The system always echoes a new line after the control-x

character.

The stty(1) command can be used to change the erase and kill characters. Backspace can also be used as an erase character, and control-x can be used as a kill character.

1.5 Read-Ahead

Read-ahead capability allows typing to be done as fast as possible, even while the system is responding to a command. If typing is done while the system is outputting text, the input characters appear intermixed with the output characters; however, they are interpreted in the correct order. Several commands can be typed one after another without waiting for each one to execute.

1.6 Stopping a Program

Most programs can be stopped by typing the character RUB (usually the delete or rubout key on the terminal). On most terminals, the "interrupt" or "break" key can also be used. In a few programs, such as the text editor, RUB stops whatever the program is doing but does not stop the program itself. Hanging up the phone also stops most programs, but this is not a recommended method of exiting a program.

1.7 Logging Out

To log out, type a control-d or type

    logout

It is not sufficient to turn off the terminal because ZEUS does not use a time-out mechanism. When using a phone, it is possible to log out by hanging up, but this is not recommended.

1.8 Mail

After logging in, the message

    you have mail.

may appear. ZEUS provides a postal system, allowing for communication with other users on the system. To read the mail, type the command

    mail

Mail appears, one message at a time, with the most recent message given first. After each message, _mail_ waits for a user response. Typing a _d_ deletes the message. Typing RETURN causes mail to continue, leaving the message on the system; it will appear again the next time mail is read. Other responses are described in _mail_(_1_) of the _ZEUS Reference Manual_.

To send mail to "joe" (a user whose login name is joe), type

    mail joe

Then enter the text of the letter, using as many lines as necessary. After the last line of text, type control-d.

There are other ways to send mail. Mail can be sent to oneself as a handy reminder mechanism. Previously prepared mail can be sent to a number of people simultaneously. For more details, see _mail_(_1_).

1.9 Writing to Other Users

A message like

    message from joe tty07...

may appear on the terminal, accompanied by a beep. This indicates that Joe is on line and wants to send a message. To respond, type the command

    write joe

This establishes a two-way communication path, and messages can be exchanged via the terminals. This path is slow compared to system response in general. It is necessary to terminate any program that is being run before messages can be received. (It is possible to temporarily escape from the editor. Refer to the editor tutorial in the _ZEUS Utilities Manual_.)

To keep the messages from becoming intermixed, care should be taken to ensure that both users do not type messages at the same time. A common way of doing this is to type an _o_ on a line by itself at the end of the message to indicate that the message is over. To terminate a conversation, each side must type a control-d or a delete character on a line by itself.

If an attempt is made to write to someone who is not logged in, the system responds with the message

person not logged in

If an attempt is made to write to someone who does not want to be disturbed, the system responds with the message

permission denied

If the target person is logged in but does not answer, type control-d to obtain a prompt.

1.10 On-Line Manual

The _ZEUS Reference Manual_ is usually kept on line, and sections of it can be displayed at the terminal. The _ZEUS Reference Manual_ also contains the most up-to-date information on commands. To print a manual section, type

man command-name.

For example, to read about the _who_ command, type

man who

1.11 Computer-Aided Instruction

The ZEUS system has a program called _learn_ that provides computer-aided instruction on the file system and basic commands, the editor, document preparation, and programming in C. Enter the command

learn

for further information.

SECTION 2

DAY-TO-DAY USE

2.1 The Editor

The ZEUS text editor, *ed*, is usually used to type papers, letters, and programs, and to store information in the computer. Refer to *ed*(*1*) and *ED* in the *ZEUS Reference Manual* for in-depth explanations on how to use the editor.

To create a file called *junk* containing some text, enter

```
ed junk       (invokes the text editor; the system
               responds by listing the number of
               characters in the file)
a             (command to ed, to add text)
text
.             (signals the end of adding text)
```

A period (.) typed by itself at the beginning of a line indicates the end of text addition. Until it is entered, everything typed is treated as text to be added, and no other *ed* commands are recognized.

To store the information that has been typed into a file, use the editor command *w*. The editor responds by listing the number of characters in the file *junk*. Until the *w* command is entered, nothing is stored permanently. Therefore, if the user hangs up or logs out, the information is lost. (There is, however, a special feature of ZEUS that saves the edited data in a file called *ed*.*hup*.) After a *w* command is issued, the stored information can be accessed at any time by typing

```
ed junk
```

To exit from the editor, type a quit (*q*) command. If the *q* command is entered before the text has been stored, *ed* prints a ? as a reminder. Entering a second *q* followed by an exclamation point (!) causes the exit to take place.

Now create a second file called *temp* in the same manner. Two files, *junk* and *temp*, should now exist.

2.2 The List Commands

The list (*ls*) command lists the names (not contents) of all files in the directory. If

    ls

is typed, the response is

```
junk
temp
```

These are the two files just created. Unless an optional argument is added to the *ls* command, the names are listed alphabetically. Other variations are possible. For example, the command

```
ls -t
```

lists the files in the order in which they were last changed, with the most recently changed file listed first. Typing

```
ls -l
```

produces a long listing similar to the following:

```
-rwxrwxrwx  1 bwk   41 Jul 22 2:56 junk
-rwxrwxrwx  1 bwk   78 Jul 22 2:57 temp
```

The date and time indicate when the last changes to the file were made. The 41 and 78 refer to the number of characters in the file. The initials *bwk* indicate the owner of the file, that is, the person who created it. The -*rwxrwxrwx* specifies who has permission to read, write, and execute the file. The first dash in each line indicates an ordinary file; a d instead of a dash indicates a directory. The left-most *rwx* indicates the read, write, and execute permissions for the owner of the file. The middle *rwx* pertains to the read, write, and execute permissions for the user group to which the owner belongs. The right-most *rwx* pertains to everyone else. In this example, everyone has read, write, and execute permission. For more information, refer to *chmod*(*1*) and *chmod*(*2*).

Listing options can be combined. For example, the command *ls* -*lt* gives a long listing (-*l*) in time order (-*t*). More information is found in *ls*(*1*).

The use of optional arguments that begin with a dash (like -*t* and -*lt*) is a common convention for ZEUS programs. In

general, if a program accepts such optional arguments, they precede any file name arguments. The various arguments must be separated with a blank space (*ls-l* is not the same as *ls -l*).

2.3 Displaying Files

Use the editor to display a file of text on the screen. Type

```
ed junk
1,$p
```

and *ed* lists the number of characters in *junk* and then displays the entire file on the screen.

It is not always feasible to use the editor for displaying files. There is a limit to the size of files that *ed* can handle, and only one file can be displayed at a time. There are alternate programs suitable to specific applications.

The *cat* command displays the contents of all the files named in a list. For example,

```
cat junk
```

displays the file *junk*, and

```
cat junk temp
```

displays the files *junk* and *temp*. The files are simply concatenated (hence the name *cat*) onto the screen.

The *pr* command produces formatted displays of files. As with *cat*, *pr* displays all the files named in a list, but *pr* displays text in formatted form, including headings with date, time, page number, and file name at the top of each page. The command

```
pr junk temp
```

displays *junk*, then skips to the top of a new page and displays *temp*.

The *pr* command can also produce multicolumn output. For example,

```
pr -3 junk
```

prints the file *junk* in three-column format. Any number of columns can be printed. See *pr*(*1*) for more information.

The command *dog* displays the contents of a specified file one page at a time. For example,

```
dog junk
```

displays the first page of the file *junk* on the terminal. Pressing the RETURN key causes the text to scroll forward, displaying the next page.

There are also programs that print ZEUS files on a high-speed printer. See *lpr* in the *ZEUS Reference Manual*. The *nroff* and *troff* programs are more complete text formatters. They are discussed in Section 3 and in the *ZEUS Utilities Manual*.

2.4 Rearranging Files

A file can be moved from one place to another (which amounts to changing the name) using the *mv* command. For example, typing

```
mv junk stuff
```

moves the contents of the file *junk* into the file *stuff*. If the *ls* command is entered, the response is now

```
stuff
temp
```

NOTE

> If a file is moved to another file that already exists, the already existing contents are lost forever.

To make a copy of a file, use the *cp* command.

```
cp stuff templ
```

makes a duplicate copy of *stuff* in *templ*.

The *rm* command removes (deletes) files from a directory. For example,

```
rm temp templ
```

deletes the files *temp* and *templ*.

NOTE

> Be very careful when using the rm command. Once files are removed with the rm command, they no longer exist in the directory and can never be recovered.

A warning is displayed if one of the named files does not exist. Otherwise _rm_, like most ZEUS commands, does its work silently.

2.5 File Names

File names can be no longer than 14 characters. Although almost any character can be used in a file name, it is recommended that only letters, numbers, and the period be used. This is to avoid characters that might have other meanings. For example, if a file were created with the name -_t_, listing it by name would be difficult, if not impossible, because -_t_ is an optional argument for requesting a time-order listing.

If a large manual is being typed, it must be divided into several smaller sections because the size of files that _ed_ can handle is limited. The document should therefore be typed as a number of smaller files. Each chapter can be in a separate file named chap1, chap2, etc., or each chapter can be broken into several files named chap1.1, chap1.2, chap1.3, chap2.1, chap2.2, etc. This naming system makes the relationship between the files obvious.

One advantage to a systematic naming convention is that the entire book can be displayed with one command, such as

```
pr chap*
```

The asterisk (*) is a pattern matching character that means "anything at all," so this command prints in alphabetical order all files whose names begin with chap. This shorthand notation is used system-wide, not just with _pr_. For example, to list all the names of the files in the manual, enter

```
ls chap*
```

This lists

```
chap1.1
chap1.2
chap1.3
...
```

The * is not limited to the last position in a file name--it can be anywhere and can occur several times. For example,

```
rm *junk* *temp*
```

removes all files that contain _junk_ or _temp_ as any part of their name. As a special case, * by itself matches every filename, so

```
pr *
```

prints all the user's files in alphabetical order; and

```
rm *
```

removes _all files_ in the current directory.

The * is not the only pattern-matching feature available. It is possible to match a group of characters by enclosing them in brackets ([ ]). For example, if only Chapters 1 through 4 and Chapter 9 are to be printed, type

```
pr chap[12349]*
```

A range of consecutive letters or digits can be abbreviated.

```
pr chap[1-49]*
```

A range of letters can also be specified with brackets. For example, [_a_-_z_] matches any character in the range _a_ through _z_.

The question mark (?) pattern matches any single character. For example,

```
ls ?
```

lists all files that have single-character names, and

```
ls -l chap?.l
```

lists the first file of each chapter (_chapl_._l_, _chap2_._l_).

To cancel the special meaning of * or ?, enclose the argument in single quotes.

```
ls '?'
```

2.5.1 Directories and Path Names

Generally, each user has a private directory containing only the files that belong to that user. When logged in, the user is in his/her private directory, and unless special action is taken when a new file is created, it is created in the directory the user is currently in. This is most often the user's own directory, and therefore, the file is unrelated to any other file of the same name that exists in someone else's directory.

All files are organized in sets located in a tree, with the individual user's files located several branches outward from the root. Any file in the system can be found by starting at the root of the tree and moving along the proper set of branches. It is also possible to move inward toward the root.

The command _pwd_ (print working directory) prints the path name of the directory the user is currently in.

The response to the _pwd_ command is something similar to

    /z/your-name

This indicates that the user is currently in the directory _your-name_, which is in the directory /_z_, which is, in turn, in the root directory, called /.

Typing

    ls /z/your-name

lists the same file names obtained from the _ls_ command alone. With no arguments, _ls_ lists the contents of the current directory; given the name of a directory, it lists the contents of that directory.

Typing

    ls /z

prints a series of names, among which is _your-name_. In many installations, _z_ is a directory that contains the directories of all users of the system.

Typing

    ls /

gives a response something like:

```
bin
dev
etc
lib
tmp
usr
```

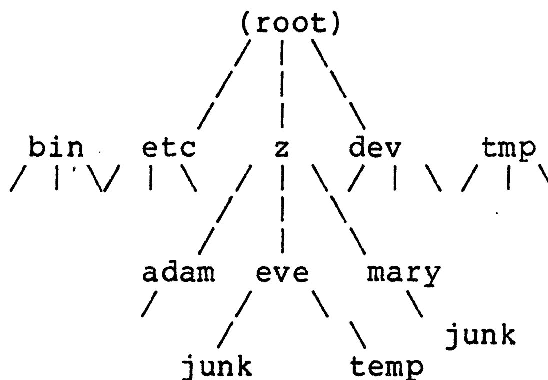
These are the basic directories of files--the root of the tree.

The full name of the path to be followed from the root through the tree of directories to get to a particular file is the path name. The path name of the file _junk_ is

```
/z/your-name/junk
```

It is a universal rule in the ZEUS system that anywhere an ordinary file name can be used, a path name can be used.

Here is a picture of the tree used in this document:



Observe that mary's _junk_ file is unrelated to eve's _junk_ file.

To obtain a listing of files in another user's directory, type

```
ls /z/neighbor-name
```

To copy of one of these files, type

```
cp /z/your-neighbor/his-file yourfile
```

If users do not want other people examining these files, privacy can be arranged. Each file and directory has read-write-execute permissions for the owner, a group, and everyone else, which control file access. (See _ls_(_1_) and _chmod_(_1_) for details.) For an executable file, the owner generally has read, write, and execute permission; other

people in the owner's group might have read or execute permission; everyone else might have only execute permission.

As a final experiment with path names, try

```
ls /bin /z/bin
```

2.5.2 Current Directory

When the name of a file (command) is entered in response to the prompt character, the system looks for an executable file of that name in the current directory. If the file is not found in the current directory, the system searches /*bin*, and finally /*usr*/*bin*. The search path, which is normally the current directory, /bin, and /usr/bin can be changed. (See *sh*(*1*) and *csh*(*1*) in the *ZEUS Reference Manual*.)

If a user works regularly with someone else on common information in the other's directory, the user could simply log in under the other's login name each time the information is needed. It is also possible to change directories. Type

```
cd /z/your-friend
```

and a file name used with a command like *cat* or *pr* refers to the file in *your*-*friend* directory. Changing directories does not affect any permissions associated with a file. That is, if a file could not be accessed from the user's own directory, changing to another directory does not alter that fact.

Type

```
pwd
```

to find out which directory is the current directory.

2.5.3 Subdirectories

It is convenient to arrange files so that all files on a related subject are in a directory that is separate from other projects. For example, when writing a manual, it might be helpful to keep the text in a directory called *book*. To make the directory, use the command

```
mkdir book
```

This creates the directory called *book*. To go to that directory, type

```
cd book
```

Separate files can now be established in this directory. The path name of this directory is:

```
/z/your-name/book
```

To move back up to the login directory (one level up in the tree), type

```
cd ..
```

The double period (..) indicates the parent of the currently accessed directory. A single period (.) is an alternate name for the working directory.

To remove the directory _book_, type

```
rm book/*
rmdir book
```

The first command removes all files from the directory, and the second removes the empty directory.

2.6 Using Files Instead of Terminal Input and Output

Most of the commands discussed so far produce output on the terminal. Some, like the editor, also take their input from the terminal. In ZEUS systems, input, output, or both can go to or from files rather than the terminal. For example,

```
ls
```

lists all files on the terminal screen. However, entering

```
ls >filelist
```

places a list of files in the file _filelist_, which is created if it does not exist or is overwritten if it does. The symbol > means that the output should go to the following file rather than the terminal screen. Several files can be combined into one by capturing the output of _cat_ in a file. For example,

```
cat f1 f2 f3 >temp
```

This concatenates _f1_, _f2_, and _f3_ into the file _temp_.

The symbol >> operates very much like > does. It means add the listed files to the end of the file that follows the symbol. That is,

```
cat f1 f2 f3 >>temp
```

means to add *f1*, *f2*, and *f3* to the end of whatever is already in *temp* (instead of overwriting the existing contents of *temp*). As with >, if *temp* does not exist, it is created.

The symbol < means take the input for a program from the following file instead of from the terminal. For example, it is possible to create a file called *script* containing a group of editing commands that produces a specified set of changes. Typing

```
ed file <script
```

causes the set of editing commands to be executed throughout the file.

As another example, *ed* can be used to prepare a letter in the file *let*. Then, the letter can be sent to several people with

```
mail adam eve mary joe <let
```

2.7 Pipes

A *pipe* is a means of connecting the output of one program to the input of another program so that the two run as a sequence of processes. A command line that uses pipes is called a pipeline.

For example,

```
pr f g h
```

displays the files *f*, *g*, and *h*, beginning each on a new page. It is possible to display them together without page breaks by entering

```
cat f g h >temp
pr <temp
rm temp
```

A simpler way to do this is to take the output of *cat* and connect it to the input of *pr* by using a pipe.

```
cat f g h | pr
```

The vertical bar (|), which is the pipe command, means take the output from *cat*, which would normally have gone to the terminal, and put it into *pr* to be formatted.

The pipeline

```
ls | pr -3
```

displays a list of files in three columns.

Any program that reads from the terminal can also read from a pipe. Any program that writes to the terminal can also drive a pipe. Any number of elements can be used in a pipeline.

Many ZEUS programs are written so that they can take their input from one or more files if file arguments are given. If no arguments are given, the programs read from the terminal and can be used in pipelines. One example is _pr_.

```
pr -3 a b c
```

prints files _a_, _b_, and _c_ in order, in three-column format. The command

```
cat a b c | pr -3
```

produces the same output; _pr_ prints the information coming down the pipeline in three-column format.

2.8 The Shell

The shell is the program that interprets the commands and arguments entered at the terminal. (See _sh_(_1_) and _csh_(_1_).) It also interprets characters that have special meaning in ZEUS. For example, two programs can be run with one command line by separating the commands with a semicolon (;). The shell recognizes the semicolon and breaks the line into two commands. In the command line

```
date; who
```

the shell executes the _date_ and _who_ commands before returning with a prompt character.

More than one program can be run simultaneously. For example, if something time consuming, like the editor script, is being run, type

```
ed file <script &
```

The ampersand at the end of a command line means start the command running in the background and then take further commands from the terminal immediately. To prevent the output from interfering with what is being done on the terminal,

type

```
ed file <script >script.out &
```

which saves the output lines in a file called *script.out*.

When initiating a command with &, the system replies with a number called the process number, which identifies the command so that it can be stopped later. To stop the command from executing, type

```
kill process-number
```

If the process number is forgotten, the command *ps* lists the process numbers of everything that *ls* is running. (It is possible to use the command *kill 0*, which kills all the user processes that are running. This command should, of course, be used with caution.) The command *ps -a* lists all programs in the system that are currently running.

The command

```
(command-1; command-2; command-3) &
```

can be used to start three commands in the background. A background pipeline can be started with

```
command-1 | command-2 &
```

Just as the editor or some similar program can take its input from a file instead of from the terminal, the shell can read a file to get commands. For instance, suppose the tabs on the terminal are to be set, and the date and who is on the system are to be displayed every time the user logs in. The three necessary commands (*tabs*, *date*, *who*) can be put into a file called *startup*. To run this program, type

```
sh startup
```

The shell then runs with the file *startup* as input. This has the same effect as entering the contents of *startup* on the terminal.

To eliminate the need to type *sh* each time, use the command

```
chmod +x startup
```

The *chmod* command marks the file as executable; the shell recognizes this and runs it as a sequence of commands. Thereafter, type only

```
startup
```

to run the sequence of commands.

If *startup* is to be run automatically after every login, place its contents in the current home directory in a file called .*profile* (if running in shell), or .*cshrc* (f the shell running is the C shell). When the shell gains control after the login, it looks for and executes the .*profile* or .*cshrc* file.

SECTION 3

DOCUMENT PREPARATION

3.1 Introduction

The ZEUS system has two major formatting programs for document preparation: *nroff*, which produces output on terminals and line printers, and *troff*, which drives a phototypesetter.

3.2 Formatting Programs

Formatting programs use commands that are entered along with the text that is to be formatted. The commands indicate in detail how the formatted text is to look. For example, there are commands that specify how long lines should be, whether to use single or double spacing, and what running titles are to be used on each page.

For *nroff* and *troff*, several packages of canned formatting requests called macro packages are available. These allow specification of formatting elements such as paragraphs, running titles, footnotes, and multicolumn output. It is not necessary to learn *nroff* and *troff* to use these macro packages. Formatting requests typically consist of a period and two uppercase letters; for example, .*TL* is used to introduce a title, and .*PP* is used to begin a new paragraph.

A document is typed so that it looks something like this:

```
.TL
title of document
.AU
author name
.SH
section heading
.PP
paragraph ...
.PP
another paragraph ...
.SH
another section heading
.PP
```

The precise meaning of .PP depends on whether the output device being used is a typesetter or terminal. For example, a paragraph is normally preceded by a space (one line in *nroff*, one half line in *troff*), and the first word is indented. These rules can be changed as required.

To print a document in standard format using -*ms*, use the command

```
troff -ms files
```

for the typesetter and

```
nroff -ms files
```

for a terminal. The -*ms* argument tells *troff* and *nroff* to use the manuscript package of formatting requests. (Refer to *ms*(7) for more information.)

There are several similar packages; see the information on text formatting in the *ZEUS Utilities Manual*.

3.3 Supporting Tools

In addition to the basic formatters, there are other supporting programs for document preparation.

Any spelling errors in a document can be detected by the programs *spell* and *typo*. The *spell* program compares the words in the document to a dictionary, then prints those that are not in the dictionary. The *typo* program searches for words that are unusual, then prints them.

The *grep* program examines a set of files for lines that contain a particular text pattern. For example,

```
grep 'ing$' chap*
```

finds all lines that end with the letters *ing* in the files *chap**. (It is always good practice to put single quotes around the pattern being searched for, in case it contains characters like * or $ that have a special meaning to the shell.) The *grep* program is useful for discovering which set of files contains the misspelled words detected by *spell*.

A list of the differences between two files is printed by *diff*. Two versions of something can be compared automatically, eliminating the necessity of proofreading.

The words, lines, and characters in a set of files are counted by *wc*.

The tr program translates characters into other characters. For example, it converts uppercase to lowercase and vice versa. The following command translates uppercase into lowercase:

```
tr A-Z a-z <input >output
```

Files can be sorted in a variety of ways by sort.

The ptx program makes a permuted index (keyword-in-context listing).

The sed program provides many of the editing facilities of ed, but can apply them to arbitrarily long inputs.

For more information on these programs, see the ZEUS Reference Manual.

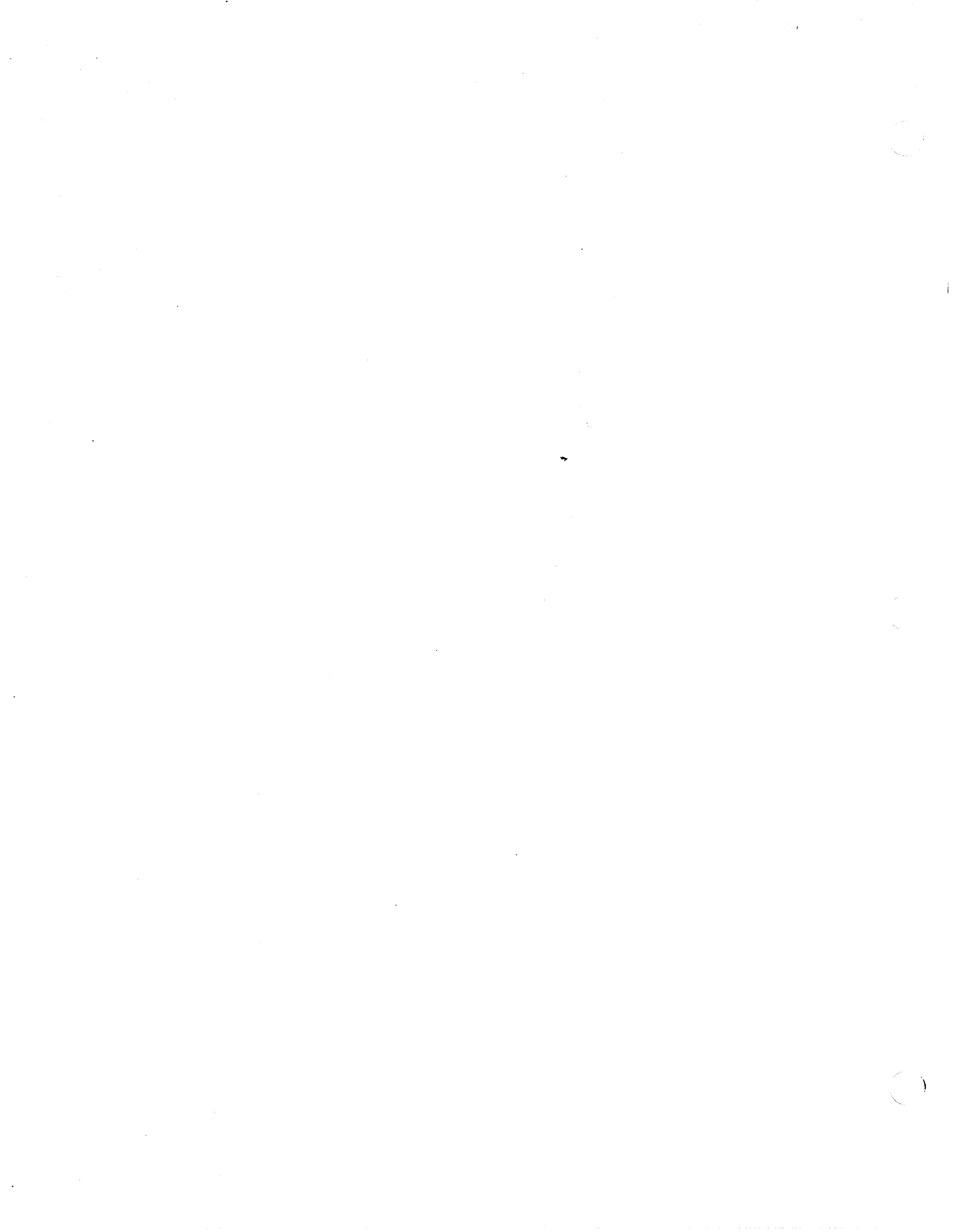
3.4 Hints for Preparing Documents

Most documents go through several drafts before they are finished. The following hints make the process of revising drafts easier.

When the text is being typed, start each sentence on a new line, make lines short, and break lines at natural places, such as after commas and semicolons. Since most people change documents by rewriting phrases and adding, deleting, and rearranging sentences, these precautions will simplify any editing done to the document.

Keep the individual files of a document short (perhaps ten to fifteen thousand characters). Larger files edit more slowly, and of course, if an error is made, it is better to have destroyed a small file rather than a big one. Split documents into files at natural boundaries.

Refrain from deciding formatting details too early. One of the advantages of the formatting packages is that they permit decisions to be delayed until the last possible moment. As long as the text has been entered in some systematic way, it can always be cleaned up and reformatted by a judicious combination of editing commands and request definitions.



SECTION 4

PROGRAMMING

4.1 Introduction

The ZEUS system is a productive programming environment because it offers a rich set of programming tools. Facilities such as pipes, I/O redirection, and the capabilities of the shell make it possible to do a job by pasting together programs that already exist instead of writing from scratch.

The pipe mechanism allows fabrication of complicated operations out of spare parts that already exist. For example, an early version of the *spell* program was

```
cat ... | tr ... | tr ... | sort | uniq | comm
```

where *cat* collected the files, the first *tr* put each word on a new line, and the second *tr* deleted punctuation. The information was then sorted into dictionary order. The *uniq* command discarded duplicates, and *comm* printed words that were in the text but not found in the dictionary.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines for each file in a set of files, the following can be laboriously typed

```
ed
e chapl.1
lp
$p
e chapl.2
lp
$p
etc.
```

An easier way is to type

```
ls chap* >temp
```

This lists the file names in the *temp* file. Then this file can be edited to incorporate the necessary series of editing commands (using the global commands of *ed*). When these commands have been written into *script*, the command

```
ed <script
```

produces the same output as the laboriously typed list of

commands. Alternately, since the shell performs loops, it is possible to repeat a set of commands over and over again for a set of arguments. For example,

```
for i in chap*
do
     ed $i <script
done
```

sets the shell variable _i_ to each file name in turn, then does the command. This command can be typed at the terminal or put in a file for later execution.

4.2 Programming the Shell

The shell itself is a programming language with variables, control flow (_if_-_else_, _while_, _for_, _case_), subroutines, and interrupt handling. Since there are many building-block programs, a new program can sometimes be created by piecing together some of the building blocks with shell command files.

Examples and rules for running the shell and the C shell can be found in _SHELL_, and _CSHELL_ in the _ZEUS Utilities Manual_.

4.3 Programming in C

ZEUS and most of the programs that run on it are written in C. C is an easy language to learn and use. It is introduced and fully described in _The C Programming Language_ by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). See the _ZEUS Reference Manual_ for additional information.

Systems Publications

Reader's Comments

Your feedback about this document helps us ascertain your needs and fulfill them in the future. Please take the time to fill out this questionaire and return it to us. this information will be helpful to us and, in time, to future users of Zilog products.

Your Name: ______________________________

Company Name: ______________________________

Address: ______________________________

Title of this document: ______________________________

Briefly describe application: ______________________________

____

____

Does this publication meet your needs? ☐ Yes ☐ No If not, why not? ______________________________

____

____

How are you using this publication?

- ☐ As an introduction to the subject?
- ☐ As a reference manual?
- ☐ As an instructor or student?

How do you find the material?

	Excellent	Good	Poor
Technicality	☐	☐	☐
Organization	☐	☐	☐
Completeness	☐	☐	☐

What would have improved the material? ______________________________

____

____

Other comments and suggestions: ______________________________

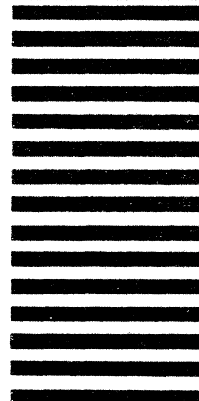
____

____

If you found any **mistakes** in this document, please let us know what and where they are: ______________________________

____

____



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 35 CAMPBELL, CA.

POSTAGE WILL BE PAID BY ADDRESSEE

Zilog

Systems Publications
1315 Dell Avenue
Campbell, California 95008
Attn: Publications Manager

Reader's Comments

Your feedback about this document helps us ascertain your needs and fulfill them in the future. Please take the time to fill out this questionaire and return it to us. this information will be helpful to us and, in time, to future users of Zilog products.

Your Name: ______________________________

Company Name: ______________________________

Address: ______________________________

Title of this document: ______________________________

Briefly describe application: ______________________________

____

____

Does this publication meet your needs? ☐ Yes ☐ No If not, why not? ______________________________

____

____

How are you using this publication?

- ☐ As an introduction to the subject?
- ☐ As a reference manual?
- ☐ As an instructor or student?

How do you find the material?

	Excellent	Good	Poor
Technicality	☐	☐	☐
Organization	☐	☐	☐
Completeness	☐	☐	☐

What would have improved the material? ______________________________

____

____

Other comments and suggestions: ______________________________

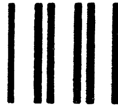
____

____

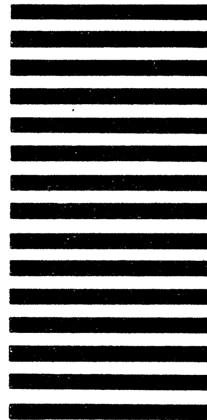
If you found any mistakes in this document, please let us know what and where they are: ______________________________

____

____



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 35 CAMPBELL, CA.

POSTAGE WILL BE PAID BY ADDRESSEE

Zilog

Systems Publications
1315 Dell Avenue
Campbell, California 95008
Attn: Publications Manager

Zilog, Inc. 1315 Dell Ave., Campbell, California 95008 Telephone (408)370-8000 TWX 910-338-7621

00-2147-01 Printed in USA