System 8000™
ZEUS Languages/Programming Tools Manual

03-3249-01

May, 1983

ZEUS Languages/Programming Tools Manual

ZEUS Release 3.2

10/14/83

Preface

This document and the related manuals listed below provide the complete software technical documentation for the standard ZEUS Operating System.

System 8000 ZEUS Utilities Manual	03-3150
System 8000 ZEUS Reference Manual	03-3255
System 8000 ZEUS Administrator Manual (Models 21/31)	03-3246
System 8000 Model 11 ZEUS Administrator Manual	03-3254

Of particular interest to the System 8000 programmer are Sections 2 (system calls) and 3 (C library functions) of the System 8000 ZEUS Reference Manual.

In addition, the following manuals, also supplied with the system, complement the System 8000 ZEUS Languages/Programming Tools Manual:

The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie

The Z8000 PLZ/ASM Assembly Language Programming Manual (03-3055)

TABLE OF CONTENTS

Introduction to ZEUS Languages/Programming Tools	INTRO
A Tutorial Introduction to ADB	ADB
System 8000 Assembly Language Reference Manual	AS
The C Programming Language	C
System 8000 Calling Conventions	CALL CONV
C-ISAM Programmer's Guide	C-ISAM
Screen Updating and Cursor Movement Optimization: A Screen Package	CURSES
Lex: A Lexical Analyzer Generator	LEX
Lint: A C Program Checker	LINT
Make	MAKE
The M4 Macro Processor	MP
Zeus Programming	PGMG
ZEUS PLZ/ASM Assembler User Guide	PLZ/ASM
System 8000 PLZ/SYS User Guide	PLZ/SYS
Screen Handling	SCREEN
YACC: Yet Another Compiler-Compiler	YACC

Introduction to ZEUS Languages/Programming Tools Manual

Languages

ZEUS supports many languages -- FORTRAN, Pascal, BASIC, among them. C is the primary programming language, however: recent changes to C and special considerations of programming in C on the System 8000 are listed in The C Programming Language (C). ZEUS Programming (PGMG) explains how C programs interact with ZEUS and handle command arguments, input/output, etc. Lint: A C Program Checker (LINT) detects implementation dependent code and other bad features.

PLZ/SYS is another ZEUS language. Along with the PLZ/ASM assembler, it can be used to design low-level programs. So too can the System 8000 assembler, known simply as the assembler. With the 3.1 release, this assembler becomes the System 8000 core assembler operating as the backend processor for the ZEUS high-level language compilers and translating programs written in the language described in System 8000 Assembly Language Reference Manual (AS).

All languages supported by ZEUS can communicate with each other and share common libraries provided they observe certain calling conventions described in System 8000 Calling Conventions (CALL CONV).

Tools

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

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.

Lex: A Lexical Analyzer Generator and YACC: Yet Another Compiler Compiler decribe tools useful in developing programs which apply translation rules to input.

The M4 Macro Processor (MP) is a user's manual for a front-end program suitable for use with high-level languages such as C and Fortran.

CURSES: Screen Updating and Cursor Movement Optimization: A Library Package and SCREEN: Screen Interface Library: describe a set of tools for developing C programs that manipulate video displays. C-ISAM Programmer's Guide describes the available tools for the creation and maintenance of indexed file systems.

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 De Bugger), 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 1-1

1.1. Basic Command Format 1-1
1.2. File Locations 1-1
1.3. Current Address 1-1
1.4. Formats 1-2
1.5. General Requests 1-3

SECTION 2 DEBUGGING C PROGRAMS 2-1

2.1. Debugging a Core Image 2-1
2.2. Calling Multiple Functions 2-2
2.3. Setting Basic Breakpoints 2-3
2.4. Setting Advanced Breakpoints 2-5
2.5. Using Other Breakpoint Facilities 2-8

SECTION 3 MAPS 3-1

SECTION 4 ADVANCED USAGE 4-1

4.1. General 4-1
4.2. Formatted Dump 4-1
4.3. Directory Dump 4-3
4.4. Ilist Dump 4-3
4.5. Value Conversion 4-3

SECTION 5 PATCHING 5-1

SECTION 6 CAUTIONS 6-1

APPENDIX A PROGRAM EXAMPLES A-1

APPENDIX B ADB SUMMARY B-1

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:

```
Ø126?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	Z8ØØØ 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 8Ø 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
```

is displayed each time. The single character on the left edge is the output from the C program.

Continue 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 entering

```
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  Ø
```

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

The **;Ø** 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 E7Ø7 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 E7Ø7 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 E7Ø7 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 (Ø2Ø bytes for an a.out file and Ø2ØØØ bytes for a core file).

The f2 field is the displacement from the beginning of the file to the data. For an E7Ø7 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 (E7Ø7 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

```
Ø2ØØØ>b
```

which sets **b** to octal 2ØØØ. 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 Ø to Ø37 is printed as @ followed by the corresponding character in the range Ø14Ø to Ø177. 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"
Ø,-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 -
Ø2ØØØ>b
?m <b
<b,-1?"flags"8ton"links,uid,gid"8t3dn",
  size"8tDn"addr"8t2Øun"times"8t2YnY2na
```

In this example, the value of the base for the map was changed to Ø2ØØØ (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:

```
Ø72 = odx
```

prints

```
Ø72      58      %3a
```

which are the octal, decimal, and hexadecimal representations of Ø72 (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.

Following 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 arg1 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 **arg1** 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+Ø22). This does not happen if ? is used for text or instructions and / is used for data.

APPENDIX A
PROGRAM EXAMPLES

Example 1

```
char *charp = "this is a sentence";

main( argc, argv )
int argc;
char **argv;
{
        int     fd;
        char cc;
        if (argc < 2 )
        {
                printf("Input file missing\n");
                exit(8);
        }
        if ( (fd = open(argv[1],Ø))== -1)
        {
                printf("%s : not found\n", argv[1]);
                exit (8);
        }
        charp = "hello";
printf("debug 1 %s\n", charp );
       while( charp++ )
                write (fd, *charp, 1);
}
***1*    **
```

Example 2

```
adb a.out core

ADB: S8000 1.2
? $c
no process
? $C
no process
? $r
r0      %0000
r1      %0000
r2      %0000
r3      %0000
r4      %0000
r5      %0000
r6      %0000
r7      %0000
r8      %0000
r9      %0000
r10     %0000
r11     %0000
r12     %0000
r13     %0000
r14     %0000
sp      %0000
fcw     %0000
pc      %0000
_main:          jr              _main+%7c
? $e
_environ:        %ffbc
_charp:          %1400
__iob:           %113c
__sobuf:         %0000
__lastbu:        %0f26
__sibuf:         %0000
nd:              %133c
_end:            %0000
_deverr          %0000
_errno:          %0009
? $m
? map             'a.out'
        b1 = %0         e1 = %f3a       f1 = %28
        b2 = %0         e2 = %f3a       f2 = %28
/ map             'core'
        b1 = %0         e1 = %1400      f1 = %400
        b2 = %fa00      e2 = %10000     f2 = %1800
```

```
? *charp/s _end+%c4:
data address not found
? charp/s
_charp:
? main.argc/d
Sorry, local variable names not implemented
? $q
***1*    **
```

Example 3

```
int fcnt, gcnt, hcnt;
h(x,y)
{
        int hi; register int hr;
        hi = x+1;
        hr = x-y+1;
        hcnt++;
        f(hr,hi);
}

g(p,q)
{
        int gi; register int gr;
        gi = q-p;
        gr = q-p+1;
        gcnt++;
        h(gr,gi);
}

f(a,b)
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++;
        g(fr,fi);
}

main()
{
        f(1,1);
}
***1*    **
```

Example 4

```
adb

ADB:  S8000 1.2
:r
?.,$c
_f()
_g()
_h()
_f()
:
? ,5$C
Local variables not implemented
_h()
        stack frame:
                %02f6: %0000
                %02f8: %0000
                %02fa: %0000
                %02fc: %0000
                %02fe: %007a (return address)
_g()
        stack frame:
                %0300: %21b4
                %0302: %10db
                %0304: %10d9
                %0306: %10db
                %0308: %00ae (return address)
_f()
        stack frame:
                %030a: %10d9
                %030c: %0002
                %030e: %21b4
                %0310: %0002
                %0312: %0048 (return address)
_h()
        stack frame:
                %0314: %10d7
                %0316: %10d8
                %0318: %10d9
                %031a: %10d8
                %031c: %007a (return address)
_g
                %031e: %21b0
                %0320: %10d9
                %0322: %10d7
                %0324: %10d9
                %0326: %00ae (return address)
? fcnt/d
_fcnt:          2157
? gcnt/d
_gcnt:          2157
```

```
? hcnt/d
 hcnt:          2157
? h.x/d
Sorry, local variable names not implemented
? $q
***1*    **
```

Example 5

```
#define MAXLINE         80
#define YES             1
#define NO              0
#define TABSP           8

char input[] = "data";
int tabs[ MAXLINE ];

main()
{
        int fd;
        int col, *ptab;
        char c;
        ptab = tabs;
        settab(ptab);
        col = 1;
        if ((fd = open(input, 0 )) == -1 )
        {
                printf("%s : not found\n", input );
                exit( 8 );
        }

        while(read(fd, &c, 1) > 0 )
        {
                switch(c)
                {
                        case '\t':
                        while(tabpos(col) != YES)
                        {
                        putchar( ' ' );
                        col++;
                        }
                        break;
                        case '\n':
                        putchar('\n');
                        col = 1;
                        break;
                        default:
                        putchar(c);
                        break;
                }
        }
}

tabpos(col)
int col;
{
        if (col > MAXLINE )
                return(YES);
        else
```

```
                return(NO);
}

settab(tabp)
int *tabp;
{
        int i;

        for ( i=Ø; i <=MAXLINE; i++ )
                (i % TABSP ) ? (tabs[i] = NO  :  (tabs[i] =
YES);
}
***1*    **
```

Example 6

```
adb a.out -

ADB: S8000 1.2
? settab:b
? open:b
? read:b
? tabpos:b
? $b
breakpoints
count    bkpt            command
1        _tabpos
1        _read
1        _open
1        _settab
? settab,5?ia
_settab:        jr      _settab%48
_settab+%2:     clr     %0002(sp)
_settab+%6:     cp      %0002(sp),#%0050
_settab+%c:     jr      gt,_settab+%44
_settab+%e:     ld      r3,%0002(sp)
_settab+%12:
? settab,5?i
_settab:        jr      _settab+%48
                clr     %0002(sp)
                cp      %0002(sp),#%0050
                jr      gt,_settab+%44
                ld      r3,%0002(sp)
? :r
fig5:  running
breakpoint      _settab:    jr      _settab+%48
? settab:d
? :c
fig5:  running
breakpoint      _open:      ld      r0,r7
? $C
_open()
                stack frame:
                            %ffb2: %0048 (return address)
_main()
                stack frame:
                            %ffb4: %0000
                            %ffb6: %0001
                            %ffb8: %0fd4
                            %ffba: %0000
                            %ffbc: %0022 (return address)
? tabs/8x
_tabs:  %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
        %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
        %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
? :c
```

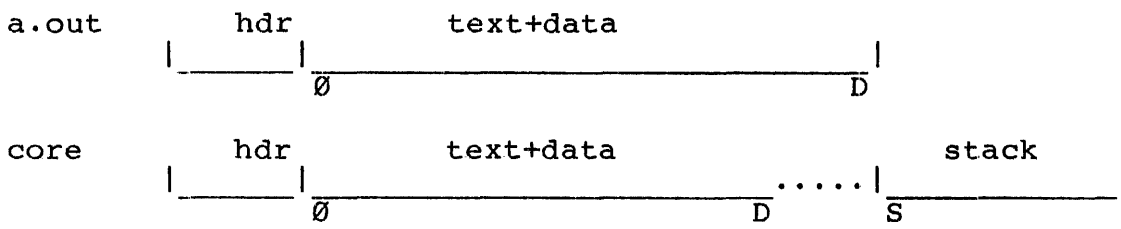
```
fig5:  running
breakpoint      _read:          ld      r0,r7
? :c
fig5:  running
breakpoint      _read:          ld      r0,r7
? tabpos:d
? settab:b settab,5?ia
? settab,5:b settab,5?ia;  0
? read,3:b tabs/8x
? $b
breakpoints
count    bkpt           command
3        _read          tabs/8x
1        _settab        settab,5?ia;  0
1        _open
?:c
? fig5:  running
T_tabs:   %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
h_tabs:   %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
i_tabs:   %0001  %0000  %0000  %0000  %0000  %0000  %0000  %0000
breakpoint      _read:          ld      r0,r7
? $q
***1*    **
```

Example 7

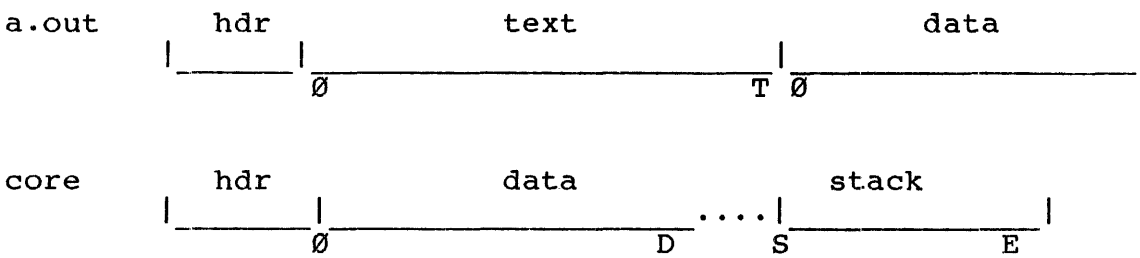
```
adb ex3 -

ADB: S8000 1.2
? h:b hcnt/d; h.hi/; h.hr/
? g:b gcnt/d; g.gi/; f.fr/
? :r
ex3:  running
_gcnt:                  Ø
Sorry, local variable names not implemented
? f:b fcnt/d; f.a/"a = "d; f.h/"b = "d; f.fi/"fi = "d
? g:b gcnt/d; g.p/"p = "d; g.q/"q = "d; g.gi/"gi = "d
? h:b hcnt/d; h.x/"x = "d; h.y/"y = "d; h.hi/"hi = "d
? :r
ex3:  running
_fcnt:                  Ø
Sorry, local variable names not implemented
? $q
***1*    **
```

Example 8

E7Ø7 files



E711 files (separated I and D space)



The following adb variables are set.

		E7Ø7	RM	E711
b	base of data	Ø	b	Ø
d	length of data03/28/83 17:20:05D-B	D		
s	length of stack	S	S	S
t	length of text	Ø	T	T

Example 9

```
adb mapE707 coreE707

ADB:  S8000 1.1
? $m
? map          'mapE707'
                        b1 = %0     e1 = %dc     f1 = %38
                        b2 = $0     e2 = %dc     f2 = %38
/ map          'coreE707'
                        b1 = %0     e1 = %100    f1 = %400
                        b2 = %200   e2 = %1000   f2 = %500
? $v
variables
  address
e = %a4
  other
d = %100
m = %e707
s = %fe00
? $q


adb mapE711 coreE711

ABD:  S8000 1.1
? $m
? map          'mapE711'
                        b1 = %0     e1 = %100     f1 = %38
                        b2 = %0     e2 = %0       f2 = %138
/ map          'coreE711"
                        b1 = %0     e1 = %100     f1 = %400
                        b2 = %200   e2 = %10000   f2 = %500
? variables
address
e = %a4
other
d = %100
m = %e711
s = %fe00
t = %100
? $q
***l*    ***
```

Example 1Ø

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

Example 11

```
adb mapE711 coreE711

ADB: S8000 1.1
? <b,-1/8oa
_str1          052150 064563 020151 071440 060440 061550 060562 060543

_str1+%10:     072145 071040 071564 071151 067147 000000 000001 000710

_lnum:         000000 002322 037640 000000 052150 064563 020151 071440

_str2+%8:      072150 062440 071545 061557 067144 020143 064141 071141

_str2+%18:     061564 062562 020163 072162 064556 063400 000000 177662

_environ+%2:   000000 000000 000000 000000 000000 000000 000000 000000

_environ+%12:  000000 000000 000000 000000 000000 000000 000000 000000

_environ+%22:  000000 000000 000000 000000 000000 000000 000000 000000

_environ+%32:  000000 000000 000000 000000 000000 000000 000000 000000

_environ+%42:  000000 000000 000000 000000 000000 000000 000000 000000

_environ+%52:  000000 000000 000000 000000 000000 000000 000000 000000

_environ+%62:  000000 000000 000000 000000 000000 000000 000000 000000

_environ+%72:  000000 000000 000000 000000 000000 000000 000000 000000

_environ+%82:  000000 000000 000000 000000 000000 000000 000000 000000

_environ+%92:  000000 000000 000000 000000 000000 000000 000000 000000

_environ+%a2:  000000 000000 000000 000000 000000 000000 000000 000000
? <b,20/4on^8Cn
_str1:         052150  064563  020151  071440  This is
               060440  061550  060562  060543  a charac
               072145  071040  071564  071151  ter stri
               067147  000000  000001  000710  ng@`@`@`@a@aH
               000000  002322  037640  000000  @`@`@dR? @`@`
               052150  064563  020151  071440  This is
               072150  062440  071545  061557  the seco
               067144  020143  064141  071141  nd chara
               061564  062562  020163  072162  cter str
               064556  063400  000000  177662  ing@`@`@`@2
               000000  000000  000000  000000  @`@`@`@`@`@`@`@`
               000000  000000  000000  000000  @`@`@`@`@`@`@`@`
               000000  000000  000000  000000  @`@`@`@`@`@`@`@`
               000000  000000  000000  000000  @`@`@`@`@`@`@`@`
```

```
                000000  000000  000000  000000  @`@`@`@`@`@`@`@`
                000000  000000  000000  000000  @`@`@`@`@`@`@`@`
                000000  000000  000000  000000  @`@`@`@`@`@`@`@`
                000000  000000  000000  000000  @`@`@`@`@`@`@`@`
                000000  000000  000000  000000  @`@`@`@`@`@`@`@`
                000000  000000  000000  000000  @`@`@`@`@`@`@`@`
? <b,20/4o4^8t8cna
_strl:          052150  064563  020151  071440  This is
_strl+%8:       060440  061550  060562  060543  a charac
_strl+%10:      072145  071040  071564  071151  ter stri
_strl+%18:      067174  000000  000001  000710  ngH
_lnum:          000000  002322  037640  000000  R?
_str2:          052150  064563  020151  071440  This is
_str2+%8:       072150  062440  071545  061557  the seco
_str2+%10:      067144  020143  064141  071141  nd chara
_str2+%18:      061564  062562  020163  072162  cter str
_str2+%20:      064556  063400  000000  177662  ing2
_environ+%2:    000000  000000  000000  000000
_environ+%a:    000000  000000  000000  000000
_environ+%12:   000000  000000  000000  000000
_environ+%1a:   000000  000000  000000  000000
_environ+%22:   000000  000000  000000  000000
_environ+%2a:   000000  000000  000000  000000
_environ+%32:   000000  000000  000000  000000
_environ+%3a:   000000  000000  000000  000000
_environ+%42:   000000  000000  000000  000000
_environ+%4a:   000000  000000  000000  000000
_environ+%52:
? <b,10/2b8t^2cn
_strl:  %0054   %0068     Th
        %0069   %0073     is
        %0020   %0069     i
        %0073   %0020     s
        %0061   %0020     a
        %0063   %0068     ch
        %0061   %0072     ar
        %0061   %0063     ac
        %0074   %0065     te
        %0072   %0020     r
? $q
*** 3168***
```

Example 12

```
adb dir -

ADB:  S8000 1.1
? =nt"Inode"t"Name"
? 0,-1?ut14cn
                Inode Name
%0000:              2  .
                    2  ..
                  102  bin
                  101  usr
                       157  lib
                  164  dev
                  148  etc
                  197  pb.image
                  957  tmp
                  261  zeus3_1.2
? $q


adb /dev/src -

ADB:  S8000 1.1
? ?m 0 %1000000 1024
? 0,-1?"flags"8ton"links,uid,gid"8t3dn"size"8tDn" \
  addr"8t20un"times"8t2Y2na
%0000:  flags    100000
        links,uid,gid        0      0        0
        size               0
        addr          0      0      0       0       0      0      0
           0        0      0      0       0       0      0      0
           0        0      0      0       0
times    1981 Feb 12 13:50:17   1981 Feb 12 13:50:17

1981 Feb 12 13:50:17

%0040:  flags    040755
        links,uid,gid          44      0       0
        size               704
        addr          3    9984    810        0     0      0     0
           0        0        0      0        0     0      0
           0        0        0      0        0
        times      1981  Jul  17  16:58:42      1981  Jul   15
10:10:41

        1981 Jul 15 10:10:41

%0080:  flags    100664
        links,uid,gid          1      25      0
        size               34
```

```
addr        52    12288       Ø      Ø      Ø      Ø      Ø
   Ø        Ø        Ø       Ø      Ø      Ø      Ø      Ø
   Ø        Ø        Ø       Ø      Ø
times     1981 Jul 16 17:Ø6:34    1981 Jul 16 17:Ø4:23

1981 Jul 16 17:94:23
```

APPENDIX B
ADB SUMMARY

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	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 Ø277
hexadecimal	for example %ff

symbols	for example flag _main main.argc
variables	for example <b
registers	for example <pc <rØ
(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

System 8000 Assembly Language Reference Manual

10/14/83

Preface

This manual describes the System 8000 assembly language and serves as the primary reference manual for the System 8000 assembly language programmer.

A brief introduction to the assembler is given in Section 1 followed by four sections that describe the language beginning with language structure and ending with program structure.

Section 2 describes: character set, numbers, identfiers, unary and binary operators and expressions. The basic unit of an assembly language program, the assembly language statement, is presented in Section 3 followed by the available addressing modes and operators in Section 4. Section 5 describes how program structure allows logical grouping of code and relocatability. The appendices contain a summary of the assembler directives, keywords and special characters, Z8000 instruction mnemonics, assembler error messages and debugger support directives.

Invocation of the assembler and the loader/linker is described in the ZEUS Reference Manual (cas(1), ld(1) and sld(1)).

A detailed description of the instruction set, architecture, and hardware-related features of the Z8000 can be found in the publication:

Z8000 CPU Technical Manual, 00-2010

A detailed description of the Z8000 floating point instruction set can be found in the publication:

Floating Point Emulator Package User's Manual, 03-8201

Table of Contents

SECTION 1 GENERAL INFORMATION 1-1

1.1. Assembler Overview 1-1
1.2. Relationship to PLZ/ASM Assembler 1-1
1.3. Implementation Notes 1-2

SECTION 2 LANGUAGE STRUCTURE 2-1

2.1. Introduction 2-1
2.2. Strings 2-1
2.3. Numbers 2-2
2.3.1. Integers 2-2
2.3.2. Floating Point Numbers 2-2
2.4. Identifiers 2-3
2.4.1. Keyword and Local Identifiers 2-3
2.5. Constants 2-4
2.6. Unary and Binary Operators 2-4
2.7. Expressions -- Assembly Time Arithmetic 2-5
2.7.1. Absolute Expressions 2-6
2.7.2. Relocatable Expressions 2-7
2.7.3. External Expressions 2-8

SECTION 3 ASSEMBLY LANGUAGE STATEMENTS 3-1

3.1. Introduction 3-1
3.2. Assembly Language Statements 3-1
3.3. Labels 3-3
3.3.1. Internal Labels 3-3
3.3.2. Global Labels 3-4
3.3.3. Local Labels 3-4
3.3.4. External Labels 3-5
3.3.5. Common Labels 3-5
3.4. Operators 3-5
3.4.1. Assembler Directives 3-5
3.4.2. Direct Assignment 3-6
3.4.3. Data Declarator 3-9
3.4.4. Instructions 3-12
3.4.5. Pseudo Instructions 3-12
3.5. Operands 3-14
3.6. Comments 3-15

SECTION 4 ADDRESSING MODES AND OPERATORS 4-1

4.1. Introduction 4-1
4.2. Addressing Modes 4-1
4.2.1. Immediate Data 4-2
4.2.2. Register Address 4-3
4.2.3. Indirect Register Address 4-4
4.2.4. Direct Address 4-5
4.2.5. Indexed Address 4-6
4.2.6. Relative Address 4-8
4.2.7. Based Address 4-9
4.2.8. Based Indexed Address 4-9
4.3. Segmented Addressing Mode Operators 4-10
4.4. Addressing Mode Directives 4-11

SECTION 5 PROGRAM STRUCTURE 5-1

5.1. Introduction 5-1
5.2. Modules 5-1
5.3. Sections and Areas 5-1
5.3.1. Program Sections 5-3
5.3.2. Absolute Sections 5-4
5.3.3. Common Sections 5-4
5.4. Local Blocks 5-5
5.5. Location Counter 5-6
5.5.1. Location Counter Control 5-6
5.5.2. Line Number Directive 5-7

APPENDIX A SUMMARY OF ASSEMBLER DIRECTIVES A-1

A.1. Introduction A-1

APPENDIX B KEYWORDS AND SPECIAL CHARACTERS B-1

APPENDIX C ASSEMBLER ERROR MESSAGES C-1

APPENDIX D DEBUGGER SUPPORT DIRECTIVES D-1

List of Tables

Table		
2-1	Special Characters Within Strings	2-1
2-2	Floating Point Conversion Operators	2-2
2-3	Unary Operators	2-4
2-4	Binary Operators in Order of Precedence	2-5
3-1	Summary of Language Statement Fields	3-2
3-2	Functional Summary of Assembler Directives ..	3-6
4-1	Segmented Addressing Mode Operators	4-11

SECTION 1
GENERAL INFORMATION

1.1. Assembler Overview

The System 8000 relocating Z8000 assembler, called cas, runs on the System 8000 under the ZEUS operating system. It translates assembly language source programs into object modules that can be either separately executed by the System 8000, or can be linked with other assembler object modules to form a complete program.

An editor is used to create an assembly language source module (file). The source filename should end with the extension .s. Instructions for invoking the assembler are contained in cas(1).

The assembler is a two-pass assembler. During the first pass, the assembler builds the symbol table and creates an intermediate file that is deleted when the assembly is complete. Symbols, which can have a variable length, appear in the symbol table in the order in which they are defined in the assembly language program. During the second pass, the assembler creates a relocatable object module in a.out(5) format and with the default filename a.out.

The relocatability feature of the assembler frees the programmer from memory management concerns during program development (since object code can be relocated in memory) and also allows programs to be developed in modules whose addresses are resolved automatically when the modules are linked.

1.2. Relationship to PLZ/ASM Assembler

With the 3.1 release of the ZEUS operating system, the System 8000 assembler becomes the core assembler for the System 8000. In addition to translating programs written in the language described in this manual, the assembler operates as the back-end processor for the C and other language compilers.

The System 8000 assembler coexists with the Z8000 PLZ/ASM assembler. PLZ/ASM programs, however, cannot be assembled by the System 8000 assembler nor can System 8000 assembler programs be assembled by the PLZ/ASM assembler. Hereafter, the System 8000 assembler will be referred to as simply the

assembler. Any references to the PLZ/ASM assembler will be explicit to avoid confusion between the two assemblers.

1.3. Implementation Notes

Any limitations associated with a particular release of the assembler are noted in the System 8000 ZEUS Reference Manual, cas(1).

SECTION 2
LANGUAGE STRUCTURE

2.1. Introduction

This section describes the basic structure of the assembly language, encompassing numbers, expressions, and unary and binary operators.

2.2. Strings

A string consists of a character sequence enclosed in double quotes (") or single quotes ('). Consecutive strings are concatenated. Strings cannot contain an actual newline character. Table 2-1 describes the special characters that can be used within a string.

Table 2-1 Special Characters Within Strings

Character	Definition
\Ø	null
\n	newline
\t	tab
\b	backspace
\l	linefeed
\r	carriage return
\f	formfeed
\\	backslash
\"	double quote
\'	single quote
\%nn	two hexadecimal digits that form an arbitrary bit pattern

The following are examples of valid strings:

```
"This is a string"
"This is a null terminated string "
"This is a \" double quote within a string"
"This is a \' single quote within a sting"
"This is \n Multi-line \nString"
"Here is a \t tab, \b backspace, and \r carriage return"
"Here is a \f formfeed \\ backslash and \%AB hex %AB"
```

2.3. Numbers

Two types of numbers are supported by the assembler: integers and floating point numbers.

2.3.1. Integers

Integers can be represented in decimal, hexadecimal, octal or binary format. The default representation is decimal. Examples of each representation follow:

```
5Ø23           decimal
%FA2E          hexadecimal
%(8)7726       octal
%(2)1ØØ111Ø1   binary
```

2.3.2. Floating Point Numbers

A floating point number consists of an integer part, a fractional part, and an exponent part. The exponent part is preceded by an "E" or "e". Either the decimal point or the "E" or "e" must be present to form a floating point number. Only decimal digits can be used in a floating point number.

```
3.Ø
.Ø23
3.23
3.23E7
4.5e6
2E7
2.
```

Floating point numbers are always preceded by a floating point conversion operator. These operators, summarized in Table 2-2, operate only on an integer or floating point number. They cannot be used in expressions.

Table 2-2 Floating Point Conversion Operators

Operator	Conversion
^F	Convert to floating double extended
^FD	Convert to floating double
^FS	Convert to floating single

Examples of both valid and invalid use follow:

VALID	INVALID
^F 3.5	^F (3+4)
^Fl0	^FS Ll
^FD 7	2 + ^FD 3.5
^FS 10E7	
^FD3.5	
^F .23E4	

2.4. Identifiers

An identifier is a nonnumeric character followed by a variable number of numeric or nonnumeric characters. In addition to the upper and lower-case letters, a nonnumeric character can be a "_" or "?". In addition to decimal digits, a numeric character can be a ".". Identifiers can be up to 128 characters. Examples of both valid and invalid identifiers follow:

VALID	INVALID
Myname	2label
countl	2_chickens
Ll	.dot
done?	
_end	
label.one	

2.4.1. Keyword and Local Identifiers

Two special forms of identifiers are supported by the assembler: keyword and local identifiers.

Keyword identifiers are a special kind of identifier reserved by the assembler as keywords. One kind of keyword, the assembler directive, is immediately recognized by the assembler as such, because it is always preceded by a period (".").

The remainder of the keyword identifier set consists of instruction mnemonics, flag codes, and condition codes. They are listed in Appendix B.

Keyword identifiers are recognized in either all upper-case or all lower-case:

```
.SEG        LD
.nonseg     ld
.word       INC
.byte       .EVEN
```

Identifiers become local labels when preceded by a "~". For more information on local labels, refer to Section 3.

```
~L1
~1
~jelly
~parm.1
```

2.5. Constants

A constant value is one that doesn't change throughout a program module. Constants can be expressed as strings or as an identifier representing a constant value. Identifiers can take the form of internal, local or global labels as described in Section 3.

2.6. Unary and Binary Operators

To perform assembly-time arithmetic, expressions are formed using unary and binary operators in conjunction with constants and variable names. (Variable names can be used as part of expressions, but not the variables themselves.) In order of precedence, the unary operators are listed in Table 2-3; the binary operators are listed in Table 2-4. Unary operators take precedence over binary operators, but parentheses can be used to override precedence of evaluation in an expression.

Table 2-3 Unary Operators

Operator	Function
+	unary plus
-	unary minus
^B	binary coded decimal
^C	ones complement
^FS	convert to floating single
^FD	convert to floating double
^F	convert to floating extended
^S	segment (see Section 4.3)
^O	offset (see Section 4.3)

Table 2-4 Binary Operators in Order of Precedence

Operator	Function
*	multiply
/	divide
^<	shift left
^>	shift right
^$	bitwise and
^|	bitwise or
^X	bitwise xor
+	binary plus
-	binary minus

2.7. Expressions -- Assembly Time Arithmetic

Arithmetic is performed in two ways in an assembly language program. Run-time arithmetic is done while the program is actually executing and is defined explicitly by an assembly language instruction:

```
SUB  R10, R12          //Subtract the contents of register
                       //12 from the contents of register 10
```

Assembly-time arithmetic is done by the assembler when the program is assembled and involves the evaluation of expressions in operands, such as the following:

```
LD  R0, #(22/7 + X)

JP  Z, LOOP1 + 12

ADD R2, #HOLDREG-1
```

Assembly-time arithmetic is more limited than run-time arithmetic in such areas as signed versus unsigned arithmetic and the range of values permitted. Only unsigned arithmetic is allowed in assembly-time expression evaluation. Run-time arithmetic uses both signed and unsigned modes, as determined from the assembly-language instruction specified and the meaning attached to operands by the programmer.

All assembly-time arithmetic is computed using 32-bit arithmetic, "modulo 4,294,967,296" (2 raised to the thirty-second power). Values greater than or equal to 4,294,967,296 are divided by 4,294,967,296 and the remainder of the division is used as the result. Depending on the number of bits required by the particular instruction, only the rightmost

4, 8, 16, or 32 bits of the resulting 32-bit value are used. If the result of assembly-time arithmetic is to be stored in four bits, the value is taken "modulo 16" to give a result in the range Ø to 15. If the result is to be stored in a single byte location, the value is taken "modulo 256" to give a result in the range Ø to 255. If the result is to be stored in a word, the value is taken "modulo 65536" to give a result in the range Ø to 65535.

```
LDB   RL4, #X+22          //Result of (X+22) must be in
                          //range Ø to 255

JP    X+22                //Modulo 65536. Result is the
                          //address 22 bytes beyond X
                          //and may wraparound through
                          //zero

ADDL  RR12, #32ØØØ*MAX    //Result of (32ØØØ*MAX) is
                          //taken modulo 4,294,967,296
```

All arithmetic expressions have a mode associated with them: absolute, relocatable and external. In the following discussions, these abbreviations are used:

AB -- absolute expression
RE -- relocatable expression
EX -- external expression

2.7.1. Absolute Expressions

An absolute expression consists of one or more numbers, or absolute constants combined with unary or binary operators. The difference between two relocatable expressions is also considered to be absolute. The relocatable expressions must be in the same area of the same section. If they are not, the absolute difference can not be determined at assembly time. (For more information on program sections and areas, see Section 5).

An absolute expression is defined as one of the following:

```
AB -->  a number or absolute constant
        AB <operator> AB
        '+' AB, '-' AB
        RE '-' RE
```

The segmented address constructors "<<" and ">>" can be used in an absolute expression to form a long value. For example:

```
<<3>>%100
```

is equivalent to the long value

```
%03000100
```

and can be used in any expression where long values can be used.

Strings can also be used as absolute values. However, only the first four characters of a string are used to form the absolute value.

At instruction assembly time, any absolute segmented direct address that does not have zeroes in the lower byte of the segment part is flagged as an error. In addition, the high bit is set for segmented addresses at this time.

Examples of valid absolute expressions (where L1 and L2 are relocatable labels and c1 is a constant identifier) are:

```
%(8)2767 + (3 * 5)
c1 * 6 + %(2)01001100
%FEFEABAB + (L1 - L2)
5 ^< 8
3 + <<2>>%100
4 + "ABCD"
```

Examples of invalid absolute expressions are:

```
2 + L1
(L1 * 3) - L2
c1 + (L1 -3)
```

2.7.2. Relocatable Expressions

A relocatable expression contains exactly one identifier subject to relocation after assembly. The expression can be extended by adding or subtracting an absolute expression. Plus and minus are the only operators allowed, however.

A relocatable expression can be defined as one of the following:

```
RE --> a relocatable identifier
       RE '+' AB
       AB '+' RE
       RE '-' AB
       +RE
```

Examples of valid relocatable expressions (where L1 and L2 are relocatable labels and c1 is constant identifier) are:

```
L1 + c1
L1 + (%(8)Ø77 - %FEØ2 / 2) ^> 4
c1 + (L1 - L2) + L2
L1 - (L1 -L2) + c1
```

Examples of invalid relocatable expressions are:

```
c1 - L1
(%2Ø3F) - 1ØØ - L2
(L1 - L2) * L2
L1 / L2
L1 + (L2 + c1)
```

Ø

2.7.3. External Expressions

An external expression contains exactly one external identifier, possibly extended by adding or subtracting an absolute expression. An external identifier is one that is used in the current module but defined in another module. The value of an external identifier is not known until the modules are linked.

An external expression is defined as one of the following:

```
EX -->  external identifier
        EX '+' AB
        AB '+' EX
        EX '-' AB
        +EX
```

Examples of valid external expressions (where L1 is a relocatable label, c1 is a constant identifier, and e1 is an external label) are:

```
e1 - c1
e1 - (L1 - L2) + 5
c1 + e1
(%3Ø4 - 5) + e1
%(2)Ø11ØØ111 * 2 + e1
```

Examples of invalid external expression are:

```
el + (Ll - el)
%FEFE - el
cl * 2 + el - Ll
2 * el
el ^> 8
```

SECTION 3
ASSEMBLY LANGUAGE STATEMENTS

3.1. Introduction

This section describes the fields and syntax of the assembly language statement. The conventions used in describing the syntax are as follows:

- Parameters shown within angle brackets represent items to be replaced by actual data or names: <section_name>
- Optional items are enclosed in parentheses: (<expression>)
- Parameters separated by a "|" indicate that one or the other parameter can be used but not both.
- Possible repetition of an item is indicated by appending a "+" (to signify one or more repetitions) or an "*" (to signify zero or more repetitions) to the item: (<expression>)* Each repetition after the first must be preceded by a comma.
- Other special characters shown in statement and command formats such as :=, (), will be enclosed in single quotes and must be written as shown.
- The special symbol ":=" means "is defined as" or "is assigned". Any label assigned a value using this construct cannot be redefined later.

3.2. Assembly Language Statements

Assembly language programs consist of assembly language statements, which can have up to four fields:

Label field -- symbolically defines a location in a program.

Operator field -- specifies the action to be performed by the statement.

Operand field -- contains the data or the address of the data to be operated upon.

Comment field -- contains a comment to document the action of the statement.

Table 3-1 summarizes these fields which are described in detail in the remainder of this section.

Each field must be separated from the other fields by one or more delimiters. A delimiter can be one of the following:

space
tab
semicolon

A comma is required to separate components in the operands field.

Each assembly language statement is terminated by the new-line or carriage return character. When a statement's length exceeds the line length, it can be continued on the next line by using the line continuation character "\".

A sample assembly language statement follows:

```
Label   Operator   Operand(s)          Comment

L1:     LD         RØ, R1       //Load the contents of
                                //Register Ø in Register 1.
```

Table 3-1 Summary of Language Statement Fields

Field	Field Types
Label	Internal Global Local External Common
Operator	Directive Direct Assignment Data Declarator Instruction
Operands	Address Data Condition Code
Comment	

Note that the order of fields shown in the example is not required. While comments are always the last field in a statement (when they are used), labels do not necessarily precede operators. When the operator is a directive, for example, a label can follow:

```
.extern L1
```

3.3. Labels

A label identifies a statement in a program allowing that statement to be referenced symbolically. Constants, instructions, directives, and data declarators can all be labeled. Any statement referenced by another statement must be labeled. There can be more than one label per statement. The following label types fit this description:

internal
global
local

Two additional label types, external and common, are defined with the assembler directives, .extern and .comm respectively. They are distinguished by the fact that they can be referenced in the current module (file) but are defined as global in another module.

external
common

3.3.1. Internal Labels

An internal label consists of an identifier followed by a ":". An internal label restricts access to the identifier to the module in which it is defined.

```
start:    LD R0,R1          //an internal label for an
                            //instruction

count_1: .word %200         //an internel label for a
                            //data declaration.

begin:  .psec mysection     //an internal label for an
                            //assembler directive
```

3.3.2. Global Labels

A global label consists of an identfier followed by "::" It allows the identifier to be accessed from modules other than the one where it is defined.

```
L1::L2::.word %ABCD      //two global labels for
                         //a data declaration

L1::
L2::     .word %ABCD     //same as preceding example

done?:: PUSH @R15, RØ    //a global label for an
                         //instruction

_start::.psec            //a global label for a directive

foobar::= %2Ø            //a global constant with
                         //value %2Ø
```

A label by itself on a line is considered a null statement. Such a statement is associated with the next non-null statement in the program.

```
start::                  //null statement consisting
                         //of a label only

begin::  .code           //mark beginning of code area
```

3.3.3. Local Labels

A local label consists of an identifier preceded by a "~" (making the identifier a local symbol) and followed by a ":". (Local labels are valid only within local blocks as described in Section 5, Program Structure.)

```
~L1:~L2: LD RØ, #2Ø      //two local labels for
                         //an instruction

~num:= %1ØØ              //a local constant with
                         //value 1ØØ (hex)

~count:.odd              //a local label for a directive
```

3.3.4. External Labels

External specifies that a label can be referenced in the current module but is defined as global in another module. External labels are defined with the external directive, .extern.

```
.extern procl, done?    //external labels are declared
.extern datum, _end
```

3.3.5. Common Labels

Common labels consist of the .comm directive followed by a constant expression that indicates the number of bytes of storage associated with the common symbol(s) and a comma. These are followed by a list of identifiers separated by commas. At link time, common symbols with the same name but from different files are inspected. The common label with the largest size is allocated as uninitialized data (BSS storage). If a global definition with the same name is found, all common labels refer to the global definition.

```
.comm 2Ø, datal, data2      //two common symbols of size 2Ø

.comm  5+3, myname,         //common symbol of size 8
```

3.4. Operators

The operator field specifies the action to be performed by the statement. This field can contain one of the following:

directive
direct assignment
data declarator
instruction

3.4.1. Assembler Directives

An assembler directive either directs the operation of the assembler or allocates storage but does not itself result in executable code. A period, ".", precedes every assembler directive. Table 3-2 gives a functional summary of the directives and a reference to the section containing a description of the directive and examples of its use.

Table 3-2 Functional Summary of Assembler Directives

Category	Directives	See
Data Storage and Initialization Directives	.byte .word .long .quad .extend .addr .blkb .blkw .blkl	Section 3
Label Control Directives	.comm .extern	
Segment Control Directives	.seg .nonseg	Section 4
Program Section Directives	.psec .csec .asec .data .bss .code	Section 5
Location Counter Control Directives	.even .odd	
Listing Directive	.line	

3.4.2. Direct Assignment

A direct assignment statement allows symbols to be associated with constants, labels, or keywords. Specifically, a direct assignment statement is a symbol (usually a label) followed by a "=" and one of the following:

32-bit absolute constant
32, 64, or 80 bit floating point constant
Relocatable expression
Location Counter
Keyword (for keyword redefinition)

32-Bit Absolute Constants

A internal, global, or local label can be assigned the value of a 32-bit constant expression.

```
c1:= 20                 //internal label c1 is assigned the
                        //constant value 20.

c3::= 2+3*5             //global label c3 is assigned the
                        //the constant value 17.

~c4:=L1-L2              //local label ~c4 is assigned the
                        //absolute difference between
                        //label L1 and L2

c5:= <<4>>%1020         //internal label c5 is assigned the
                        //long value %04001020
```

Floating Point Constants

An internal or local label (but not a global label) can be assigned the value of a 32, 64, or 80-bit floating point constant. The floating point constant can be a constant expression or floating point number preceded by a floating point type conversion unary operator as described in Section 2. Floating point constants can only replace floating point numbers.

```
L3:= ^F 3               //internal label L3 is assigned
                        //the extended floating point
                        //representation of 3.

glbl:= ^FS 3.5          //internal label glbl is assigned
                        //the single floating point
                        //representation of 3.5.

~loc2:= ^FD 2.23E7      //local label ~loc2 is assigned
                        //the double floating point
                        //representation of 2.23E7.
```

Relocatable Expressions and Symbols

An internal, global, or local label can be assigned the value of a relocatable expression.

```
c1:= .+2            //internal label c1 is assigned
                    //the value of the current
                    //location counter plus 2.

g3::= L47-%3Ø        //global label g3 is assigned
                    //the address of label L47-%3Ø.

~dum:= 6Ø+start      //local label ~dum is assigned
                    //the value 6Ø plus the address
                    //of label start.
```

NOTE

If assembled in segmented mode, ".", "L47", and "start" are full segmented addresses.

Location Counter Control

The location counter symbol "." can be assigned the value of a constant expression, a relocatable expression, or a location counter relative expression.

```
.=.+1Ø              //the location counter is increased
                    //by 1Ø

.=2Ø                //the location counter is assigned
                    //the value 2Ø

.=.-(3+5)           //the location counter is decreased
                    //by 8

.= L2 + 1Ø          //the location counter is set to
                    //1Ø bytes beyond the symbol L2
```

Keyword Redefinition

A local or internal label, but not a global label, can be associated with a keyword for purposes of keyword redefinition. Keyword redefinition gives the label all the attributes of the keyword being assigned to it.

```
location:= .            //the internal label location
                        //is synonymous with "."

sdefault:= .psec        //the internal label sdefault
                        //is synonymous with .psec

~wval:= .word           //the local label ~wval is
                        //synonymous with .word

~p1:=RØ                 //the local label ~p1 is
                        //now synonymous with the
                        //register RØ
```

3.4.3. Data Declarator

A data declaration statement allocates and initializes storage. Such a statement consists of a data declaration directive preceded by a label (optional) and followed by a series of constant and relocatable expressions.

The nine data declaration directives are:

```
.byte
.word
.long
.quad
.extend
.addr
.blkb
.blkw
.blkl
```

.byte (<number> '('<expression>')'|<expression>| '"'string'"')*

Allocates storage and initializes it with the specified byte value(s) which can be a series of constant and relocatable expressions or an ascii string. Number is the repetition factor. When a number is specified, the expression must be enclosed in parentheses; strings are enclosed in double quotes:

```
name:.byte "Babe Ruth"    //allocates storage for ascii
                          //representation of named string

place:.byte "Anytown"\    //Continues a long string onto
            "USA"         //the next line

L4::.byte 3, "joe"        //allocates four bytes with initial
                          //value three and ascii string joe
```

.word (<number> '('<expression>')'|<expression>)*

Allocates storage and initializes it with the specified word value(s) which can be a series of constant and relocatable expressions. Number is the repetition factor. When a number is specified, the expression must be enclosed in parentheses.

```
count:.word %20           //allocates a word with initial
                          //value %20

L2:.word 20, 3+5, 5       //allocates three words with initial
                          //values 20,8, and 5
```

.long (<number> '('<expression>')'|<expression>)*

Allocates storage and initializes it with the specified long value(s) which can be a series of constant and relocatable expressions. Number is the repetition factor. When a number is specified, expression must be enclosed in parentheses.

```
.long 10 (%ABCDABCD)      //allocates 10 long values
                          //with initial value %ABCDABCD
```

.quad (<number> '('<expression>')'|<expression>)*

Reserves 64 bits of storage. Only double precision floating point numbers fill the allocated storage completely. If the value does not fill the allocated storage completely, no sign extension is performed. Number is the repetition factor. When a number is specified, the expression must be enclosed in parentheses.

```
.quad %FFFFFFFF          //initializes the lower 32 bits
                         //of the quad with %FFFFFFFF

.quad ^FS3.5             //initializes the lower 32 bits
                         //of the quad with the floating
                         //point value 3.5

.quad ^FD3.4             //initializes entire quad with
                         //double floating point number 3.4
```

.extend (<number> '('<expression>')'|<expression>)*

Allocates 8Ø bits of storage. Only extended precision floating point numbers fill the allocated storage completely. If the value does not fill the allocated storage completely, no sign extension is performed. Number is the repetition factor. When a number is specified, the expression must be enclosed in parentheses.

```
.extend 1Ø (^F1.234E5)   //allocates 1Ø extended floating
                         //point numbers with the value
                         //1.234E5
```

.addr (<number> '('<expression>')'|<expression>)*

When assembling in non-segmented mode, allocates storage and initializes it with the specified 16-bit value which can be a series of constant and relocatable expressions. Number is a repetition factor. When a number is used, expression must be enclosed in parentheses.

When assembling in segmented mode, allocates storage and initializes it with a 32-bit value.

```
.addr L2                 //allocates two (non-segmented) or
                         //four (segmented) bytes for
                         //address L2
```

.blkb <expression>

Allocates storage in bytes. The number of bytes is specified by the expression. No initialization occurs.

```
.blkb 20                //allocates storage for 20 bytes
```

.blkw <expression>

Allocates storage in words. The number of words is specified by the expression. No initialization occurs.

```
.blkw (3+5)             //allocates storage for 8 words
```

.blkl <expression>

Allocates storage in long words. The number of long words is specified by the expression. No initialization occurs.

```
cl:= 20                 //defines constant
.blkl (2*cl)            //allocates storage
                        //for 40 long words
```

Commas are required in initializer lists; consider this data declarator:

```
.byte 2 -3
```

It has one value, 2-3. If two values are to be initialized, use a comma:

```
.byte 2, -3
```

3.4.4. Instructions

An instruction is the assembly language mnemonic describing a specific action to be taken. Instructions comprising the Z8000 instruction set are described in the Z8000 CPU Technical Manual. The floating point instruction set is described in the Floating Point Emulator Package User's Manual.

3.4.5. Pseudo Instructions

The majority of code in the assembly language program will normally be assembler directives, data declarators, direct assignment statements, the assembly language instructions

described in the Z8000 CPU Technical Manual, and the floating point instructions described in the Floating Point Emulator Package User's Manual.

The System 8000 is capable of performing jump and call optimization. When the assembler encounters the pseudo jump and call instructions (JPR and CALLR), it determines the range of the jump or call and produces the relative (short) form of the instruction (JR or CALR) wherever possible. If it cannot produce the relative form of the instruction, it produces the absolute (long) form of the instruction (JP or CALL).

Jump Optimization

Jump optimization is explicitly provided to the programmer via a JPR control instruction with the following form:

```
JPR [cc] ',' <jpr_expr>
```

where:
cc is any condition code that can be used with a JP or JR instruction

```
<jpr_expr> => <label> [('+'| '-') <const_expr>]
```

where:
<label> is an internal, global, or local label.
<const_expr> is a constant expression

A JPR (<jpr_expr>) expression is a relocatable expression containing exactly one relocatable value (<label>). The destination of a JPR must be a program label with an optional constant added to or subtracted from it. However, one particular form of <label> + <const_expr> cannot be optimized. This form is best explained by the following example:

```
L1:   JPR   L2-300
        .
        .
        .
L3:   JPR   L99
        .
        .
        .
L2:
```

If L2-L1 is less than 300 bytes, the JPR at L1 is actually a backward jump. The destination actually becomes further

away if the JPR at L3 is optimized. This case is very expensive to handle and is rare enough not to be optimized.

Certain JPR and CALLR instructions cannot be optimized to a short (relative) instruction. The following types of statements cannot be optimized:

1. A JPR or CALLR instruction whose target is not in the same section.

2. A JPR or CALLR instruction whose target is not in the same module (external).

During the course of assembly, it is possible to encounter a statement that cannot be assembled unless jump optimization has occured. If jump optimization is performed before the target label for a particular jump is found, the jump is made long. The following conditions cause jump optimization to occur before the end of the first pass of the assembler.

1. Location counter direct assignment (as in .=.+20)

2. A constant expression that contains the difference between two relocatable values (for example, L1-L2) when there is an optimizable jump between the two relocatable values.

Call Optimization

The assembler also provides call optimization via a CALLR control instruction that will produce a relative call whenever possible. The call control instruction has the following form:

```
CALLR <jpr_expr>
```

where <jpr_expr> is a simple, relocatable expression, as described previously.

Calls are optimized under the same conditions that cause jump optimization.

3.5. Operands

Operands supply the information an instruction needs to carry out its action. Depending on the instruction specified, this field can have zero or more operands. An operand can be:

- Data to be processed (immediate data).
- The address of a location from which data is to be taken (source address).
- The address of a location where data is to be put (destination address).
- The address of a program location to which program control is to be passed.
- A condition code, used to direct the flow of program control.

Although there are a number of valid combinations of operands, there is one basic convention to remember: the destination operand always precedes the source operand. Refer to the specific instructions in the Z8000 CPU Technical Manual for valid operand combinations.

With the exception of immediate data and condition codes (described in the Z8000 CPU Technical Manual and Floating Point Evaluator Package User's Manual), all operands are expressed as addresses: register, memory, and I/O addresses. For example, an operand may name a register whose contents are added to the contents of another register to form the address of the memory location containing the source data (based indexed addressing).

Addressing modes and operators are the subject of Section 4.

3.6. Comments

Comments are used to document program code as a guide to program logic and also to simplify present or future program debugging. Two types of comments are available: the end-of-line comment and the multi-line comment. The end-of-line comment begins with the characters "//" and ends at the next carriage return.

```
LD RØ, Rl                //This is an end-of-line comment
```

The multi-line comment begins with the characters "/*", ends with the characters "*/" and spans one or more lines in between.

```
LD RØ, Rl                /* This is an example of a
                         ** multi-line comment
                         */
```

SECTION 4
ADDRESSING MODES AND OPERATORS

4.1. Introduction

This section describes the System 8000 addressing modes and operators and contains examples of assembler instructions that use them.

4.2. Addressing Modes

Data can be specified by eight distinct addressing modes:

- Immediate Data
- Register
- Indirect Register
- Direct Address
- Indexed Address
- Relative Address
- Based Address
- Based Indexed Address

Special characters are used in operands to identify certain of these address modes. The characters are:

- "R" preceding a word register number;
- "RH" or "RL" preceding a byte register number;
- "RR" preceding a register pair number;
- "RQ" preceding a register quadruple number;
- "@" preceding an indirect-register reference;
- "#" preceding immediate data;
- "()" used to enclose the displacement part of an indexed, based, or based indexed address;
- "." signifying the current program counter location, usually used in relative addressing.

The use of these characters is described in the following sections.

Not every address mode can be used by every instruction. The individual instruction descriptions in the Z8000 CPU Technical Manual tell which address modes can be used for each instruction.

4.2.1. Immediate Data

Although considered an addressing mode for purposes of this discussion, Immediate Data is the only mode that does not indicate a register or memory address.

The operand value used by the instruction in Immediate Data addressing mode is the value supplied in the operand field itself.

Immediate data is preceded by the special character "#" and can be either a constant expression (including character constants and symbols representing constants) or a relocatable expression. Immediate data expressions are evaluated using 32-bit arithmetic. Depending on the instruction being used, the value represented by the rightmost 4, 8, 16, or 32 bits is actually used. An error message is generated for values that overflow the valid range for the instruction.

```
LDB    RH0, #100     //Load decimal 100 into byte
                     // register RH0

LDL    RR0, #%8000 * REP_COUNT
                     //Load the value resulting from
                     //the multiplication of hexadecimal
                     //8000 and the value of constant
                     // REP_COUNT into register pair RR0
```

If a variable name or address expression is prefixed by "#", the value used is the address represented by the variable or the result of the expression evaluation, not the contents of the corresponding data location. In non-segmented mode, all address expressions result in a 16-bit value.

For segmented addresses, the assembler automatically creates the proper format for a long offset address which includes the segment number and the long offset in a 32-bit value. It is recommended that symbolic names be used wherever possible since the corresponding segment number and offset for the symbolic name will be managed automatically by the assembler and can be assigned values later when the module is linked or loaded for execution.

For those cases where a specific segment is desired, the following notation can be used (the segment designator is enclosed in double angle brackets):

```
<<segment>>offset
```

where "segment" is a constant expression that evaluates to a 7-bit value, and "offset" is a constant expression that evaluates to a 16-bit value. This notation is expanded into a long offset address by the assembler.

4.2.2. Register Address

In Register addressing mode, the operand value is the content of the specified general-purpose register. There are four different sizes of registers on the Z8000:

- Word register (16 bits),
- Byte register (8 bits),
- Register pair (32 bits), and
- Register quadruple (64 bits).

A word register is indicated by an "R" followed by a number from 0 to 15 (decimal) corresponding to the 16 registers of the machine. Either the high or low byte of the first eight registers can be accessed by using the byte register constructs "RH" or "RL" followed by a number from 0 to 7. Any pair of word registers can be accessed as a register pair by using "RR" followed by an even number between 0 and 14. Register quadruples are equivalent to four consecutive word registers and are accessed by the notation "RQ" followed by one of the numbers 0, 4, 8, or 12.

If an odd register number is given with a register pair designator, or a number other than 0, 4, 8, or 12 is given for a register quadruple, an assembly error will result.

In general, the size of a register used in an operation depends on the particular instruction. Byte instructions, which end with the suffix "B" are used with byte registers. Word registers are used with word instructions, which have no special suffix. Register pairs are used with long word instructions, which end with the suffix "L". Register quadruples are used only with three instructions (DIVL, EXTSL and MULTL) which use a 64-bit value. An assembly error will occur if the size of a register does not correspond correctly with the particular instruction.

```
LD     R5, #%3FFF        //Load register 5 with the
                         //hexadecimal value 3FFF

LDB    RH3, #%F3         //Load the high order byte of
                         //word register 3 with the
                         //hexadecimal value F3

ADDL   RR2, RR4          //Add the register pairs 2-3 and
                         //4-5 and store the result in 2-3

MULTL RQ8, RR12          //Multiply the value in register
                         //pair 10-11 (low order 32 bits of
                         //register quadruple 8-9-10-11) by
                         //the value in register pair 12-13
                         //and store the result in register
                         //quadruple 8-9-10-11
```

4.2.3. Indirect Register Address

In Indirect Register addressing mode, the operand value is the content of the location whose address is contained in the specified register. A word register is used to hold the address in non-segmented mode, whereas a register pair must be used in segmented mode. Any general-purpose word register (or register pair in segmented mode) can be used except RØ or RRØ.

Indirect Register addressing mode is also used with the I/O instructions and always indicates a 16-bit I/O address. Any general-purpose word register can be used except RØ.

An Indirect Register address is specified by a "commercial at" symbol (@) followed by either a word register or a register pair designator. For Indirect Register addressing mode, a word register is specified by an "R" followed by a number from 1 to 15, and a register pair is specified by a "RR" followed by an even number from 2 to 14.

```
JP     @R2               //Pass control (jump) to the
                         //program memory location
                         //addressed by register 2
                         //(non-segmented mode)

LD     @R3, R2           //Load contents of register
                         //2 into location addressed by
                         //register 3 (non-segmented mode)
```

```
LD    @RR2, #30          //Load immediate decimal value 30
                         //into location addressed by regis-
                         //ter pair 2-3 (segmented mode)

state2: CALL @R3         //Call indirect through register 3
                         //(non-segmented mode)
```

4.2.4. Direct Address

The operand value used by the instruction in Direct addressing mode is the content of the location specified by the address in the instruction. A direct address can be specified as a symbolic name of a memory or I/O location, or an expression that evaluates to an address. For non-segmented mode and for all I/O addresses, the address is a 16-bit value. In segmented mode, the memory address is either a 16-bit value (short offset) or a 32-bit value (long offset). All assembly-time address expressions are evaluated using 32-bit arithmetic, with only the rightmost 16 bits of the result used for non-segmented addresses.

```
LD  R10, datum           //Load the contents of the
                         //location addressed by datum
                         //into register 10

LD  struct+8, R10        //Load the contents of register
                         //10 into the location addressed
                         //by adding 8 to struct

JP  C, %2F00             //Jump to location %2F00 if the
                         //carry flag is set (non-segmented
                         //mode)

INB  RH0, 77             //Input the contents of the I/O
                         //location addressed by decimal
                         //77 into RH0

L2:: INC count, #2       //Increment instruction with
                         //direct address "count" and
                         //immediate value 2
```

For segmented addresses, the assembler automatically creates the proper format which includes the segment number and the offset. It is recommended that symbolic names be used wherever possible, since the corresponding segment number and offset for the symbolic name will be automatically managed by the assembler and can be assigned values later when the module is linked or loaded for execution.

For those cases where a specific segment is desired, the following notation can be used (the segment designator is enclosed in double angle brackets):

```
<<segment>>offset
```

where "segment" is a constant expression that evaluates to a 7-bit value, and "offset" is a constant expression that evaluates to a 16-bit value. This notation is expanded into a long offset address by the assembler.

To force a short offset address, a short offset operator is available which can be used with a direct address in segmented mode only. The short offset operator is a pair of vertical bars "|" which surround the address. For a valid address, the offset must be in the range Ø to 255; the final address includes the segment number and the short offset in a 16-bit value.

NOTE

Since short offset addresses can be relocatable, they are checked for validity at link time.

Examples of the short offset address operator:

```
.seg                          //enter segmented mode
L1:.word  %ABAB               //declare data

.code                         //enter code area
          LD RØ, |L1|         //load register Ø from
                              //short address L1
          CP RØ, %ØD          //compare with %ØD
          JP EQ, |L2+ 1Ø|     //jump to short address
                              //L2 + 1Ø
          ADD RØ, R2          //add RØ to R2
L2:       RET                 //return
```

4.2.5. Indexed Address

An Indexed address consists of a memory address displaced by the contents of a designated word register (the index). This displacement is added to the memory address and the resulting address points to the location whose contents are used by the instruction. In non-segmented mode, the memory address is specified as an expression that evaluates to a 16-bit value. In segmented mode, the memory address is

specified as an expression that evaluates to either a 16-bit value (short offset format) or a 32-bit value (long offset format). All assembly-time address expressions are evaluated using 32-bit arithmetic, with only the rightmost 16 bits of the result used for non-segmented addresses. This address is followed by the index, a word register designator enclosed in parentheses. For Indexed addressing, a word register is specified by an "R" followed by a number from 1 to 15. Any general-purpose word register can be used except RØ.

```
LD  RlØ, table(R3)          //Load the contents of the
                            //location addressed by table
                            //plus the contents of reg-
                            //ister 3 into register lØ

LD  24Ø+38(R3), RlØ         //Load the contents of reg-
                            //ister lØ into the location
                            //addressed by 278 plus the
                            //contents of register 3
                            //(non-segmented mode)

ADD R2, tab(R4)             //Load register 2 with
                            //contents of register 2
                            //added to contents of the
                            //address "tab" indexed
                            //by the value in register 4
```

For segmented addresses, the assembler automatically creates the proper format for the memory address, which includes the segment number and the offset. As with Direct addressing, symbolic names should be used wherever possible so that values can be assigned later when the module is linked or loaded for execution.

For those cases where a specific segment is desired, the following notation can be used (the segment designator is enclosed in double angle brackets):

```
<<segment>>offset
```

where "segment" is a constant expression that evaluates to a 7-bit value, and "offset" is a constant expression that evaluates to a 16-bit value. This notation is expanded into a long offset address by the assembler.

4.2.6. Relative Address

Relative address mode is implied by its instruction. It is used by the Call Relative (CALR), Decrement and Jump If Not Zero (DJNZ), Jump Relative (JR), Load Address Relative (LDAR), and Load Relative (LDR) instructions and is the only mode available to these instructions. The operand, in this case, represents a displacement that is added to the contents of the program counter to form the destination address that is relative to the current instruction. The original content of the program counter is taken to be the address of the instruction byte following the instruction. The size and range of the displacement depends on the particular instruction, and is described with each instruction in the Z8000 CPU Technical Manual.

The displacement value can be expressed in two ways. In the first case, the programmer provides a specific displacement in the form ".+n" where n is a constant expression in the range appropriate for the particular instruction and "." represents the contents of the program counter at the start of the instruction. The assembler automatically subtracts the size of the relative instruction from the constant expression to derive the displacement.

```
JR  OV, .+K     //Add value of constant K to program
                //counter and jump to new location if
                //overflow has occurred!

JR .+4          //Jump relative to program counter "."
                //plus 4
```

In the second method, the assembler calculates the displacement automatically. The programmer simply specifies an expression that evaluates to a number or a program label as in Direct Addressing. The address specified by the operand must be in the valid range for the instruction, and the assembler automatically subtracts the value of the address of the following instruction to derive the actual displacement.

```
DJNZ R5, loop       //Decrement register 5 and jump to
                    //loop if the result is not zero

LDR  R10, data      //Load the contents of the location
                    //addressed by data into register 10
```

4.2.7. Based Address

A based address consists of a register that contains the base and a 16-bit displacement. The displacement is added to the base and the resulting address indicates the location whose contents are used by the instruction.

In non-segmented mode, the based address is held in a word register that is specified by an "R" followed by a number from 1 to 15. Any general-purpose word register can be used except RØ. The displacement is specified as an expression that evaluates to a 16-bit value, preceded by a "#" symbol and enclosed in parentheses.

In segmented mode, the segmented based address is held in a register pair that is specified by an "RR" followed by an even number from 2 to 14. Any general-purpose register pair can be used except RRØ. The displacement is specified as an expression that evaluates to a 16-bit value, preceded by a "#" symbol and enclosed in parentheses.

```
LDL    RR2, R1(#255)        //Load into register pair 2-3 the
                            //long word value found in the
                            //location resulting from adding
                            //255 to the address in register
                            //1 (non-segmented mode)

LD     RR4(#%4000), R2      //Load register 2 into the loca-
                            //tion addressed by adding %4000
                            //to the segmented address found
                            //in register pair 4-5
                            // (segmented mode)

LD     RØ, R2(#1Ø)          //Load register Ø from 1Ø bytes
                            //past the base address in
                            //register 2 (non-segmented mode)
```

4.2.8. Based Indexed Address

Based Indexed addressing is similar to Based addressing except that the displacement (index) as well as the base is held in a register. The contents of the registers are added together to determine the address used in the instruction.

In non-segmented mode, the based address is held in a word register that is specified by an "R" followed by a number from 1 to 15. The index is held in a word register specified in a similar manner and enclosed in parentheses. Any general-purpose word registers can be used for either the base or index except RØ.

In segmented mode, the segmented based address is held in a register pair that is specified by an "RR" followed by an even number from 2 to 14. Any general-purpose register pair can be used except RRØ. The index is held in a word register that is specified by an "R" followed by a number from 1 to 15. Any general-purpose word register can be used except RØ.

```
LD     R3, R8(R15)        //Load the value at the location
                          //addressed by adding the address
                          //in R8 to the displacement in
                          //R15 into register 3 (nonseg-
                          //mented mode)

LDB    RR14(R4), RH2      //Load register RH2 into the
                          //location addressed by the
                          //segmented address in RR14
                          //indexed by the value in R4
                          //(segmented mode)

init:  LD RØ, R2(R4)      //Load into register Ø
                          //the base address
                          //in register 2 indexed
                          //by the value in register 4
                          //(non-segmented mode)
```

4.3. Segmented Addressing Mode Operators

Two special operators, summarized in Table 4-1, ease the manipulation of segmented addresses. While addresses can be treated as a single value with a symbolic name assigned by the programmer, occasionally it is useful to determine the segment number or offset associated with a symbolic name.

The "^S" unary operator is applied to an address expression that contains a symbolic name associated with an address, and returns a 16-bit value. This value is the 7-bit segment number associated with the expression and a one bit in the most significant bit of the high-order byte, and all zero bits in the low-order byte.

The "^S" operator can be used in segmented mode only.

The "^O" unary operator is applied to an address expression and returns a 16-bit value that is the offset value associated with the expression.

The offset operator can be used in either segmented or non-segmented mode, but has no effect in non-segmented mode.

Because of the special properties of these address operators, no other operators can be applied to a subexpression containing a segment or offset operator, although other operators can be used within the subexpression to which either is applied.

```
.seg                        //segmented mode
L1:.word %ABCD              //declare data

.code                       //enter the code area
        LD R4, #^S L1       //load the segment value
        LD R5, #^O L1       //load the offset value
        LD R3, @RR4         //load indirect through RR4
.nonseg                     //non-segmented mode
L2:.word %BBCC              //declare data

.code                       //enter the code area
        LD R5, #^O L1       //offset operator has
        LD R5, #L1          //no effect; the first
                            //instruction is
                            //equivalent to second
                            //instruction
```

Table 4-1 Segmented Addressing Mode Operators

Operator	Function
^S	Access segment portion of address
^O	Access offset portion of address

4.4. Addressing Mode Directives

Two directives allow the programmer to determine whether the assembly process takes place in segmented or non-segmented mode.

.seg

Directs the assembler to begin assembling in segmented mode. By default, the assembler assembles in non-segmented mode. Any program that contains a .seg directive is assumed to be a segmented program.

.nonseg

Directs the assembler to return to assembling in non-segmented mode.

SECTION 5
PROGRAM STRUCTURE

5.1. Introduction

The structuring of programs and the concept of relocatability are the subject of Section 5.

5.2. Modules

An assembly language program consists of one or more separately-coded and assembled modules (also referred to as files.) These modules are combined into an executable program using the module linkage and relocation facilities of the operating system.

Modules are made up of assembly language statements that define data or perform some action, as described in Section 3.

The assembler produces relocatable object modules. This relocatability feature of the assembler frees the programmer from memory management concerns during program development. Relocatability is supported by several directives, discussed below, that determine where data and action statements are loaded into memory.

5.3. Sections and Areas

In addition to the logical structuring provided by modules, it is possible to divide a program into sections which can be mapped into various areas of memory when the module is linked or loaded for execution. For example, the programmer may choose to group a set of data structures and statements that manipulate them together in the same module. But it may also be desirable to physically separate the object code for the statements from the data in a system where read-only memory is used for the statements and read/write memory is used for the data.

Each section might be allocated to a different address space. In segmented mode, each section might be mapped into a different segment, or several sections from different modules might be combined into the same segment. A single module may contain several sections, each of which will be allocated a different area in memory. Alternatively, the

portions of a single section may be spread through several modules and the portions automatically combined into a single area by the linker.

There is a one-to-one mapping between sections and segments in segmented mode. In non-segmented mode, the capability for such one-to-one mapping does not exist, although sections allow portions of a user's program to be grouped logically as they do segmented mode.

Currently on the System 8000, the code, data, and bss areas of a module can be manipulated separately. If all of the data in one module is contained in one section, it is possible to manipulate that section at link time. The capability to manipulate sections by name, whatever their contents, is not yet implemented.

The assembler allows a program to be divided into up to three types of sections: program section, absolute section and common section. Each section can contain up to three areas: a code area, a data area, and a bss (uninitialized data area). The code and data areas can contain any legal assembler statement, but the bss area can contain uninitialized data only. In addition, the code area is limited to 64K; the the data and bss areas combined cannot exceed 64K.

There are three area assembler directives.

AREA ASSEMBLER DIRECTIVES

.code

Directs the assembler to change to the code area of the current section.

.data

Directs the assembler to change to the data area of the current section.

.bss

Directs the assembler to change to the uninitialized data area of the current section.

```
    .code                   //enter the code area of current
                            //section
```

```
.data                   //enter the data area of current
                        //section

.bss                    //enter the bss area of current
                        //section
```

5.3.1. Program Sections

A program section contains any legal assembly language statement. Each module must have one unnamed program section but can have additional named program sections. A section name consists of a valid identifier. By default, a module is in the data area of the unnamed program section at the beginning of assembly. The assembler directive .psec both indicates the beginning of a program section and allows changing among program sections.

.psec [<section_name>]

Indicates the beginning of a program section or directs the assembler to change to the specified program section, or to the default program section if section is not specified.

Whenever a new program section is entered, the module is in the data area of that section, by default. Upon return to a section, the module is in the the last previously entered area of that section. An example of the use of the .psec directive along with the area directives follows:

```
.psec arithmetic        //enter the data area (default) of
                        //the program section named
                        //arithmetic

count: .word Ø          //declare a word named count
                        //with initial value Ø

.code                   //enter code area of program
                        //section arithmetic

LD RØ, count            //load the count into register Ø

INC RØ                  //increment the count by 1

LD store, RØ            //load the new count into bss
                        //symbol store

.bss                    //enter the bss area
```

```
store: .word          //word value store

.psec                 //return to the unnamed (default)
                      //program section. The program
                      //returns to the last previously entered
                      //area of the default program section
```

5.3.2. Absolute Sections

An absolute section is one whose memory image reflects the absolute location of the section in memory. Since there can be only one absolute section per module, it has no name.

Absolute Section Directive

.asec

Directs the assembler to change to the absolute section of the current module. The module is in the data area of the section, by default. Examples of its use follow:

```
.asec                 //enter the absolute section
.extern proc1, proc2\ //define external symbols
        proc3,proc4

.=1Ø
jumptab::.addr proc1\ //absolute location 1Ø
       proc2, proc3\  //define jump table
       proc4\         //containing addresses
                      //of external routines
```

5.3.3. Common Sections

Common sections allow reference to sections of the same name in several different modules. At link time, such sections are merged into one section using the size of the largest section for the merged one. A common section name consists of a valid identifier.

Common Section Directive

.csec <section_name>

Indicates the beginning of a common section or directs the assembler to change to the specified common section. The

following example shows the use of common sections in three different modules:

Module 1, named file1.s, contains:

```
.csec mycommons
.=.+10
L1::.word %FEFE  //a word at location 10
```

Module 2, named file2.s, contains:

```
.csec mycommons
.=.+20
L2::.word %ABAB  //a word at location 20
```

Module 3, named file3.s, contains:

```
.csec mycommons
.=.+10
L3::.word %CDCD  //a word at location 10
```

When these three files are linked with the command

```
ld -o final file1.o file2.o file3.o
```

the resulting common section is equivalent to a single module that contains the following code:

```
.csec mycommons
L1::L3:: .word %CDCD  //a word at location 10
L2::.word %ABAB        //a word at location 20
```

NOTE

The value at location 10 in the last file takes precedence over the value at location 10 in the first file because of the order in which the files were placed in the linker command line.

5.4. Local Blocks

Local blocks allow further structuring of assembly language programs. Local blocks are enclosed within the the symbols "{" and "}". They can be nested. The symbols "{" and "}" must be the only characters on the line.

Locals labels, described in Section 3, can be used within local blocks only. The scope of any local symbol is the nearest enclosing local block delimiter. For example:

```
{                           //start local block

    ~L1:=2Ø                 //declare local constant
    ld rØ, #~L1             //reference local constant
~L2:ld r2, #%1Ø             //declare local label
    ld rØ, #~L2             //reference the local label

    {                       //start nested local block
          ~L1:=1Ø           //declare local constant
          ld rØ, #~L1       //reference ~L1 equals 1Ø

    }                       //end nested local block

    ld rØ, #~L1             //reference ~L1 equals 2Ø

}                           //end local block
```

5.5. Location Counter

The assembler tracks the location of the current statement with a location counter, just as an executing program does with its program counter. There is a location counter associated with each of the three possible areas of a section: data, code and bss. The counter value represents a 16-bit offset within the current area. The offset can be an absolute value if the area falls within an absolute section or it can be a relocatable value if the area falls within a program or common section. If it is an absolute value, the location counter reflects the absolute memory location of the current statement. If it is relocatable value, the location counter reflects the relocatable offset of the statement. The relocatable offset can be adjusted at link time, depending on where the section is finally allocated.

The location counter symbol "." can be used in any expression, and represents the address of the first byte of the current instruction or directive.

5.5.1. Location Counter Control

Two assembler directives enable control of the location counter:

.even

Increases the location counter by one if it holds an odd address. Has no effect if the location counter holds an even address.

.odd

Increases the location counter by one if it holds an even address. Has no effect if the location counter holds an odd address.

5.5.2. Line Number Directive

An additional directive is provided by the assembler:

.line <number> ['"'(filename)'"']

Sets the current line number to the number specified. An optional filename can be provided to indicate the name of the file (module) being processed. In conjunction with the System 8000's include facility, this directive can be used for error reporting.

APPENDIX A
SUMMARY OF ASSEMBLER DIRECTIVES

A.1. Introduction

This appendix summarizes the assembler directives. The grammar rules that apply to their use are described in Section 3.

.seg

Directs the assembler to begin assembling in segmented mode. By default, the assembler assembles in non-segmented mode. Any program that contains a .seg directive is assumed to be a segmented program.

.nonseg

Directs the assembler to return to assembling in non-segmented mode.

.even

Increases the location counter by one if it holds an odd address. Has no effect if the location counter holds an even address.

.odd

Increases the location counter by one if it holds an even address. Has no effect if the location counter holds an odd address.

.line <number> ['"'(filename)'"']

Sets the current line number to the number specified. An optional filename can be provided to indicate the name of the file (module) being processed. In conjunction with the System 8000's include facility, this directive can be used for error reporting.

.comm <expression> <label>+

Defines a label as a common label.

.extern <label>+

Defines a label as an external label.

.code

Directs the assembler to enter the code area of the currect section.

.data

Directs the assembler to enter the data area of the current section.

.bss

Directs the assembler to enter the uninitialized data area of the current section.

.psec [<section_name>]

Directs the assembler to change to the specified section.

.csec <section_name>

Directs the assembler to change to the specified common section.

.asec

Directs the assembler to change to the the absolute section.

.byte (<number> '('<expression>')'|<expression>| '"'string'"')*

Allocates storage and initializes it with the specified byte value(s) which can be a series of constant and relocatable expressions or an ascii string. Number is the repetition factor. When a number is specified, the expression must be enclosed in parentheses; strings are enclosed in double quotes.

.word (<number> '('<expression>')'|<expression>)*

Allocates storage and initializes it with the specified word value(s) which can be a series of constant and relocatable expressions. Number is the repetition factor. When a number is specified, the expression must be enclosed in parentheses.

.long (<number> '('<expression>')'|<expression>)*

Allocates storage and initializes it with the specified long value(s) which can be a series of constant and relocatable expressions. Number is the repetition factor. When a number is specified, expression must be enclosed in parentheses.

.quad (<number> '('<expression>')'|<expression>)*

Reserves 64 bits of storage. Only double precision floating point numbers fill the allocated storage completely. If the value does not fill the allocated storage completely, no sign extension is performed. Number is the repetition factor. When a number is specified, the expression must be enclosed in parentheses.

.extend (<number> '('<expression>')'|<expression>)*

Reserves 8Ø bits of storage. Only extended precision floating point numbers fill the allocated storage completely. If the value does not fill the allocated storage completely, no sign extension is performed. Number is the repetition factor. When a number is specified, the expression must be enclosed in parentheses.

.addr (<number> '('<expression>')'|<expression>)*

When assembling in non-segmented mode, allocates storage and initializes it with the specified 16-bit value which can be a series of constant and relocatable expressions. Number is a repetition factor. When number is used, expression must be enclosed in parentheses.

When assembling in segmented mode, allocates storage and initializes it with a 32-bit value.

.blkb <expression>

Allocates storage in bytes. The number of bytes is specified by the expression. No initialization occurs.

.blkw <expression>

Allocates storage in words. The number of words is specified by the expression. No initialization occurs.

.blkl <expression>

Allocates storage in long words. The number of long words is specified by the expression. No initialization occurs.

APPENDIX B
KEYWORDS AND SPECIAL CHARACTERS

KEYWORDS

Certain special symbols are reserved for the assembler and can not be redefined as symbols by the programmer. These are the names of condition codes, register symbols, assembly language instructions.

CONDITION CODES

C	LE	NE	PE	UGT
EQ	LT	NOV	PL	ULE
GE	MI	NZ	PO	ULT
GT	NC	OV	UGE	Z

CONTROL REGISTER SYMBOLS

FCW	PSAP
FLAGS	PSAPOFF
NSP	PSAPSEG
NSPOFF	REFRESH
NSPSEG	

Flag Names

S	C
V	P
Z	VI
NVI	

FLOATING POINT KEYWORDS

AFF	FOP1	INV	OUFLG	RP	USER
CMPFLG	FOP2	INX	PCI	RZ	WARN
DBL	FOV	IN	PCZ	SCON	
DE	INTFLG	IX	PROJ	SGL	
DZ	INVFLG	NAN	RM	SYSFLG	
FFLAGS		NORM	RN	TRAPS	

FLOATING POINT CONDITION CODES

FEQ	FGU	FLU
FGE	FLE	FNEU
FGEU	FLEU	FORD
FGT	FLT	FUN

ASSEMBLY LANGUAGE INSTRUCTIONS

ADC	EI	FMULS	LD	POPL	SLLB
ADCB	EX	FNEG	LDA	PUSH	SLLL
ADD	EXB	FNEGD	LDAR	PUSHL	SOTDR
ADDB	EXTS	FNEGS	LDB	RES	SOTDRB
ADDL	EXTSB	FNORM	LDCTL	RESB	SOTIR
AND	EXTSL	FNORMD	LDCTLB	RESFLG	SOTIRB
ANDB	FABS	FNORMS	LDD	RET	SOUT
BIT	FABSD	FNXM	LDDB	RL	SOUTB
BITB	FABSS	FNXMD	LDDR	RLB	SOUTD
CALL	FADD	FNXMS	LDDRB	RLC	SOUTDB
CALR	FADDD	FNXP	LDI	RLCB	SOUTI
CLR	FADDS	FNXPD	LDIB	RLDB	SOUTIB
CLRB	FCLR	FNXPS	LDIR	RR	SRA
COM	FCP	FREM	LDIRB	RRB	SRAB
COMB	FCPD	FRESFLAG	LDK	RRC	SRAL
COMFLG	FCPF	FRESTRAP	LDL	RRCB	SRL
CP	FCPS	FSCL	LDM	RRDB	SRLB
CPB	FCPX	FSETFLAG	LDPS	SBC	SRLL
CPL	FCPXD	FSETMODE	LDR	SBCB	SUB
CPD	FCPXF	FSETTRAP	LDRB	SC	SUBB
CPDB	FCPZ	FSIGQ	LDRL	SDA	SUBL
CPDR	FCPZX	FSQR	MBIT	SDAB	SWAP
CPDRB	FDIV	FSQRD	MREQ	SDAL	TCC
CPI	FDIVD	FSQRS	MRES	SDL	TCCB
CPIB	FDIVS	FSUB	MSET	SDLB	TEST
CPIR	FEXM	FSUBD	MULT	SDLL	TESTB
CPIRB	FEXPL	FSUBS	MULTL	SET	TESTL
CPSD	FINT	HALT	NEG	SETB	TRDB
CPSDB	FINTD	IN	NEGB	SETFLG	TRDRB
CRSDR	FINTS	INB	NOP	SIN	TRIB
CPSDRB	FLD	INC	OR	SINB	TRIRB
CPSI	FLDBCD	INCB	ORB	SIND	TRTDB
CPSIB	FLDCTL	IND	OTDR	SINDB	TRTDRB
CPSIR	FLDCTLB	INDB	OTDRB	SINDR	TRTIB
CPSIRB	FLDD	INDR	OTIR	SINDRB	TRTIRB
DAB	FLDIL	INDRB	OTIRB	SINI	TSET
DBJNZ	FLDIQ	INI	OUT	SINIB	TSETB
DEC	FLDP	INIB	OUTB	SINIR	XOR
DECB	FLDPD	INIR	OUTD	SINIRB	XORB
DI	FLDPS	INIRB	OUTDB	SLA	
DIV	FLDS	IRET	OUTI	SLAB	
DIVL	FMUL	JP	OUTIB	SLAL	
DJNZ	FMULD	JR	POP	SLL	

Pseudo Instructions

JPR CALLR

When defining symbols, users must also avoid the forms:

Rn	where n is a number from Ø to 15
RHn or RLn	where n is a number from Ø to 7
RRn	where n is any of the even numbers from Ø to 14
RQn	where n is any of the numbers Ø, 4, 8, 12
Fn	where n is a number from Ø to 7

SPECIAL CHARACTERS

The list of special characters below includes delimiters and special symbols. The difference between them is that delimiters have no semantic significance (for example, two tokens can have any number of blanks separating them), whereas special symbols do have semantic meaning (for example, # is used to indicate an immediate value).

The class of delimiters includes the space (blank), tab, line feed, carriage return, semicolon (;), and comma (,). The comment construct enclosed in the symbols /* is also considered a delimiter.

The special symbols and their uses are as follows:

+	Binary addition; unary plus
-	Binary subtraction; unary minus
*	Unsigned multiplication
/	Unsigned division
^<	Shift left
^>	Shift right
^$	Bitwise and
^|	Bitwise or
^x	Bitwise xor
:	Internal label terminator
::	Global label terminator

~	Local label indicator
:=	Constant and variable initialization
%	Nondecimal number base specifier
#	Immediate data specifier
@	Indirect address specifier
()	Enclose expressions selectively; enclose octal or binary number base indicator; enclose index part of indexed, based, and based indexed address
.	Location counter indicator
//	Begin comment
<< >>	Denotes segmented address
| |	Enclose short offset segmented address
^S	Access segment portion of address
^O	Access offset portion of address
^B	Binary-coded decimal
^C	Ones complement
^FS	Convert to floating single
^FD	Convert to floating double
^F	Convert to floating extended

APPENDIX C
ASSEMBLER ERROR MESSAGES

Appendix C describes the assembler warning and error messages.

Warnings

Errors that cause warning messages do not interfer with the operation of the assembler, but they should be corrected before the a.out file is executed.

Operand too large
Value too large
A number too large for a data type or instruction field has been used; for example, ".byte %ffff".

Syntax errors

Most syntax error messages are self explanatory; those that are not self-explanatory are listed here.

When there is more than one syntax error per line, only the first is reported. When syntax errors are detected, no a.out file is created. Check the appropriate section of this manual for correct syntax.

Expecting carriage return or linefeed
Extra characters (identifiers, expressions, punctuation etc.) were found at the end of a statement. The most common situations where this can occur are listed below.

A label is followed by something other than a statement beginning or the ":=" operator.

An opcode is followed by something other than an operand, or floating point rounding or infinity mode.

A ".psec" directive is followed by something other than an identifier.

A ".byte," ".word," ".long," or ".addr" is followed by something other than an address expression or an address expression repeat count.

Commas are missing between operands, rounding modes, infinity modes, address expressions in ".byte",

".word", ".long", or ".addr" statements, identifiers in ".comm" and ".extern" statements, or between the expression and the first identifier in a ".comm statement."

Missing binary operators such as "+", "-", "^X" in expressions.

Addressing modes must be punctuated exactly as explained in Section 4 of this manual. In certain cases a missing left parenthesis will cause this error message.

The ".line" directive has an optional string argument for the filename. Anything other than a string after the line number will result in this error.

Expecting beginning of line
The first symbol in the statement is one that cannot legally begin a statement.

Expecting beginning of program
The first symbol in the first line of the program is one that cannot legally begin a statement.

Semantic or Fatal errors

Only the first semantic error for a line is reported. An a.out file may be created but it will be corrupt. Semantic errors are often associated with syntax errors. Correct the syntax errors first.

Block specifier must be constant
Expression used to specify the size of a block of storage for a ".blkb", ".blkl", or ".blkw" statement was relocatable.

Bss cannot be initialized
This indicates that the programmer entered the bss area with a ".bss" directive and placed instructions or initialized data there. The ".bss" area can only contain uninitialized storage as in ".blkb", ".blkl", ".blkw" etc.

Invalid assignment
An attempt was made to associate a symbol name with an external or undefined value in a ":=" direct assignment.

Invalid constant
An expression that should have been constant was found to be relocatable or external.

Invalid operand combination
The operand combination used with a specific instruction was invalid. Refer to the Z8000 CPU Technical Manual or Floating Point Emulator User's Manual entry for the specific instruction to find the valid operand combinations.

Invalid section name
A ".psec", ".csec", or ".asec" directive was used with a name that had been defined as a label elsewhere. Section names can only be defined with section directives.

Invalid token
An invalid token was discovered, such as an improper identifier or floating point number.

Mixed relocatable and absolute
A relative instruction's target was absolute yet the instruction was in a relocatable section or vice versa.

Nesting too deep
Nested blocks denoted by enclosing brackets "{" and "}" exceeded the currently implemented nesting depth.

Out of nodes
The assembler has run out of space for initialization. This usually indicates that too many values were used in a data statement. It is suggested that the statement be broken into several statements.

Register must be 0-7
A byte register was used with a register number other than 0-7. Only the first seven registers of the Z8000 may be used as byte registers.

Symbol redefined
A symbol that was previously defined has been redefined.

Segment overflow
More than 64K bytes of code, data and bss has been placed into a single section.

Too many segments
More than 256 sections (i.e. ".psec", ".csec", or ".asec") have been defined.

Undefined symbol
A symbol has been referenced but no definition has been found. If the -u option of the assembler is used, all such references will be made external without an explicit ".extern" statement.

Unknown keyword
A symbol beginning with "." was used but no keyword by that name was found. Only assembler keywords are allowed to begin with ".".

Both sides of <binary operator> must be constants
Invalid addition (or subtraction) expression
Invalid expression type for ^S (or ^o, or | |) operator
Invalid expression type for left (or right) side of <binary operator>
The rules for relocatable, constant or external expressions may have been violated. Also some operators have additional constraints (for example, "^S", "^O") and the rules for operators may have been violated.

Bad relocation bits
Cannot determine expression type
Erroneous expression type
Nodes allocated at end of statement
Too many bits assembled for word
Unexpected tag in intermediate file
Unexpected tag in symbol file
Unknown area
Unknown expression type
Unknown scope
Unknown tag in intermediate file
Generally these are errors associated with syntax errors, or a previous semantic error. Correct the other errors first, before attempting to fix these. If one of these errors occurs without other errors it may be an assembler internal error.

Fatal errors

These errors cause the assembler to abort immediately.

Invalid option
An unknown command line option was invoked.

Yacc stack overflow
Too many states were used in parsing the grammar.

APPENDIX D
DEBUGGER SUPPORT DIRECTIVES

The following directives are for debugger support and are only produced by compilers. They are not intended for use by assembly language programmers.

.stable <number>
Allocate space for source code line number with associated assembly language code.

.stabn <constant_expr> ',' <constant_expr> ',' <constant_expr> ',' '"' string '"'
Allocate symbol table entry for non-relocatable symbol debug information.

.stabp <constant_expr> ',' <constant_expr> ',' <constant_expr> ',' <constant expr> ',' '"' string '"'
Allocate symbol table entry for parameter debug information.

.stabr <constant_expr> ',' <constant_expr> ',' '"' string '"'
Allocate symbol table entry for relocatable symbol debug information.

THE C PROGRAMMING LANGUAGE

Preface

The System 8000 uses the C programming language extensively. 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. For information on the calling conventions, see System 8000 Calling Conventions (CALL CONV).

Each installation contains machine dependencies that affect the C programming language, despite the universality of the 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 1-1

1.1. Introduction 1-1
1.2. Setret and Longret Routines 1-1
1.3. Impact of Passing Parameters in Registers 1-1
1.4. Object Format Dependencies 1-7
1.5. Byte Order Within Words 1-7
1.6. Machine Architecture Dependencies 1-8
1.7. C Compiler Features 1-8

SECTION 2 RECENT CHANGES TO C 2-1

2.1. General 2-1
2.2. Structure Assignment 2-1
2.3. Structure and Union Members 2-1
2.4. Enumeration Type 2-2
2.5. Void Data Type 2-4

List of Illustrations

Figure
1-1 Example of PDP-11 Program 1-1
1-2 System 8000 Version of Figure 1-1 Program ... 1-2
1-3 S8000 Program: Different Type Arguments 1-6

2-1 Named Structured Fields Within Unions 2-2

SECTION 1
CONVERSION OF PROGRAMS TO ZEUS

1.1. Introduction

Although the standard System III UNIX runs on the System 8000 and the C compiler accepts the C language, users must be aware of machine dependencies that may be present in their programs. This section describes the most common machine dependencies that must be removed when porting programs to the System 8000.

1.2. Setret and Longret Routines

When using the C language routine on the System 8000, 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 makes the program executable on the System 8000.

The System 8000 C compiler's stackframes differ from the PDP-11 UNIX. The System 8000 contains 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 set than the PDP-11 processor. To use these registers efficiently, parameters are passed in registers on the System 8000 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 System 8000. Most programs need only to be recompiled to accommodate this change. In cases when procedures handle 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 accommodate 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 = Ø;
        if (*p == Ø)
                return Ø;
        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, ap1, ap2)
char    *ap1, *ap2;
{
        reg char        *p, *np;
        char            *onp;
        reg int         n;
        p = ap1;
        n = Ø;
        if (*p == Ø)
                return Ø;
        do
        {
                n++;
        } while (*p++);
        if (na > 1)
        {
                p = ap2;
                while (*p++)
                        n++;
        }
        onp = np = alloc(n);
        p = ap1;
        while (*np++ = *p++)
                continue;
        if (na > 1)
        {
                p = ap2;
                np--;
                while (*np++ = *p++)
                        continue;
        }
        return onp;
}
```

Figure 1-2. System 8ØØØ 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("%cØ,'z');
printz("double: %DØ,1L);
printz("decimal: %dØ,69);
printz("%c%c%c%c%c%cØ,'a','b','c','d','e','f','g');
printz("%D %D %D %DØ,1ØØL,123456L,1L,98765432L);
printz("%D %d %c %dØ,32L,1Ø,'x',52);
}
```

Figure 1-3. A System 8ØØØ 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 System 8000. The entire object file produced by the language processors on the System 8000 conform to the System 8000 object code format. Refer to a.out (5) for a complete description of the System 8000 object code format.

1.5. Byte Order Within Words

Byte order on the System 8000 differs from byte order on the PDP-11. On the System 8000, 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 System 8000. Also, transporting files between a System 8000 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 System 8000, 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 System 8000 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 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 reduced 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 for the floating representation of 1.1 are used to initialize d. In converting PDP-11 C programs to System 8000 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 System 8000.

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. Structure and Union Members

Structure and union members are now uniquely identified by the struct or union of which they are a part. It is legal for the same identifier to be used, even with different type and location, in different structures or unions. A simple example is shown in Figure 2-1.

```
/*
**    The following is a simple example
**    of the use of named structured fields
**    within unions or structures.
**    The value of the "all" field should
**    be 00010203 <hex>.
*/

main ()

{
      union
      {
           struct
           {
             int j;
             int k;
           } s1;

           struct
           {
             int p;
             char j;
             char k;
           }s2;

           long all;

      } u,*p;

p = &u;
p->s1.j = 1;
p->s2.j + 2;
p->s2.k + 3;
}
```

Figure 2-1. Sample Code For Named Structured Fields

2.4. 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 enumlist 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 the ZEUS implementation, all enumeration variables are treated as integers.

2.5. Void Data Type

A new data type, void, allows a routine to return nothing. This data type makes it unnecessary to declare such routines as returning an integer. An error message is issued if an attempt is made to use a value from a function returning a void or if such a function tries to return a value.

SYSTEM 8000 CALLING CONVENTIONS

Table of Contents

SECTION 1 SYSTEM 8000 CALLING CONVENTIONS 1-1

1.1. Introduction 1-1
1.2. Register Usage 1-2
1.3. Stack Organization 1-4
1.4. Parameters 1-7
1.4.1. The Parameter Register Assignment Algorithm 1-8
1.4.2. The Algorithm 1-10

APPENDIX A SAMPLE PROGRAM USING CALLING CONVENTIONS .. A-1

List of Illustrations

Figure		
1-1	Z8000 Register Usage	1-2
1-2	Z8000 Floating Point Register Usage	1-4
1-3	Stack Upon Entry To and After Return From a Procedure	1-5
1-4	Stack During Procedure Execution	1-6
1-5	Underlying Registers	1-9
A-1	A Sample C Program	A-1
A-2	Registers Upon Entry To And Return From Called Routine	A-2
A-3	Stack Frame During Execution of Called Routine	A-3
A-4	Assembly Language Code For Program in Figure A-1	A-5

List of Tables

Table
1-1 Definition of Algorithm Elements 1-12

SECTION 1
SYSTEM 8000 CALLING CONVENTIONS

1.1. Introduction

The System 8000 Calling Conventions allow programs written in various System 8000 languages to communicate with each other and to share common libraries. The conventions include argument passing, Stack Pointer status, and register assignments on entry to and exit from a routine. The conventions described here apply to all programming languages supported by the Z8000-based System 8000.

The calling conventions:

- Satisfy the requirements of languages such as C, PLZ/SYS, FORTRAN, and PASCAL.
- Do not introduce undue call and return overhead in code generated by one language processor at the expense of another.
- Minimize the complexity of the code generators.
- Allow passing of structure parameters by value.
- Encourage efficiency by allowing local variables to be kept in registers and parameters to be passed in registers.

The calling conventions have three parts which are described in the following sections. These three parts describe:

- How registers may be used by procedures and what happens to the register contents when calling or returning.
- How the stack must be organized when entering, executing in, and returning from a procedure.
- Where parameters must be when entering or returning from a procedure.

1.2. Register Usage

As shown in Figure 1-1, the Z8000's general-purpose register set is divided into three groups for the purposes of this calling convention.

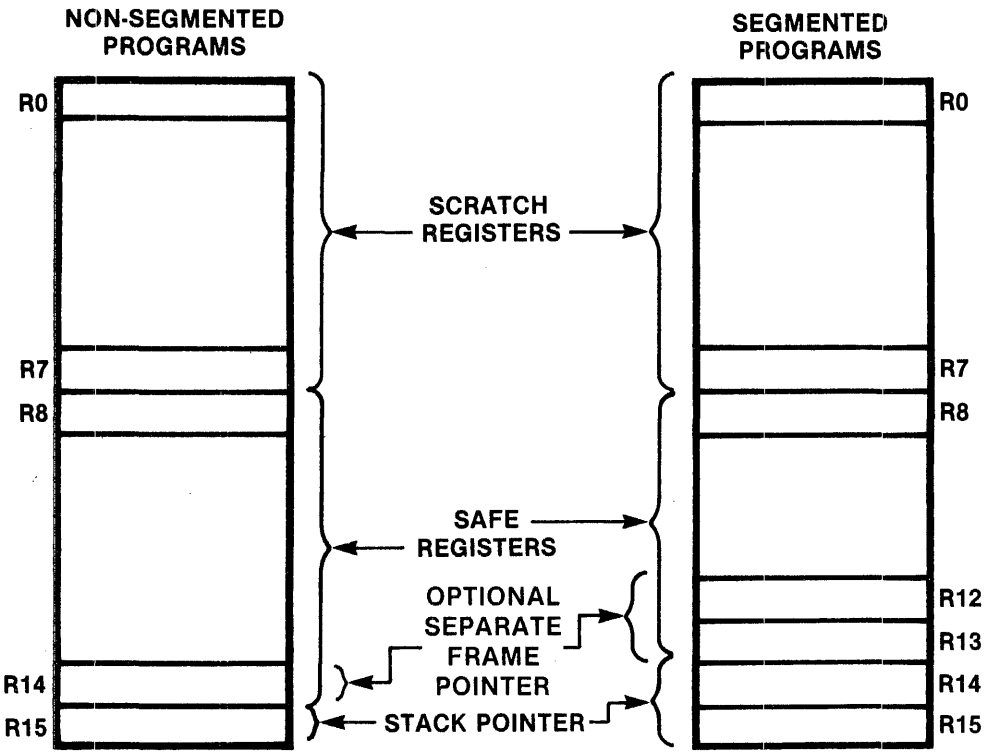


Figure 1-1 Z8000 Register Usage

The first group is called the scratch registers and consists of R2-R7.

NOTE

RØ and R1, although also considered scratch registers, are never used for parameter passing.

These registers contain value or reference parameters when entering a procedure and result parameters when returning

from a procedure. While executing, the procedure may use these registers in any way and does not need to restore them to their original values when it returns.

The second group is called the safe registers and consists of R8-R14 for nonsegmented programs and R8-R13 segmented programs. The value in these registers must be the same when a procedure returns as they were when the procedure was entered. This means a safe register can hold the value of a local variable, because procedure calls do not alter its value. If a procedure changes the value of a safe register, it must save the value of that register when it is entered, and restore it when it returns.

The third group consists of the stack pointer (SP), which is R15 for nonsegmented programs and R14 and R15 for segmented programs. The stack pointer always points to the top of the stack.

The calling convention also allows for, but does not require, the use of a separate frame pointer to point to the current stack frame (described in the next section). When a separate frame pointer is used, it is always the highest safe register, R14 for a nonsegmented program, RR12 for a segmented program.

The Z8000 Floating-Point Registers (either simulated in software by the Z8070 emulation package or provided in hardware by the Z8070 arithmetic processing unit) are similarly divided into two groups as shown in Figure 1-2.

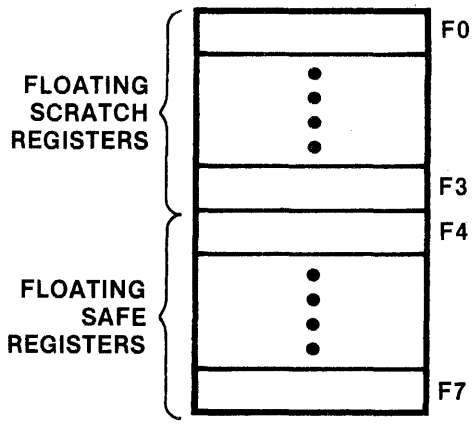


Figure 1-2 Z8000 Floating-Point Register Usage

The first group is the floating scratch registers, FØ-F3. These registers contain floating-point value parameters upon entering a procedure and floating-point result parameters when returning from a procedure. While executing, the procedure can use these registers in any way and does not need to restore them to their original values.

The second group is the floating safe registers, F4-F7. These registers are used in the same way as the general-purpose safe registers and thus the values in these registers must be the same when a procedure returns as they were when the procedure was entered.

1.3. Stack Organization

Figure 1-3 shows how the top of the stack must look when a procedure is entered. The return address must be on the top of the stack (pointed to by the stack pointer), followed by any parameters that must be passed in on the stack. This figure also shows the stack after the same procedure has returned. The only difference is that the return address has been popped off the stack.

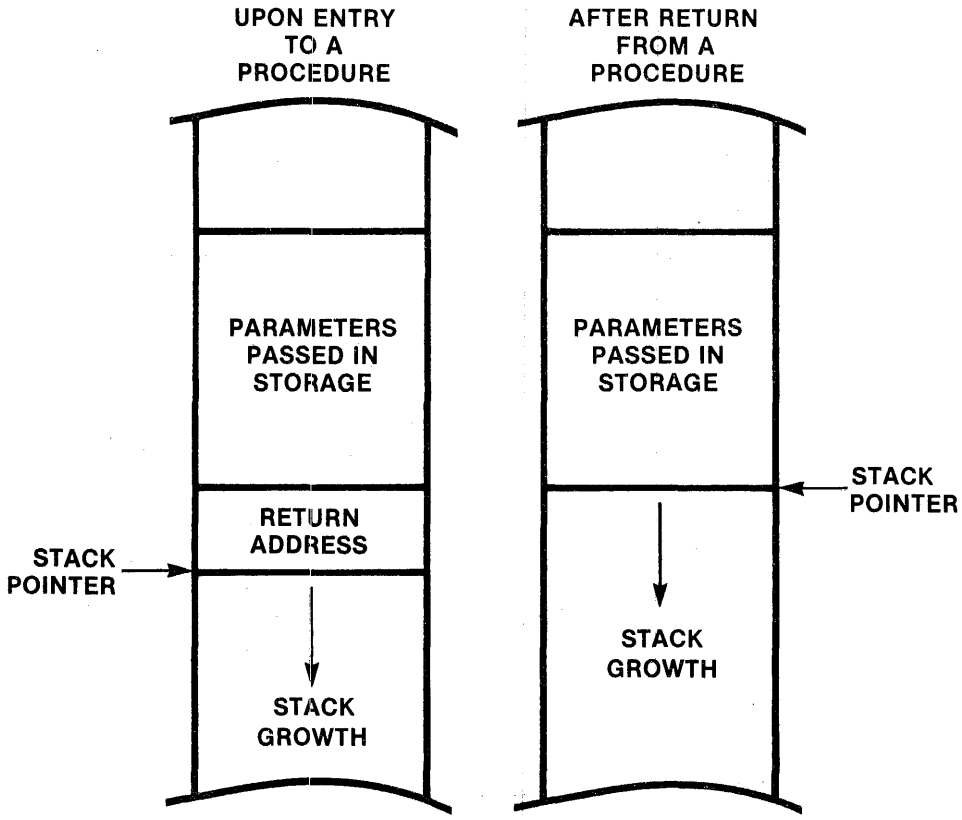


Figure 1-3 Stack Upon Entry to and After Return From a Procedure

During the execution of a procedure, the stack will contain a data area called the stack frame (also known as the activation record) for that procedure. The stack frame is allocated on the stack by the procedure and contains saved values, local variables, and temporary locations for the procedure. Figure 1-4 shows the stack while a procedure is executing. The called procedure may or may not use a separate frame pointer as shown. If no separate frame pointer is used, the size of the stack frame must not change while the procedure is executing. Thus parameters passed in storage by calls from this procedure must be accommodated in temporary locations at the bottom of the stack frame, and not pushed onto the stack. This organization of the stack substantially shortens the subroutine entry and exit sequence.

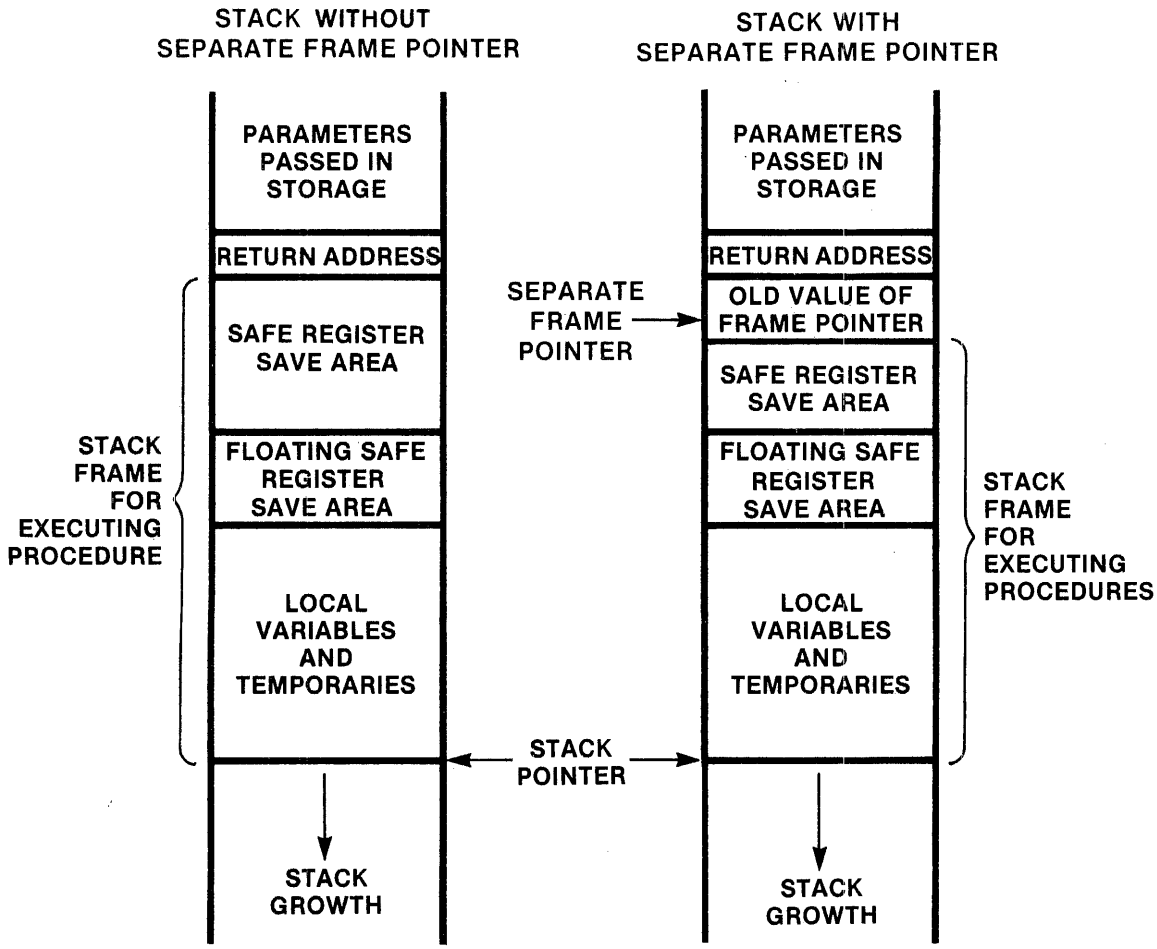


Figure 1-4 The Stack During Procedure Execution

If a separate frame pointer is used, the calling procedure's frame pointer must be saved on the stack by the called routine as shown in Figure 1-4. In this case, the size of the stack frame can vary, and thus parameters can be pushed onto the stack if desired.

The calling convention allows procedures with and without a frame pointer to be mixed on the stack. From this point of view, the frame pointer is just a safe register that is used in an agreed upon way by certain procedures.

If a procedure modifies the contents of any of the safe registers or floating safe registers while it executes, then it must save the values of these registers in its stack from when it is entered so that it can restore them when it returns. The highest safe register not used as a frame

pointer should be saved at the top of the activation record (nearest the return address) with lower number registers saved at lower addresses. This is the same order used by the LDM instruction. Only those safe registers actually modified by the procedure need to be saved.

Any floating safe registers that are modified by the procedure are saved in the activation record just below the last general purpose safe register. Higher numbered floating registers are saved toward the top of the activation record.

1.4. Parameters

Parameters provide a substitution mechanism that permits a procedure's activity to be repeated, varying its arguments. Parameters are referred to as either formal or actual. Formal parameters are the names that appear in the definition of a procedure. Actual parameters are the values that are substituted for the corresponding formal parameters when the procedure is called.

The System 8000 parameter-passing conventions cover three kinds of parameters: value, reference, and result. Value and reference parameters are passed from the calling routine to the called routine. For value parameters, the value of the actual parameter is passed. For reference parameters, the address of the actual parameter is passed. For result parameters, the value of the formal parameter in the called routine is passed to the corresponding actual parameter of the calling routine when the called routine returns.

Each kind of parameter has a length given in bytes (denoted as length(p) for a parameter p). For value and result parameters, this is the length of the declared formal parameter as determined by its type. In the absence of formal parameters, the length of the actual parameter is used. For reference parameters, the length is the length of an address, in other words, two bytes in nonsegmented mode and four bytes in segmented mode.

In addition to a parameter's length, the calling convention distinguishes between parameters of floating-point type and parameters of all other types.

The kind, type and length of a parameter are determined by the conventions of the language in which the calling and the called procedures are written. The user must ensure that these conventions match when making interlanguage calls.

1.4.1. The Parameter Register Assignment Algorithm: This section describes an algorithm that assigns every parameter in a parameter list to either a general-purpose register, floating register, or storage offset. The parameter assigned to storage offset is passed in a storage location whose address is the given offset from the Stack Pointer on entry to the called routine. The algorithm assigns as many parameters to general-purpose registers r2-r7 and floating-point registers fØ-f3 as possible.

The algorithm makes the following assumption:

There are four kinds of general-purpose registers:

- Byte (denoted as rln, rhn, n = Ø...15)
- Word (denoted as rn, n = Ø....15)
- Long Word (denoted as rrn, n = Ø 2, 4, 6, 8, 1Ø, 12, 14)
- Quad Word (denoted as rqn, n = Ø, 4, 8, 12)
- The length of a general-purpose register r [(denoted length(r)] is 1 for a byte register, 2 for a word register, 4 for a long word register, and 8 for a quad word register.
- Each general-purpose register has a set of underlying byte registers as follows:

  The underlying register of a byte register is the register itself.

  The underlying registers of a word register (rn) are the byte registers rln and rhn.

  The underlying registers of a long word register (rrn) are rln, rhn, rln+1, and rhn+1.

  The underlying registers of a quad word register (rqn) are rln, rhn, rln+1, rhn+1, rln+2, rhn+2, rln+3, and rhn+3.

This is illustrated in Figure 1-5.

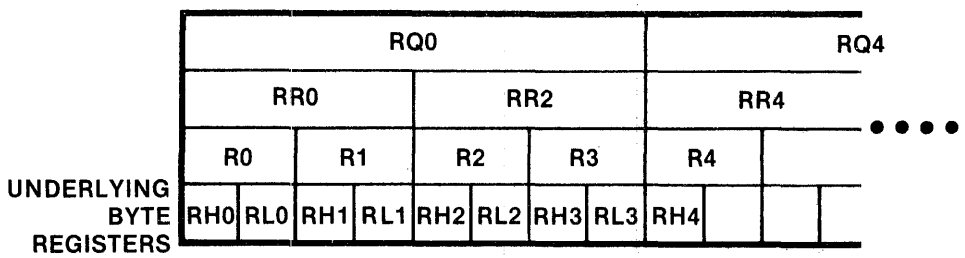


Figure 1-5 The Underlying Registers

- If n > m, general-purpose register rxn or rn is higher than a general-purpose register rxm or rm. A byte register rln is higher than a byte register rhn.

- There are eight floating-point registers, fØ-f7, each capable of holding one floating point value of any precision.

- A floating register fn is higher than a floating register fm if n > m.

The algorithm starts by processing each value or reference parameter in the call in left-to-right order. If there are available registers of the same size and type as the parameter, the parameter is assigned to the highest of these registers; otherwise, it is assigned to the next available storage location. Once a parameter is assigned to storage, all the parameters in the parameter list that follow it are also assigned to storage. The same thing is then done for the result parameters, except they are assigned to the lowest available registers in sequence r2, r3, r4, ... r7 (or fØ, fl, f2, f3), whereas the other parameters are assigned to the registers in sequence r7, r6, r5, ... r2 (or f3, f2, fl, fØ). The result parameters can overlap value or reference parameters in registers, but not in storage.

The algorithm marks byte registers and floating point registers as available or unavailable to keep track of which

registers have been assigned to parameters, and it uses a variable, current offset, to indicate which storage offsets have been assigned parameters.

1.4.2. The Algorithm: This algorithm assigns parameters to registers and storage. The phrases in bold are defined in detail in Table 1-1.

1. Mark all byte registers underlying r2-r7 as available, and mark all other byte registers as unavailable. Mark floating-point registers f0-f3 as available and mark all other floating-point registers unavailable.

2. Initialize current offset to 4 if in segmented mode or to 2 if in nonsegmented mode (this allows for the return address to which the stack pointer points).

3. For every value or reference parameter in left-to-right order in the parameter list, do the following:

   a. **Determine whether p will fit into a register.**

   b. If p will fit into a register, **assign p to a value/reference register and mark the underlying byte registers as unavailable.**

   c. If p will not fit into a register, **assign p to storage** and mark all available byte and floating-point registers as unavailable.

4. Mark all byte registers underlying r2-r7 as available and all other byte registers as unavailable. Mark floating-point registers f0-f3 as available and all other floating-point registers as unavailable.

5. For every result parameter in left-to-right order in the parameter list, do the following:

   a. **Determine whether p will fit into a register.**

   b. If p will fit into a register, **assign p to a result register.**

   c. If p will not fit into a register, **assign p to storage** and mark all available byte and floating-point registers as unavailable.

Table 1-1. Definition of Algorithm Elements

1. **Determine whether p will fit into a register:**

   If p is a floating-point value or result parameter, then p will fit into a register if there is a floating-point register which is available. Otherwise, p will fit into a register if there is a register r such that length(p) = length(r) and all byte registers underlying r are available.

   NOTE

   C structure parameters greater than four bytes will not fit in a register.

2. **Assign p to a value/reference register:**

   If parameter p is a floating-point value parameter then:

   a. Assign p to the highest available floating-point register r.

   b. Mark floating-point register r as unavailable.

   Otherwise:

   a. Find the highest general-purpose register r such that length (p) = length(r) and all byte registers underlying r are available.

   b. Assign parameter p to register r.

   c. Mark all byte registers underlying r as unavailable, and mark any higher available byte registers as unavailable.

3. **Assign p to a result register:**

   If parameter p is a floating-point result parameter then:

   a. Assign p to the lowest available floating-point register r.

   b. Mark floating-point register r as unavailable.

Otherwise,

a. Find the lowest general-purpose register r such that length(p) = length(r) and all byte registers underlying r are available.

b. Assign parameter p to register r.

c. Mark all byte registers underlying r as unavailable, and mark any lower available byte register as unavailable.

4. **Assign p to storage:**

a. If length(p) > 1 and current offset is odd, then add 1 to current offset.

b. Assign parameter p to storage at offset current offset.

c. Add length(p) to current offset.

APPENDIX A
SAMPLE PROGRAM USING CALLING CONVENTIONS

This appendix gives an example that uses the System 8000 calling conventions for a C language routine, "caller", which calls another routine, "called".

Figure A-1 shows the C code, and Figure A-4 shows the corresponding assembly language code. Figure A-2 shows the registers upon entry to "called" and after returning from routine "called". Figure A-3 shows how the stack looks during execution of "called".

```
long
/*
**      Called routine - returns long
*/
called (a, b, c, d, e)
long    b, c;
int     a, d, e;

{
        long    y;

        return y;
}

/*
**      Calling routine
*/
caller()
{
        long    a2, a3, x;
        int     a1, a4, a5;

        x = called(a1, a2, a3, a4, a5);
}
```

Figure A-1 A Sample C Program

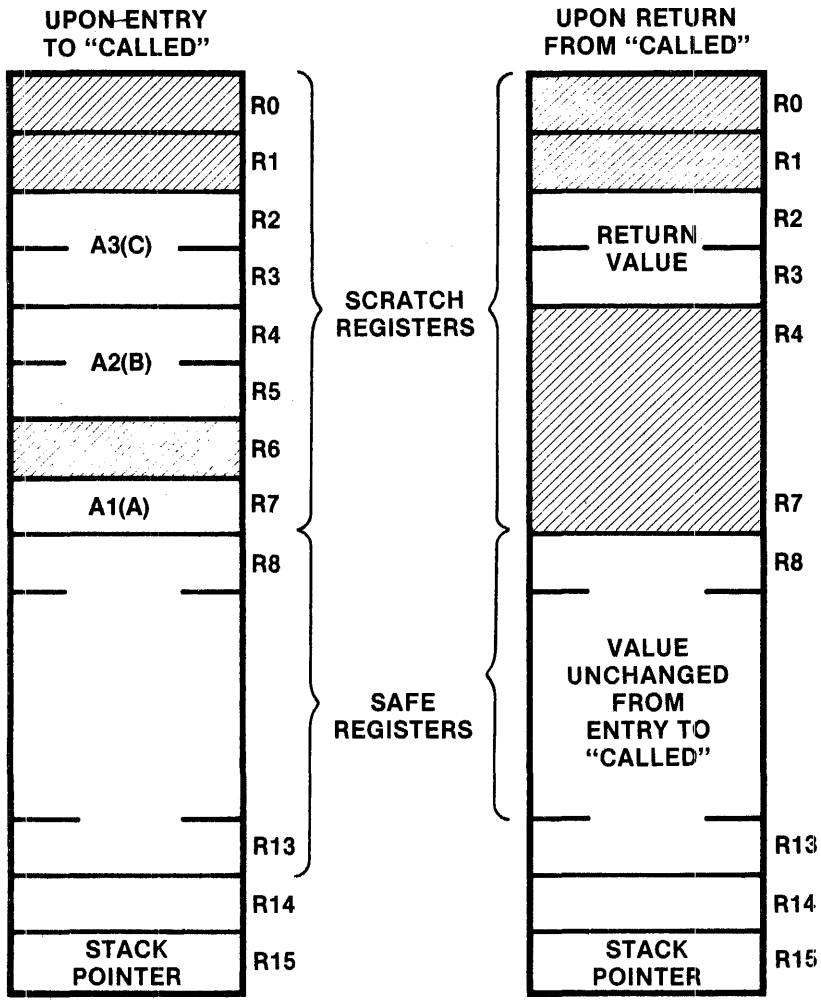


Figure A-2 Registers Upon Entry To and Return From Routine Called

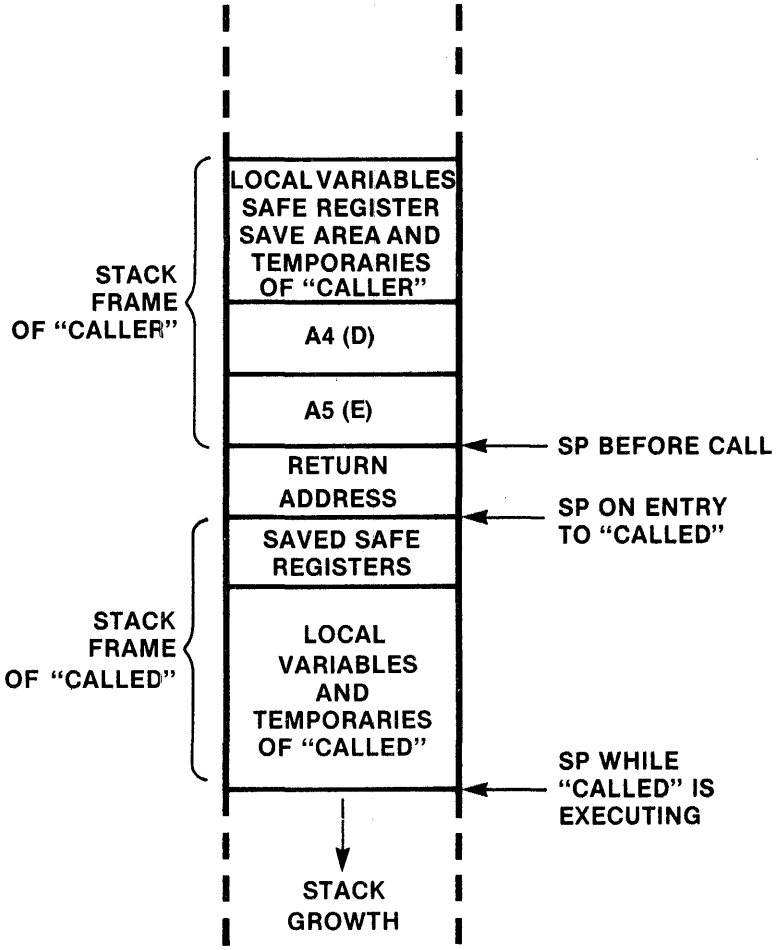


Figure A-3 The Stack Frame When the Routine Called is Executing

```
            fp      := r15;
            sp      := r15;

            .code

             called::
            {

            jpr     L10001
    L10002:
            ldl     rr2,~L1+12(fp)
            jpr     L11
    L11:
    L10000:
            add     fp,#~L2
            ret
    L10001:
            sub     fp,#~L2
            ldm     @fp,r2,#6
            jpr     L10002

            ~L1 := 0
            ~L2 := 20

            }       /* _called */

            .data


            .code

             caller::
            {

            jpr     L10004
    L10005:
            ld      r2,~L1+14(fp)
            ld      @fp,r2
            ld      r3,~L1+16(fp)
            ld      2(fp),r3
            ld      r7,~L1+12(fp)
            ldl     rr4,~L1(fp)
            ldl     rr2,~L1+4(fp)
            callr    called
            ldl     ~L1+8(fp),rr2
    L13:
    L10003:
            add     fp,#~L2
            ret
    L10004:
```

```
        sub     fp,#~L2
        jpr     L10005

        ~L1 := 4
        ~L2 := 22

        }       /* _caller */

        .data
```

Figure A-4. Actual Z8001 Assembly Language Code for Program in Figure A-1.

Indexed Sequential Access Method (C-ISAM)

Table of Contents

SECTION 1 OVERVIEW 1-1

1.1. Introduction 1-1
1.2. Indexed File Systems 1-3
1.2.1. Data (.dat) File 1-3
1.2.2. Index (.idx) File 1-4

SECTION 2 FILE CREATION AND INDEX DEFINITION 2-1

2.1. Introduction 2-1
2.2. C-ISAM File Creation 2-1
2.3. Index Definition 2-1
2.3.1. Keydesc Structure 2-2
2.4. Building A C-ISAM File 2-3
2.5. Adding Secondary Indexes 2-5
2.6. Adding Data 2-6
2.6.1. Reading and Locating Records 2-7
2.6.2. Updating A File 2-8
2.7. Sequential Access 2-12
2.8. Random Access 2-14
2.9. Chaining 2-16

SECTION 3 INDEX COMPRESSION AND CHECKING 3-1

3.1. Introduction 3-1
3.2. Index Compression 3-1
3.3. Index Checking 3-4

SECTION 4 FILE AND RECORD LOCKING 4-1

4.1. Introduction 4-1
4.2. File Locking 4-1
4.2.1. Exclusive File Locking 4-1
4.2.2. Manual File Locking 4-1
4.3. Record Locking 4-2
4.3.1. Automatic Record Locking 4-3
4.3.2. Manual Record Locking 4-3

APPENDIX A C-ISAM CALLS IN SUMMARY A-1

APPENDIX B ERROR MESSAGES AND STATUS BYTES B-1

APPENDIX C DATA TYPES C-1

APPENDIX D HEADER FILES D-1

D.1. HEADER FILE D-1

APPENDIX E FILE FORMATS E-1

List of Illustrations

Figure

2-1	Keydesc and Keypart Structures	2-1
2-2	Program To Create C-ISAM File	2-4
2-3	Program To Add Secondary Indexes	2-6
2-4	Program To Add Data	2-9
2-5	Program To Access File Sequentially	2-12
2-6	Program To Access File Randomly	2-15
2-7	Program To Interactively Add Record	2-18
3-1	Leading Character Compression	3-2
3-2	Leading and Trailing Character Compression	3-2
3-3	Combined Compression	3-3

List of Tables

Table		
1-1	Functional Summary of C-ISAM Functions	1-2
2-1	Mode Parameter for Isread/Isstart Functions .	2-7
B-1	C-ISAM Error Codes	B-1
B-2	Status Byte One	B-3
B-3	Status Byte Two	B-3

SECTION 1
OVERVIEW

1.1. Introduction

The C-Indexed Sequential Access Method (C-ISAM) is a library of C language functions that create and manipulate indexed file systems. The C-ISAM library, /usr/lib/libcisam.a (non-segmented) or /usr/slibcisam.a (segmented), is available to the loader when the C compiler, cc, is invoked with the -lcisam option. When linked with a user-written C language program (or any program written in a language with access to C libraries), the C-ISAM library enables programmers to:

* Create an indexed file system

* Define primary and secondary keys (indexes)

* Add and delete indexes

* Add and delete data records

* Sequentially or randomly access records

* Lock individual records, groups of records, or whole file systems

* Rename and erase indexed file systems

* Compress index files to optimize disk access and save space

This manual describes the use of these functions (summarized in Table 1-1) in building indexed file systems. The individual functions are described in the ZEUS Reference Manual, in keeping with ZEUS documentation conventions. A summary of the function calls can be found in Appendix A.

Table 1-1. Functional Summary of C-ISAM Functions

Group	FUNCTION	DESCRIPTION
Unopen File Operations	isbuild	create and open a file
	isopen	open an existing file
	isrename	rename a file
	iserase	erase a file
Open File Operations	isclose	close an open file
	isaddindex	add a secondary index
	isdelindex	delete a secondary index
	isstart	reset the current key and the current record
	islock	read-lock the file
	isunlock	unlock the file
	isindexinfo	get index/directory information about the file
	isuniqueid	define a unique key for the file
	isaudit	perform audit trail functions
Record operations	isread	read a record in sequential or random mode
	iswrite	insert a record into a file
	isrewrite (isrewcurr)	rewrite a record
	isdelete	delete a record
Misc	isperror	print C-ISAM error
	isld	load value from byte string
	isst	store value in byte string

1.2. Indexed File Systems

An indexed file system is composed of two types of files:

Data File
Index File(s)

1.2.1. Data (.dat) File:

ZEUS imposes no structure on a file; a file is treated simply as a string of bytes. In contrast, C-ISAM allows a structure to be imposed upon a data file, making information access easier and quicker. This structure allows a data file to be treated as a collection of records, and a record to be treated as a collection of fields, with one or more fields within a record defined as the primary key. The primary key serves to identify the record and as an index to the file.

For example, consider an employee file that has one record for each employee. Such a file might have the employee identification number defined as the primary key. One or more secondary keys can be defined for a file providing an alternate index to the file. With the employee file, a secondary key could be defined for the employee's last name.

All C-ISAM data filenames (10 characters maximum) are appended automatically with the suffix .dat when they are created.

Records

A record is a logical unit of information, composed of one or more fields. A typical example is an employee record within a company employee file that contains one such record for each employee.

Fields

A field is a logical unit of information within a record. For example, an employee record could contain several fields including employee id number, name, salary, department number and so on.

C-ISAM recognizes fields with the following data types (described in detail in Appendix C):

Fixed-Length Character Strings (Ø-255 bytes)
Integers
Long Integers
Floating Point
Double Floating Point

A field can start at any offset within a record, allowing data to be packed within a record.

Primary Key

Every C-ISAM file must have a primary key by which the records of the file are indexed, and hence, accessed. A key can comprise one to eight fields. By default, a primary key must uniquely identify the records of a file. Otherwise, it must be defined as allowing duplicates.

For example, an employee's last name could be defined as the primary key for the employee file. But such a key would not index each record uniquely since more than one employee could have the same last name. Such a primary key must be defined as allowing duplicates. Furthermore, it could be defined to comprise three fields: an employee's first, middle and last names.

A special C-ISAM function, isuniqueid, supplies a primary key for a file when a natural one does not exist.

Secondary Keys

In addition to the indexing provided by the primary key, any number of secondary keys can be defined for a file.

1.2.2. Index (.idx) File:

Every C-ISAM data file has an associated index file created when the the C-ISAM file is built. The index filename is the C-ISAM filename appended with the .idx suffix. The index file holds a dictionary describing the file's primary and secondary keys.

Since there is no limit to the number of keys that can be defined for a file, the index file can grow quickly. This consumes disk space and degrades performance. C-ISAM has the capacity, however, to compress the key values held in the index file. In addition to space savings, compression can improve performance. Index compression is the subject of Section 3.

SECTION 2
FILE CREATION AND INDEX DEFINITION

2.1. Introduction

This section describes, through the use of several sample programs, the creation and manipulation of C-ISAM files, including index definition, and the addition of indexes and data.

2.2. C-ISAM File Creation

The C-ISAM function isbuild(3) defines and creates a C-ISAM file. As a result of a call to this function two files are actually created: a data file with the suffix .dat appended to the filename parameter and an index file with the suffix .idx appended to the filename parameter. The .dat file contains data only; the .idx file holds a dictionary that decribes the file's indexes, and the indexes themselves.

2.3. Index Definition

Every C-ISAM file must have a primary key; secondary keys can also be defined for a file either at the time the file is created or at a later date. The keydesc and keypart structures, shown in Figure 2-1, define a file's indexes. These structures are used by the isbuild and isaddindex functions.

```
struct keydesc
    {
    int  k_flags;          /* flags                          */
    int  k_nparts;         /* number of parts in key         */
    struct keypart
        k_part[NPARTS];    /* each key part                  */
    };

struct keypart
    {
    int kp_start;          /* starting byte of key part      */
    int kp_leng;           /* length in bytes                */
    int kp_type;           /* type of key part               */
    };
```

Figure 2-1. Keydesc and Keypart Structures For Index Definition

2.3.1. Keydesc Structure:

In the keydesc structure, the integer k_flags holds compression information and indicates if duplicate key values are allowed. This integer is the arithmetical sum of the values of the following key descriptors:

```
ISNODUPS   - No duplicates (default)
ISDUPS     - Duplicates
DCOMPRESS  - Duplicate Compression
LCOMPRESS  - Leading byte compression
TCOMPRESS  - Trailing byte compression
COMPRESS   - Complete compression (all three of the above)
```

Index compression is described in the next section.

Integer k_nparts indicates how many parts (fields) make up the key. Each part must be described by a keypart structure. The number of elements in the k_part array should be equal to the integer value in k_nparts.

Keypart Structure

The keypart structure allows a key to be composed of multiple fields, referred to as parts. A key can have as many as eight parts. The parts of an index need not be contiguous within record, nor do they have to exist in any particular order within the record. kp_start indicates the starting byte of the key part, defined as the byte offset from the beginning of the record. kp_length is a count of the number of bytes in the part, and kp_type designates the data type of the part. The types allowed by C-ISAM are described in Appendix C. If this part of the key is in descending order, the type macro should be arithmetically added to the ISDESC macro.

The following examples, based on a mythical personnel system, illustrate file creation and index definition. The personnel system consists of two C-ISAM files, the "employee" file, and the "performance" file. The employee file holds a record for each employee consisting of:

```
employee number
name
address
```

The performance file holds information pertaining to all job performance reviews for each employee. There is one record for each performance review, job title change, or salary change an employee has had. Thus, for every employee record in the personnel file there may be many records in the performance file. The field definitions for the records in both the personnel file and the performance file are shown below.

Employee File Definition

Field Name	Location in Record
Employee number	0 - 3 LONGTYPE
Last name	4 - 23 CHARTYPE
First Name	24 - 43 CHARTYPE
Address	44 - 63 CHARTYPE
City	64 - 83 CHARTYPE

Performance File Definition

Field Name	Location in Record
Employee number	0 - 3 LONGTYPE
Review date	4 - 9 CHARTYPE
Job rating	10 - 11 CHARTYPE
Salary after review	12 - 19 DOUBLETYPE
Title after review	20 - 50 CHARTYPE

2.4. Building A C-ISAM File

Figure 2-2 shows a sample program that creates both the employee and performance files, using the isbuild function. For the employee file, the primary key is defined as the employee ID number. For the performance file, the primary key is a two-part key consisting of the employee ID number and the review date.

```
#include <isam.h>

struct keydesc key;
int fdemploy, fdperform;

/*
 * This program builds the C-ISAM file systems for the
 * data files employees and performance.
 */

main()

{
   mkemplkey();
   fdemploy = isbuild("employee", 84, &key,ISINOUT + ISEXCLLOCK);
   if (fdemploy < 0)
           {
           printf("isbuild error %d for employee file\n", iserrno);
           exit(1);
           }


   mkperfkey();
   fdperform = isbuild("perform", 49, &key,ISINOUT + ISEXCLLOCK);
   if (fdemploy < 0)
          {
          printf("isbuild error %d for performance file\n", iserrno);
          exit(1);
          }
   isclose(fdperform);
}


mkemplkey()
{
   key.k_flags = 0;                /* no dups, no compression */
   key.k_nparts = 1;               /* one part index          */

   key.k_part[0].kp_start = 0;     /* offset is zero */
   key.k_part[0].kp_type = LONGTYPE;       /* type long */
   key.k_part[0].kp_leng = 4;      /* 4 bytes */
}


mkperfkey()
{
   key.k_flags = 0;          /* no dups, no compression*/
   key.k_nparts = 2;

   key.k_part[0].kp_start = 0;     /* offset is zero */
   key.k_part[0].kp_type = LONGTYPE;       /* type long */


   key.k_part[1].kp_start = 4;     /* offset is four */
   key.k_part[1].kp_type = CHARTYPE;       /* type char */
   key.k_part[1].kp_leng = 6;      /* 6 bytes */

}
```

Figure 2-2. Program To Build Files and Create Indexes

2.5. Adding Secondary Indexes

With some applications, a primary key is not sufficient to fully index a file. In such cases, one or more secondary indexes can be defined. There is no limit to the number of such indexes; in practice, however, space and access time must be considered. In the case of the sample employee file system, two secondary indexes are desirable -- an index on "last name" in the employee file, and an index on the field "salary" in the performance file. The following program (Figure 2-3) creates these two indexes. It is important to note that while adding indexes the file must be opened with an exclusive lock. Exclusive file locks are specified in the mode parameter of the isopen call by initializing that parameter to ISINOUT + ISEXCLLOCK. The ISINOUT specifies that the file is to be opened for both input and output, and the ISEXCLLOCK macro added to ISINOUT indicates that the file is to be exclusively locked for the current process and that no other process will be allowed to access this file. Note also that duplicates are to be allowed for both secondary indexes and that the name field is to have full compression for its values stored in the index file.

```
#include <isam.h>

#define SUCCESS 0

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;

/*
 * This program adds secondary indexes for the
 * last name field in the employee file, and the
 * salary field in the performance file.
 */

main()

{
int cc;

   fdemploy = cc = isopen("employee", ISINOUT + ISEXCLLOCK);
   if (cc < SUCCESS)

      printf("isopen error %d for employee file\n", iserrno);
      exit(1);
      }

      mklnamekey();
      cc = isaddindex(fdemploy, &key);
      if (cc != SUCCESS)
        {
        printf("isaddindex error %d for employee lname key\n",iserrno);
        exit(1);
        }
      isclose(fdemploy);

   fdperform = cc = isopen("perform", ISINOUT + ISEXCLLOCK);
   if (cc < SUCCESS)
      {
      printf("isopen error %d for perform\n", iserrno);
      exit(1);
      }
```

```
    mksalkey();
    cc = isaddindex(fdperform, &key);
    if (cc != SUCCESS)
       {
       printf("isaddindex error %d for perform\n", iserrno);
       isclose(fdperform);
       exit(1);
       }
    isclose(fdperform);
}
mklnamekey()
{
    key.k_flags = ISDUPS + COMPRESS;
    key.k_nparts = 0;
    cstart = 4;
    nparts = 0;

    addpart(&key, 20, CHARTYPE);
}

mksalkey()
{
    key.k_flags = ISDUPS;
    key.k_nparts = 0;
    cstart = 12;
    nparts = 0;

    addpart(&key, sizeof(double), DOUBLETYPE);
}

addpart (keyp, len, type)

register struct keydesc *keyp;
int len;
int type;
{
    keyp->k_part[nparts].kp_start = cstart;
    keyp->k_part[nparts].kp_leng   = len;
    keyp->k_part[nparts].kp_type   = type;
    keyp->k_nparts = ++nparts;
    cstart += len;

}
```

Figure 2-3. Program to Add Secondary Indexes

2.6. Adding Data

When a file is opened with the isopen function, the type of operation that is to take place and the type of locking desired must be specified. These are the function's mode parameter.

The three types of operations are:

```
ISINPUT  - Read requests only
ISOUTPUT - Write requests only
ISINOUT  -       Read and write requests
```

Available locking options are discussed in the next section.

2.6.1. Reading and Locating Records:

Records are accessed with the isread function or the isstart function. isread reads the record into the buffer, whereas isstart only locates the record, but does not return it.

Both of these functions use a mode parameter, defined in Table 2-1.

Table 2-1. Mode Parameter For Isread and Isstart Functions

Mode	Description
ISFIRST	Locate the first record
ISLAST	Locate the last record
ISNEXT	Locate the next record
ISPREV	Locate the previous record
ISCURR	Locate the current record
ISEQUAL	Locate the record with key value equal to the specified value
ISGREAT	Locate the record with key value greater than the specified value
ISGTEQ	Locate the record with key value greater or equal to the specified value

When ISEQUAL, ISGREAT or ISGTEQ is specified, the call searches for a record according to the value specified by the user. With isread, it must be the current key. In the case of isstart any key may be specified in the key descriptor parameter. It is the user's responsibility to place the search value into the record buffer, at the location the value is located in the record.

For example, if the primary key is a 3 byte character string starting at offset 2 within the record, and the first record to be accessed has the primary key value "ABC", the string "ABC" must be located at offset 2 within the record buffer.

With isstart, partial key searches can be used. For example, to retrieve the first record with key value starting with "A", put a single "A" at offset 2 with a specified

length of 1. This allows retrieving record "AAA" before record "ABC".

If isread is used, and if manual locking is specified when the file is opened, the record can be locked by adding the ISLOCK value to the mode. Refer to the next section.

2.6.2. Updating a File:

Inserting a record in a data file is accomplished with the iswrite function. When the record is inserted, the indexes for each of the keys (primary and secondaries) are updated. An error message is issued if an attempt is made to insert a record with a duplicate key value when the file does not allow duplicate values.

When a record is rewritten (with the isrewrite or isrewcurr functions) the existing record is replaced by the new one. The value of the primary key cannot be changed during this operation.

To change the value of a primary key, insert the record with the new key value and delete the record with the old key value.

The commands to update and delete records have two forms. If the file has a unique primay key, use the isrewrite or isdelete functions to add or delete a record. If the file does not have a unique primary, locate the record using the isread or isstart function and update it using the isrewcurr or isdelcurr functions.

Figure 2-4 is a sample program that adds records to the "employee" file by prompting standard input for values of the fields in the data record. Note that the "employee" file is opened with the ISOUTPUT flag as its mode parameter.

```
#include <isam.h>
#include <stdio.h>

#define WHOKEKEY 0
#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char emprec[85];
char perfrec[51];
char line[82];
long empnum;

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;
int finished = FALSE;

/*
 * This program adds a new employee record to the employee file.
 * It also adds that employee's first employee performance record
 * to the performance file.
 /

main()

{
int cc;

   fdemploy = cc = isopen("employee", ISMANULOCK+ISOUTPUT);
   if (cc < SUCCESS)
      {
      printf("isopen error %d for employee file\n", iserrno);
      exit(1);
      }
   fdperform = cc = isopen("perform", ISMANULOCK+ISOUTPUT);
   if (cc < SUCCESS)
      {
      printf("isopen error %d for performance file \n", iserrno);
      exit(1);
      }
   getemployee();
   while(!finished)
      {
      addemployee();
      getemployee();
      }
   isclose(fdemploy);
   isclose(fdperform);
}

getperform()

{
double new_salary;

   if (empnum == 0)
      {
      finished = TRUE;
      return(0);
      }
   stlong(empnum, perfrec);
```

```
   printf("Start Date: ");
   getline(line, 80);
   stchar(line, perfrec+4, 6);

   stchar("g", perfrec+10, 1);

   printf("Starting salary: ");
   getline(line, 80);
   sscanf(line, "%lf", &new_salary);
   stdbl(new_salary, perfrec+11);

   printf("Title : ");
   getline(line, 80);
   stchar(line, perfrec+19, 30);

   printf("\n\n\n");
}

addemployee()

{
int cc;
   cc = iswrite(fdemploy, emprec);
   if (cc != SUCCESS)
      {
      printf("iswrite error %d for employee\n", iserrno);
      isclose(fdemploy);
      exit(1);
      }
}

addperform()

{
int cc;
  cc = iswrite(fdperform, perfrec);
  if (cc != SUCCESS)
      {
      printf("iswrite error %d for performance\n", iserrno);
      isclose(fdperform);
      exit(1);
      }
}

putnc(c, n)

char *c;
int n;
{
   while(n--) putchar(*(c++));
}




getemployee()

{
   printf("Employee number (enter 0 to exit): ");
   getline(line, 80);
   sscanf(line, "%ld", &empnum);
   if (empnum == 0)
      {
      finished = TRUE;
      return(0);
      }
   stlong(empnum, emprec);
```

```
   printf("Last name: ");
   getline(line, 80);
   stchar(line, emprec+4, 20);

   printf("First name: ");
   getline(line, 80);
   stchar(line, emprec+24, 20);

   printf("Address: ");
   getline(line, 80);
   stchar(line, emprec+44, 20);

   printf("City: ");
   getline(line, 80);
   stchar(line, emprec+64, 20);

   getperform();
   addperform();

   printf("\n\n\n");
}

getline(s, lim)
char s[];
int lim;
{
int c, i;
          {
          for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
 s[i] =c;
 if (c=='\n')
           {
     s[i] =c;
     ++i;
   }
   s[i] = '\0';
   return(i);
}
stchar(a,b,c)
char *a, *b;
int c;
{
   register int i;

   for (i=0;*a && (i < c); i++)
         *b++ = *a++;
   return(0);
}
```

Figure 2-4. Program to Add Data

2.7. Sequential Access

Figure 2-5 shows how to read a file sequentially. In this case, the "employee" file is being read in order of the primary key "employee number". Since the "employee number" index is defined as ascending with no duplicate key values allowed, the sequence of records will print from the lowest value of employee number to the highest value of employee number. This continues until the isread function using ISNEXT returns a -1 with an EENDFILE value in the iserrno field, indicating the end-of-file.

```
#include <isam.h>

#define WHOLEKEY 0
#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char emprec[83];

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;
int eof = FALSE;

/*
 * This program sequentially reads through the
 * employee file by employee number, printing each
 * record to stdout as it goes.
 */

main()
{

int cc;

   fdemploy = cc = isopen("employee", ISINPUT+ISAUTOLOCK);
   if (cc < SUCCESS)
      {
      printf("isopen error %d for employee file\n", iserrno);
      exit(1);
      }
   mkemplkey();
   cc = isstart(fdemploy, &key, WHOLEKEY, emprec, ISFIRST);
   if (cc != SUCCESS)
      {
      printf("isstart error %d\n", iserrno);
      isclose(fdemploy);
      exit(1);
      }
   getfirst();
   while(!eof)
      {
      showemployee();
      getnext();
      }
   isclose(fdemploy);
}

showemployee()

{
   printf("Employee number: %ld", ldlong(emprec));
   printf("\nLast name: ");     putnc(emprec+4, 20);
   printf("\nFirst name: ");    putnc(emprec+24, 20);
   printf("\nAddress: ");       putnc(emprec+44, 20);
   printf("\nCity: ");          putnc(emprec+64, 20);
   printf("\n\n\n");
}
```

```
putnc(c, n)

char *c;
int n;
{
   while(n--) putchar(*(c++));
}

getfirst()

{
int cc;

   if (cc = isread(fdemploy, emprec, ISFIRST))
      {
   switch(iserrno)
         {
         case EENDFILE:  eof = TRUE;
                         break;
         default:
            {
            printf("isread ISFIRST error %d \n", iserrno);
            eof = TRUE;
            return(1);
            }
         }
      }
   return(0);
}

getnext()

{
int cc;
   if (cc = isread(fdemploy, emprec, ISNEXT))
      {
   switch(iserrno)
         {
         case EENDFILE:  eof = TRUE;
                         break;
         default:
            {
            printf("isread ISNEXT error %d \n", iserrno);
            eof = TRUE;
            return(1);
            }
         }
      }
   return(0);
}

mkemplkey()
{
   key.k_flags = 0;                 /* no dups, no compression */
   key.k_nparts = 1;                /* one part index          */

   key.k_part[0].kp_start = 0;      /* offset is zero */
   key.k_part[0].kp_type = LONGTYPE;        /* type long */
   key.k_part[0].kp_leng = 4;       /* 4 bytes */
}
```

Figure 2-5. Program to Access File Sequentially

2.8. Random Access

Figure 2-6 describes how random access to a C-ISAM file can be accomplished. This program interactively retrieves an employee number from standard input, searches for it in the employee file, and prints the results of its search to standard output. Note that the ISEQUAL macro is used to specify the read mode to isread in the C function called "reademp". If no record which corresponds to the value entered by the user is found with the employee number a condition code of ENOREC is returned in iserrno and isread returns a -1. It is the responsibility of the C programmer to handle that return code in an appropriate manner. If ENOREC is returned, the record buffer sent as the record parameter to the isread call will not have been changed (i.e. no record will have been read).

```
#include <isam.h>
#include <stdio.h>

#define WHOLEKEY 0
#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char emprec[83];
long empnum;

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;
int eof = FALSE;

/*
 * This program interactively retrieves an employee's
 * employee number from stdin, searches for it in
 * the employee file, and prints the employee
 * record which has that value as its employee number
 * field.
 */

main()

{
int cc;

   fdemploy = cc = isopen("employee", ISINPUT+ISAUTOLOCK);
   if (cc < SUCCESS)
      {
      printf("isopen error %d for employee file\n", iserrno);
      exit(1);
      }
   mkemplkey();
   getempnum();
   while(empnum != 0)
         {
         if (reademp() == SUCCESS) showemployee();
         getempnum();
         }
   isclose(fdemploy);
}
```

```
getempnum()

{
    char *line;

   printf("Enter the employee number (zero to quit): ");
   getline(line, 80);
   sscanf(line, "%ld", &empnum);
   stlong(empnum, emprec);

}

showemployee()

{
   printf("Employee number: %ld", ldlong(emprec));
   printf("\nLast name: ");      putnc(emprec+4, 20);
   printf("\nFirst name: ");     putnc(emprec+24, 20);
   printf("\nAddress: ");        putnc(emprec+44, 20);
   printf("\nCity: ");           putnc(emprec+64, 20);
   printf("\n\n\n");

}

putnc(c, n)

char *c;
int n;
{
   while(n--) putchar(*(c++));
}

reademp()

{
int cc;

   cc = isread(fdemploy, emprec, ISEQUAL);
   if (cc != SUCCESS)
      {
      switch(iserrno)
         {
         case EENDFILE:
            {
            eof = TRUE;
            break;
            }
         default:
            {
            printf("isread ISEQUAL error %d \n", iserrno);
            eof = TRUE;
            return(1);
            }
         }
      }
   return(0);
}
```

```
mkemplkey()
{
   key.k_flags = 0;       /* no dups, no compression */
   key.k_nparts = 1;      /* one part index          */

   key.k_part[0].kp_start = 0;       /* offset is zero */
   key.k_part[0].kp_type = LONGTYPE; /* type        long */
   key.k_part[0].kp_leng = 4;        /* 4 bytes */
}


getline(s, lim)
char s[];
int lim;
{
  int c, i;

  for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] =c;
  if (c=='\n') {
        s[i] =c;
        ++i;
        }
    s[i] = '\0';
    return(i);
}
```

Figure 2-6. Program to Access File Randomly

2.9. Chaining

The following example shows how to chain to a record that is the last record in a chain of associated records. An illustration of how the performance records appear logically by the primary key follows. The primary index is a composite index made up of the employee number and review date.

Emp. no.	Review date	Job rating	New Salary	New Title
1	790501	g	20,000	PA
1	800106	g	23,000	PA
1	800505	f	24,725	PA
2	760301	g	18,000	JP
2	760904	g	20,700	PA
2	770305	g	23,805	PA
2	770902	g	27,376	SPA
3	800420	f	18,000	JP
4	800420	f	18,000	JP

The following program's function is to interactively add a new performance file record. The record contains the date that the salary review took place, the employee's current job rating, the employee's new salary (based on rating), and the employee's new or current job title. All the fields except new salary are entered by the user. The new salary is calculated by multiplying the employee's most recent salary, which can be found at the end of a "chain" of associated performance history records for that employee, by a factor that depends upon that employee's job rating. To find the most recent performance history record for a given employee, the record pointer in C-ISAM is positioned to the record immediately after the highest possible review date for that employee. In the example, every possible date is smaller than 999999. To retrieve the most recent performance history record for that employee, an isread is executed with the ISPREV option as the mode parameter, This technique is considerably faster than finding the first performance history record for a particular employee, and then executing ISNEXTs to "chain" through them all.

```
#include <isam.h>
#include <stdio.h>

#define WHOLEKEY 0
#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char perfrec[51];
char operfrec[51];
char line[81];
long empnum;
double new_salary, old_salary;

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;
int finished = FALSE;

/*
 * This program interactively reads data from
 * stdin and adds performance records to the
 * "perform" file.
 * Depending on the rating given the employee
 * on job performance the following salary
 * increases are placed in the salary field
 * of the performance file.
 *
 *    rating      percent increase
 *    ------      ----------------
 *    p (poor)        0.0 %
 *    f (fair)        7.5 %
 *    g (good)        15.0 %
 *
 */
```

```
main()

{
int    cc;

  fdperform = cc = isopen("perform", ISINOUT+ISAUTOLOCK);
  if (cc < SUCCESS)
     {
     printf("isopen error %d for performance file\n", iserrno);
     exit(1);
     }
  mkperfkey();
  getperformance();
  while(!finished)
     {
     if (get_old_salary())
        {
        finished = TRUE;
        }
     else
        {
        addperformance();
        getperformance();
        }
     }

     isclose(fdperform);
     }

     addperformance()

    { int cc;

       cc = iswrite(fdperform, perfrec);
       if (cc != SUCCESS)
          {
          printf("iswrite error  %d\n", iserrno);
          isclose(fdperform);
          exit(1);
          }
    }


    getperformance()

    {

          printf("Employee      number (enter 0 to exit): ");
          getline(line, 80);
          sscanf(line, "%ld", &empnum);
          if (empnum == 0)
          {
            finished = TRUE;
             return(0);
          }
          stlong(empnum, perfrec);

          printf("Review Date: ");
          getline(line, 80);
          stchar(line, perfrec+4, 6);

          printf("Job Rating (p=poor, f=fair, g=good): ");
          getline(line, 80);
          stchar(line, perfrec+10,      1);

          printf("Salary After Review: (Sorry, you don't get to add this) \n");
          new_salary = 0.0;
          stdbl(new_salary, perfrec+11);

          printf("Title After Review: ");
          getline(line, 80);
          stchar(line, perfrec+19,      30);

          printf("\n\n\n");
      }
```

```
get_old_salary()

{ int mode, cc;
   stchar(perfrec, operfrec, 4);  /* get id number */
   stchar("999999", operfrec+4, 6); /*largest  possible
data */

   cc  =  isstart(fdperform,  key,  WHOLEKEY,  operfrec,
ISGTEQ);
   if (cc !=SUCCESS)

{
switch(iserrno)
   {
   case ENOREC:
   case EENDFILE: mode = ISLAST;
                  break;
   default:          printf("isstart    error    %d ",iserrno);
                  printf("in get_old_salary\n");
                  return(1);
   }
}
 else
{
mode = ISPREV;
}
 cc = isread(fdperform, operfrec,      mode);
 if (cc != SUCCESS)
{
printf("isread  error    %d   in   get_old_salary",
iserrno);
return(1);
}
 if (cmpnbytes(perfrec, operfrec,      4))
{
printf("No performance record  for employee  number %ld.\n",
iserrno);
return(1);
}
 else
{
printf("\nPerformance  record found.\n\n");
old_salary = new_salary = lddbl(operfrec+11);
printf("Rating: ");
switch(*(perfrec+10))
   {
   case 'p':       printf("poor\n");
                    break;
   case 'f':       printf("fair\n");
                    new_salary *= 1.075;
                    break;
   case 'g':       printf("good\n");
                    new_salary *= 1.15;
                    break;

   }
stdbl(new_salary, perfrec+11);
printf("Old salary was %f\n",  old_salary);
printf("New salary is  %f\n", new_salary);
}
}
getline(s, lim)
char s[];
int lim;
{
   int c, i;

   for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
       s[i] =c;
   if (c=='\n') {
       s[i] =c;
       ++i;
   }
```

```
    s[i] = '\0';
    return(i);
}

mkperfkey()
{
    key.k_flags = 0;                /* no dups, no compression*/
    key.k_nparts = 2;

    key.k_part[0].kp_start =      0;      /* offset is zero */
    key.k_part[0].kp_type = LONGTYPE;     /* type long */


    key.k_part[1].kp_start =      4;      /* offset is four */
    key.k_part[1].kp_type = CHARTYPE;     /* type char */
    key.k_part[1].kp_leng = 6;    /* 6 bytes */

}
stchar(a,b,c)
char *a, *b;
int c;
{
    register int i;

    for (i=0;*a && (i < c); i++)
     *b++ = *a++;
    return(0);
}

cmpnbytes(a,b,c)
char *a, *b;
int c;
{
    register int i;

    for (i=0; i < c; i++)
      if (*a++ != *b++)
     return(1);
    return(0);
}
```

Figure 2-7. Program to Interactively Add Record

SECTION 3
INDEX COMPRESSION AND CHECKING

3.1. Introduction

Index compression, the ability to compress key values in an index file to save space and enhance performance, and index checking, the abiltiy to check and repair index files, are the subject of Section 3.

3.2. Index Compression

C-ISAM can compress key values held in the index files, using three types of compression: leading character compression (LCOMPRESS), trailing character compression (TCOMPRESS) and duplicate compression (DCOMPRESS). In addition to disk space savings, key compression can improve real time response to random access requests, by increasing the number of key values which can be held in an index file page. With more key values per index file page, fewer disk accesses are neccessary to find any given data record. Since disk accesses use the overwhelming percentage of real time during file accesses, key compression in index files can improve real time response. This improvement becomes more dramatic as field size increases and where duplicate values, leading duplicate characters, and trailing blanks become a large percentage of the characters of the key. With the use of leading compression alone, a savings of 5 percent in index page size can result. Much more dramatic savings can result if trailing blanks are compressed as well. Assuming a field length of 20, the page size savings from using both TCOMPRESS and LCOMPRESS is 67.5 percent.

There are some disadvantages to compression: LCOMPRESS and TCOMPRESS each add one byte, DCOMPRESS adds two bytes, and all three combined (COMPRESS) add four bytes to each index entry.

Figures 3-1 through 3-3 illustrate LCOMPRESS, LCOMPRESS and TCOMPRESS, and combined compression (LCOMPRESS, TCOMPRESS, and DCOMPRESS.)

Key Value	Compressed with LCOMPRESS	Bytes Saved
Abbot.....	Abbot.....	-1*
Able.....	le.....	+1
Acre.....	cre.....	Ø
Albert.....	lbert.....	Ø
Albertson.....	son.....	5
Morgan.....	Morgan.....	-1
McBride.....	cBride.....	Ø
McCloud.....	Cloud.....	1
Richards.....	Richards.....	-1
Richardson.....	on.....	6
2ØØ bytes	18Ø bytes	1Ø bytes 5 %

* There is a one byte penalty for LCOMPRESSION

Figure 3-1. Leading Character Compression

Key Value	Compressed with LCOMPRESS + TCOMPRESS	Bytes Saved
Abbot.....	Abbot	13*
Able.....	le	16
Acre.....	cre	15
Albert.....	lbert	13
Albertson.....	son	15
Morgan.....	Morgan	12
McBride.....	cBride	12
McCloud.....	Cloud	13
Richards.....	Richards	1Ø
Richardson.....	on	16
2ØØ bytes	135 bytes	67.5%

* There is a two-byte penalty for using LCOMPRESS and TCOMPRESS

Figure 3-2. Leading and Trailing Character Compression

The third compression method is duplicate compression (DCOMPRESS.) When duplicate entries are allowed, DCOMPRESS can be used to eliminate them. Fields holding city or state values are often duplicate intensive. Figure 3-3 illustrates duplicate compression combined with leading and trailing character compression (COMPRESS).

Key Value	Compressed with LCOMPRESS + TCOMPRESS + DCOMPRESS	Bytes Saved
Abbot.....	Abbot	11*
Abbot.....	(no entry)	16
Abbot.....	(no entry)	16
Able.....	le	14
Able.....	(no entry)	16
Acre.....	cre	13
Albert.....	lbert	11
Albertson.....	son	13
Albertson.....	(no entry)	16
Morgan.....	Morgan	1Ø
McBride.....	cBride	1Ø
McCloud.....	Cloud	11
Richards.....	Richards	8
Richardson.....	on	14
Richardson.....	(no entry)	16
3ØØ bytes	46 bytes	195 bytes 65 %

* COMPRESS (all three compression types) adds four bytes per entry (with DCOMPRESS adding two of the four bytes)

Figure 3-3. Combined Compression

3.3. Index Checking

The bcheck program checks and repairs index files. Bcheck checks the consistency of the files which have the .dat or .idx suffix. The options and syntax for bcheck are listed below. If there seems to be a problem with corrupted indexes, bcheck should be run on the suspect files. Unless the -n or -y option is used, bcheck is interactive and waits for the user to respond to each error that is found. The -y option should be used with caution. Bcheck should not be run using the -y option if it is the first time the files are being checked.

```
usage: bcheck -ilny cisamfiles
       -i   check index file only
       -l   list entries in b-trees
       -n   answer no to all questions
       -y   answer yes to all questions
```

The following is an example of a bcheck run with no errors. (Note that for each index a group of numbers is printed out. There can be up to eight groups of numbers for each index. These numbers indicate the position of the key in each record and are for use in reporting problems to Zilog.)

The command used for this bcheck run was:

```
bcheck sale.pros

BCHECK C-ISAM B-tree Checker version 1.ØØ
Copyright (C) 1982, Relational Database Systems, Inc.
Software Serial Number 1

C-ISAM File: sale.pros.idx

** Check Dictionary
** Check Data File Records
** Check Indexes and Key Descriptions
**    Index 1 = unique key (Ø,4,2)
**    Index 2 = unique key (1Ø,2,1)
**    Index 3 = unique key(62,35,Ø)
**    Index 4 = duplicates (37,25,Ø)
**    Index 5 = duplicates (264,2Ø,Ø)
** Check Data Record and Index Node Free Lists
479 index node(s) used, Ø free -- 2638 data record(s) used, Ø
```

The following is a sample run of bcheck where errors were found.

The -n option was used to answer no to all questions. The command used was:

```
bcheck -n sale.ship.idx

BCHECK C-ISAM B-tree Checker version 1.00 Copyright
(C) 1982, Relational Database Systems, Inc. Software
Serial Number 1

C-ISAM File:  sale.ship.idx

** Check Dictionary and File Sizes ** Check Data File Records
** Check Indexes and Key Descriptions ** Index 1 = unique
key (0,4,2)

ERROR:  12 bas data record(s) Delete index ? no

** Index 2 = duplicates (4,2,1)

ERROR:  12 bad data record(s) Delete index ? no

** Index 3 = duplicates (6,6,0)

ERROR:  12 bas data record(s) Delete index ? no

** Check Data Record and Index Node Free Lists

ERROR:  12 missing data record(s) Fix data record free
list ? no

5 index node(s) used, 0 free -- 0 data record(s) used,
12 free
```

In this case, the indexes must be deleted and rebuilt. To correct these indexes, the -y option would be used to answer yes to all questions asked by bcheck.

The command used to correct the errors was:

```
bcheck -y sale.ship.idx

BCHECK C-ISAM B-tree Checker version 1.00
Copyright (C) 1982, Relational Database Systems, Inc.
Software Serial Number 1

C-ISAM File: sale.ship.idx

** Check Dictionary and File Sizes
** Check Data File Records
** Check Indexes and Key Descriptions
**    Index 1 = unique key (0,2,4)
```

```
ERROR:  12 bad data record(s)
Delete index ? yes

Remake index ? yes

**     Index 2 = duplicates (4,3,1)

ERROR: 12 bas data record(s)
Delete index ? yes

Remake index ? yes

**     Index 4 = duplicates (6,6,Ø)

ERROR:  12 bad data record(s)
Delete index ? yes

Remake index ? yes

** Check Data Record and Index Node Free Lists

ERROR:  12 missing data record(s)
Fix data record free list ? yes

** Recreate Data Record Free List
** Recreate Index 3
** Recreate Index 2
** Recreate Index 1

5 index node(s) used, Ø dree -- Ø date record(s) used,
12 free
```

SECTION 4
FILE AND RECORD LOCKING

4.1. Introduction

There are two levels of locking available from C-ISAM -- file-level locking, and record-level locking. Within these two levels, C-ISAM offers several methods from which to choose.

4.2. File Locking

File locking can be accomplished in two ways: exclusive file locking and manual file locking.

4.2.1. Exclusive File Locking:

Exclusive file locking prevents other processes from either reading from or writing to a given C-ISAM file. This lock remains in effect from the moment the file is opened, using isopen or isbuild, until the file is closed using isclose. Exclusive file locking is specified by adding ISEXCLLOCK to the mode parameter of the isopen or isbuild function call. Exclusive file level locking is not necessary for most situations, but it must be used when an index is being added using isaddindex, or when an index is being deleted using isdelindex. The skeleton program shown below illustrates how exclusive file level locking is done.

```
myfd = isopen("myfile", ISEXCLLOCK+ISINOUT);

        .
        .
        .

isclose(myfd);
```

4.2.2. Manual File Locking:

The manual file level locking method is referred to as a "shared" lock. It prevents other processes from writing to a given C-ISAM file, but allows other processes to read the locked C-ISAM file. Shared file level locking is specified with the islock and isunlock function calls (MODE ISINPUT

specified). When a C-ISAM file is to be locked in this manner, C-ISAM must first be notified of the user's intention to use manual locking. This is done by adding ISMANULOCK to the mode parameter of the isopen or isbuild call. Later in the program, when locking is desired, islock should be called to lock the file. When the file is to be unlocked, isunlock should be called.

```
myfd = isopen("myfile", ISMANULOCK+ISINOUT);
        .
        . ("myfile" is unlocked in this section)
        .
islock(myfd);
        .
        . ("myfile" is locked in this section)
        .
isunlock(myfd);
        .
        . ("myfile" is unlocked in this section)
        .
isclose(myfd);
```

4.3. Record Locking

There are two basic types of record level locking implemented in C-ISAM -- Automatic and Manual.

Automatic record locking locks a record just before it is read using the isread call. It unlocks the record after the next C-ISAM call has completed. Automatic record locking locks one record at a time without regard to the length of time it is locked.

Manual record locking, on the other hand, can lock any number of records. Manual locking locks a record when that record is read using isread. It unlocks that record, and any other records that are currently locked, when isreleaseis called. Manual record locking is used when more control is required over when a record, or set of records, is to be locked and unlocked.

Both automatic and manual locking techniques are "shared" locks. Other processes may read records locked by the current process, but they may not lock or re-write them.

4.3.1. Automatic Record Locking:

Automatic record locking must be specified to C-ISAM when the file is opened. This is done by adding ISAUTOLOCK to the mode parameter of the isopen or isbuild function call. From when the file is opened until it is closed, every record will be locked automatically before it is read. Each record remains locked until the next C-ISAM function call is completed for the current file. Therefore, while using the automatic record locking mechanism of C-ISAM, only one record per C-ISAM file can be locked at a given time. An example of automatic record locking is shown below.

```
myfd = isopen("myfile", ISINOUT+ISAUTOLOCK);
        .
        .
        .

isread(myfd, myrecord, ISNEXT); /* record locked here */
                                /* before record is read */
        .
        .
        .

isrewcurr(myfd, myrecord);      /* record unlocked here */
                                /* after completion     */
        .
        .
        .

isclose(myfd);
```

4.3.2. Manual Record Locking:

The user's intention to use manual record locking must be specified before any processing takes place. This is done by adding ISMANULOCK to the mode parameter of isopen or isbuild function calls when the file is opened. After the file is open, if the user wishes a record to be locked, ISLOCK must be added to the mode parameter of the isread function call which is reading that record. Each and every record which is read in this manner remains locked until they are all unlocked by a call of the isrelease function of C-ISAM. The number of records which may be locked in this manner at any one time is operating system dependent. The following example shows how a number of records in a particular file are locked and unlocked using manual record locking.

```
myfd = isopen("myfile", ISINOUT+ISMANULOCK);
        .
        .
        .
isread(myfd, first_record, ISEQUAL+ISLOCK);
        .
        .
        .
isread(myfd, second_record, ISEQUAL+ISLOCK);
        .
        .
        .
isread(myfd, third_record, ISEQUAL+ISLOCK);
        .
        .
        .
isrelease(myfd);          /* unlock all three records */
        .
        .
        .
isclose(myfd);
```

APPENDIX A
C-ISAM CALLS IN SUMMARY

Appendix A summarizes the C-ISAM function calls, which are described in detail in Section 3 of the System 8000 ZEUS Reference Manual.

All calls to C-ISAM return either a 0 or -1 as the value of the function and set the global integer iserrno to either 0 or an error indicator. In the case of isbuild and isopen, the return value will be a legal C-ISAM file descriptor or a -1. In the case of the other calls, the return value will be a 0 or a -1. A -1 indicates that an error has occurred, and iserrno has been set.

```
isbuild(filename, recordlength, keydesc, mode)
char *filename;
int recordlength;
struct keydesc *keydesc;
int mode;

isopen(filename, mode)
char *filename;
int mode;

isclose(isfd)
int isfd;

isaddindex(isfd, keydesc)
int isfd;
struct keydesc *keydesc;

isdelindex(isfd, keydesc)
int isfd;
struct keydesc *keydesc;

isstart(isfd, keydesc, length, record, mode)
int isfd;
struct keydesc *keydesc;
int length;
char record[];
int mode;

isread(isfd, record, mode)
int isfd;
char record[];
int mode;
```

```
iswrite(isfd, record)
int isfd;
char record[];

isrewrite(isfd, record)
int isfd;
char record[];

isrewcurr(isfd, record)
int isfd;
char record[];

isdelete(isfd, record)
int isfd;
char record[];

isuniqueid(isfd, uniqueid)
int isfd;
long *uniqueid

isindexinfo(isfd, buffer, number)
int isfd;
struct keydesc *buffer;
int number;

isaudit(isfd, filename, mode)
int isfd;
char *filename;
int mode;

iserase(filename)
char *filename;

islock(isfd)
int isfd;

isunlock(isfd)
int isfd;

isrelease(isfd)
int isfd;

isrename(oldname, newname)
```

isld -- load routines

isst -- store value routines

APPENDIX B
ERROR MESSAGES AND STATUS BYTES

B.1. Error Messages

When a C-ISAM error occurs, iserrno can assume values ranging from 1 to 113. ZEUS errors range from 1 - 99 and C-ISAM errors range from 100 - 113. ZEUS error codes that can appear in errno can also appear in iserrno.

Table B-1 defines C-ISAM error codes.

Table B-1. C-ISAM Error Codes

Macro	Number	Text	Status Byte 1	Status Byte 2
EDUPL	100	An attempt was made to add a duplicate value to an index via iswrite, isrewrite, isrewcurr or isaddindex.	2	2
ENOTOPEN	101	An attempt was made to perform some operation on a C-ISAM file that was not previously opened using the isopen call.	9	0
EBADARG	102	One of the arguments of the C-ISAM call is not within the range of acceptable values for that argument.	9	0
EBADKEY	103	One or more of the elements that make up the key description is outside of the range of acceptable values for that element.	9	0
ETOOMANY	104	The maximum number of files that may be open at one time would be exceeded if this request were processed.	9	0
EBADFILE	105	The format of the C-ISAM file has been corrupted.	9	0
ENOTEXCL	106	In order to add or delete an index, the file must have been opened with exclusive access.	9	0

ELOCKED	107	The record or file requested by this call cannot be accessed because it has been locked by another user.	9	0
EKEXISTS	108	An attempt was made to add an index that has been defined previously.	9	0
EPRIMKEY	109	An attempt was made to delete the primary key value. The primary key may not be deleted by the isdelindex call.	9	0
EENDFILE	110	The beginning or end of file was reached.	1	0
ENOREC	111	No record could be found that contained the requested value in the specified position.	2	3
ENOCURR	112	This call must operate on the current record. One has not been defined.	2	1
EFLOCKED	113	The file is exclusively locked by another user.		

B.2. Status Bytes

Two bytes (isstat1 and isstat2) hold status information after C-ISAM calls. The first byte (Table B-2) holds status information of a general nature, such as success or failure of a C-ISAM call. The second byte (Table B-3) contains more specific information, which has meaning based on the status code in byte one.

Table B-2. Status Byte One

Byte One Value	Status
Ø	Successful Completion
1	End of File
2	Invalid Key
3	System Error
9	User Defined Errors

Table B-3. Status Byte Two

Status Byte One	Status Byte Two
Ø - 9	Ø - No further information is available
Ø	2 - Duplicate key indicator - After a READ indicates that the key value for the current key is equal to the value of that same key in the next record. - After a WRITE or REWRITE indicates that the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.
2	1 - The primary key value has been changed between the successful execution of a READ statement and the execution of the next REWRITE statement. 2 - An attempt has been made to write or rewrite a record that would create a duplicate key in an indexed file. 3 - No record with the specified key can be found. 4 - An attempt has been made to write beyond the externally defined boundaries of an indexed file.
9	The value of status key two is defined by the user.

APPENDIX C
Data Types

The types of data which can be defined and manipulated using C-ISAM functions are described in this Appendix. Descriptions of how each data type is stored in data files and how each data type must be treated are also discussed.

The data types for which C-ISAM can maintain properly ordered indexes are character, 2 byte integer, 4 byte integer, machine float (floating point), and machine double (double precision floating point). The macro definitions used to describe these types to C-ISAM are shown below. These definitions can also be found in "isam.h".

CHARTYPE	character
INTTYPE	2 byte integer
LONGTYPE	4 byte integer
FLOATTYPE	machine float
DOUBLETYPE	machine double

C.1. CHARTYPE

The data type CHARTYPE signifies to C-ISAM that a particular region of a data file consists of byte values from 0 to 255. A typical example of data type CHARTYPE is a city name or an address field.

C.2. INTTYPE and LONGTYPE

The data type INTTYPE and LONGTYPE consist of 2 and 4 byte binary signed integer data. Integer data is always stored in the data and index files as high/low, most significant byte first, least significant byte last. This storage technique is independent of the form in which integers are stored on the System 8000, although there is no difference between the integer and long formats used by C-ISAM and their C language counterparts, except that the C-ISAM values can be placed on non-word boundaries. Four routines are supplied to the user of C-ISAM for the conversion to and from C-ISAM integer storage format.

ldint(p)
: which returns a machine-format integer if p is a char pointer to the starting byte of a C-ISAM-format

2 byte integer;

stint(i, p)
which stores a machine-format integer i as a C-ISAM-format 2 byte integer at location p where p is a char pointer to the first byte of a C-ISAM-format 2 byte integer;

ldlong(p)
which returns a machine-format 4 byte integer if p is a char pointer to the first byte of C-ISAM-format 4 byte integer;

stlong(l, p)
which stores a machine-format integer i as a C-ISAM-format 4 byte integer at location p where p is a char pointer to the first byte of a C-ISAM-format 4 byte integer;

The typical use for the above routines is after a C-ISAM data record has been read into the user buffer. Integer values which are to be used by the user program first have to be converted to machine usable format by using ldint() for type INTTYPE and ldlong() for LONGTYPE. This is shown below.

```
int  int_machine;
long long_machine;
char *p_cisam_int, *p_cisam_long;

     . . .

int_machine  = ldint(p_cisam_int);
long_machine = ldlong(p_cisam_long);
```

Storage of machine-format integer data as C-ISAM-format integer data requires the use of the stint() and stlong() routines.

```
stint(int_machine, p_cisam_int);
stlong(long_machine, p_cisam_long);
```

Note that the C-ISAM formatted integers need not be aligned along word boundaries as do machine formatted integers.

C.3. FLOATTYPE and DOUBLETYPE

The Bata type FLOATTYPE and DOUBLETYPE are the two floating point data types. The data type FLOATTYPE is the same as the C data type float while the data type DOUBLETYPE is the

same as the C data type double. There is no difference between the floating point format used by C-ISAM and its counterpart in the C language except that C-ISAM floating point numbers can be placed on non-word boundaries. Four additional routines have been provided to the C-ISAM user to retrieve or replace these non-aligned floating point numbers from their positions in C-ISAM data records.

ldfloat(p)
: which returns a machine-format float if p is a char pointer to the starting byte of a C-ISAM-format FLOATTYPE;

stfloat(f, p)
: which stores a machine-format float f as a C-ISAM-format FLOATTYPE at location p where p is a char pointer to the starting (leftmost) byte of a C-ISAM-format FLOATTYPE;

lddbl(p)
: which returns a machine-format double if p is a char pointer to the starting byte of a C-ISAM-format DOUBLETYPE;

stdbl(d, p)
: which stores a machine-format double d as a C-ISAM-format DOUBLETYPE at location p where p is a char pointer to the starting (leftmost) byte of a C-ISAM-format DOUBLETYPE.

The use of the floating point load and store routines is analogous to the use of the integer load and store routines.

APPENDIX D
HEADER FILES

D.1. The Header File Isam.h

The C-ISAM header file, isam.h, contains defines that are used for the mode arguments and also definitions of structures that are used in the functions.

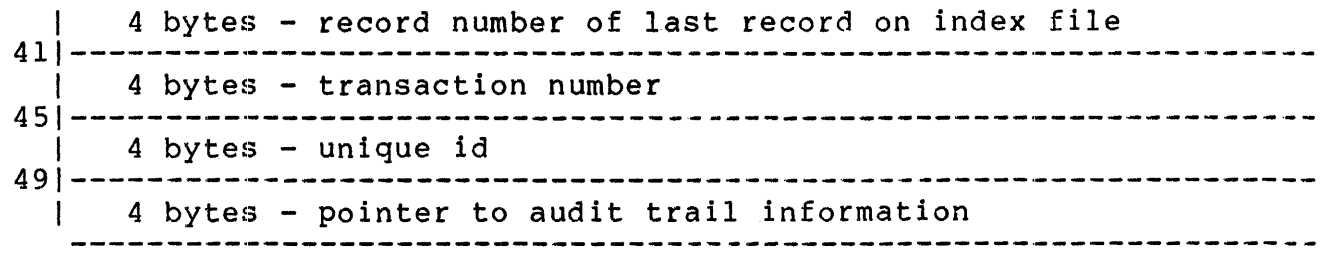
```
#define CHARTYPE      0
#define INTTYPE       1
#define LONGTYPE      2
#define FLOTYPE       3
#define DBLTYPE       4
#define MAXTYPE       5
#define ISDESC        0200

#define ISFIRST       0      /* position to first record      */
#define ISLAST        1      /* position to last record       */
#define ISNEXT        2      /* position to next record       */
#define ISPREV        3      /* position to previous record   */
#define ISCURR        4      /* position to current record    */
#define ISEQUAL       5      /* position to equal value       */
#define ISGREAT       6      /* position to greater value     */
#define ISGTEQ        7      /* position to >= value          */

/* isread lock modes */
#define ISLOCK      (1<<8)      /* lock record before reading   */

/* isopen, isbuild lock modes */
#define ISAUTOLOCK (3<<8)       /* automatic record lock        */
#define ISMANULOCK (4<<8)       /* manual record lock           */
#define ISEXCLLOCK (5<<8)       /* exclusive isam file lock     */

#define ISINPUT       0      /* open for input only           */
#define ISOUTPUT      1      /* open for output only          */
#define ISINOUT       2      /* open for input and output     */

/* audit trail mode parameters */
#define AUDSETNAME    0      /* set new audit trail name      */
#define AUDGETNAME    1      /* get audit trail name          */
#define AUDSTART      2      /* start audit trail             */
#define AUDSTOP       3      /* stop audit trail              */

#define NPARTS        8      /* maximum number of key parts   */

struct keypart
```

```
    {
    int kp_start;           /* starting byte of key part    */
    int kp_leng;            /* length in bytes              */
    int kp_type;            /* type of key part             */
    };

struct keydesc
    {
    int  k_flags;           /* flags                        */
    int  k_nparts;          /* number of parts in key       */
    struct keypart
        k_part[NPARTS];     /* each key part                */
    };

#define ISDUPS    001       /* duplicates allowed           */
#define DCOMPRESS 002       /* duplicate compression        */
#define LCOMPRESS 004       /* leading compression          */
#define TCOMPRESS 010       /* trailing compression         */
#define COMPRESS  016       /* all compression              */

struct dictinfo
    {
    int  di_nkeys;          /* number of keys defined       */
    int  di_recsize;        /* data record size             */
    int  di_idxsize;        /* index record size            */
    long di_nrecords;       /* number of records in file    */
    };
```

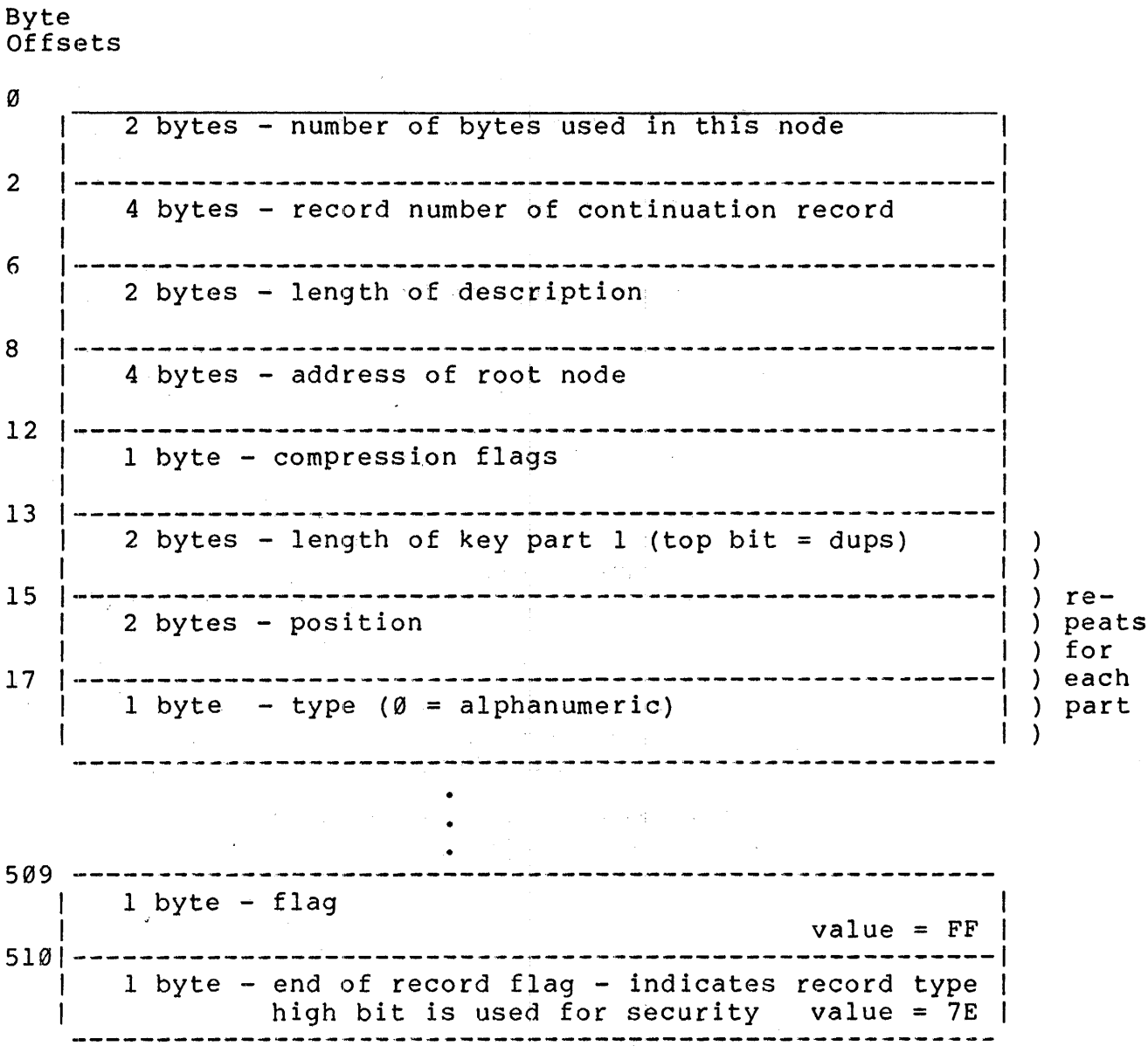
APPENDIX E
FILE FORMATS

DICTIONARY FORMAT

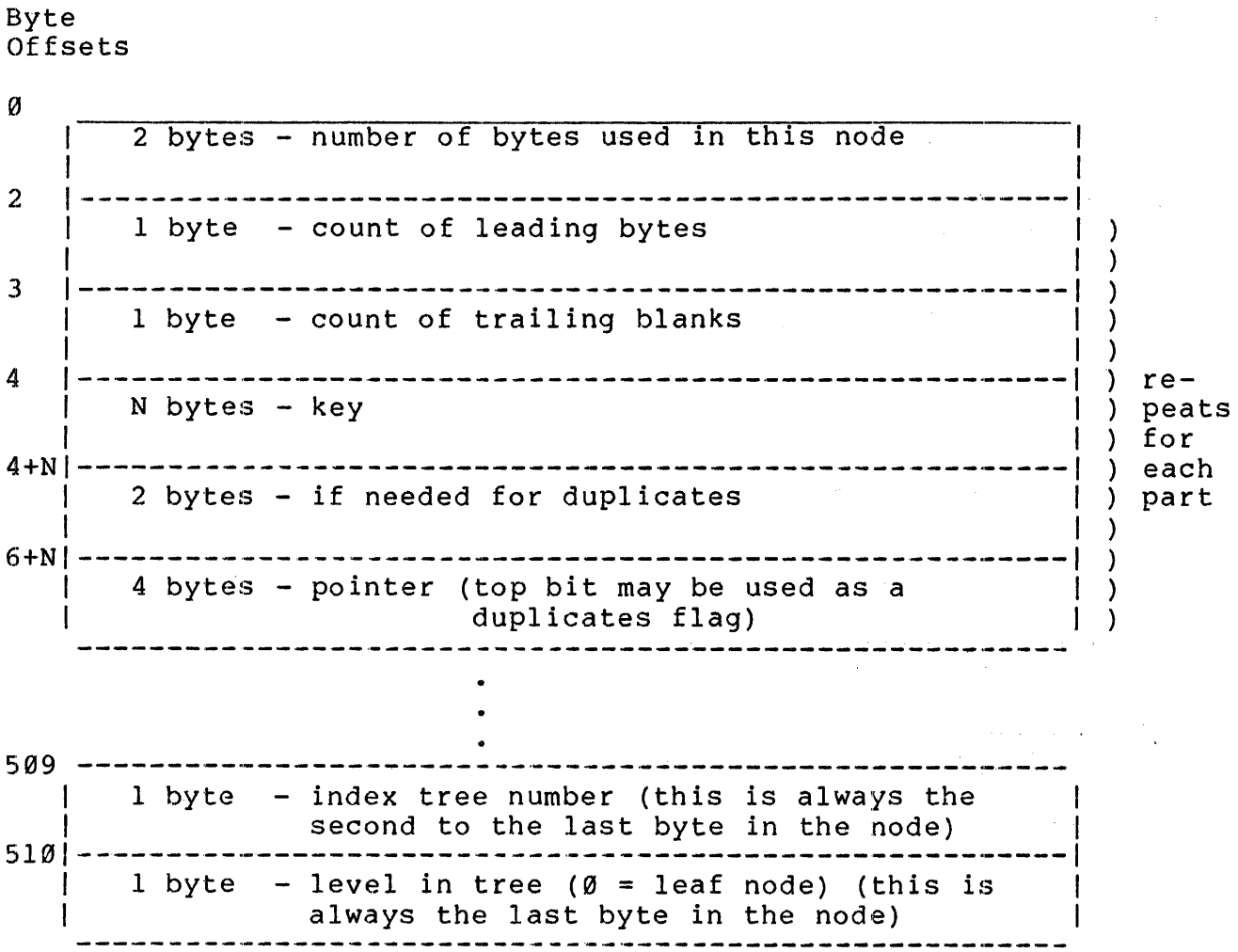
Byte Offsets		
0	2 bytes - validation	value = FE53
2	1 byte - number of reserved bytes at start of index record	value = 2
3	1 byte - number of reserved bytes at end of index record	value = 2
4	1 byte - number of reserved bytes per key entry includes record number)	value = 4
5	1 byte - pointer type and length indicator	value = 4
6	2 bytes - index file record length (excludes relative file flag bytes)	value = 511
8	2 bytes - number of keys	
10	2 bytes - flags (see explanation of flags, next page)	value = 0
12	1 byte - file version number	
13	2 bytes - data record length (excludes relative file flag bytes)	
15	4 bytes - record number of first key information record	
19	6 bytes - reserved for future use	
25	4 bytes - record number of first data free space record	
29	4 bytes - record number of first index free space record	
33	4 bytes - record number of last record on data file	
37		


4 bytes - record number of last record on index file
41
4 bytes - transaction number
45
4 bytes - unique id
49
4 bytes - pointer to audit trail information

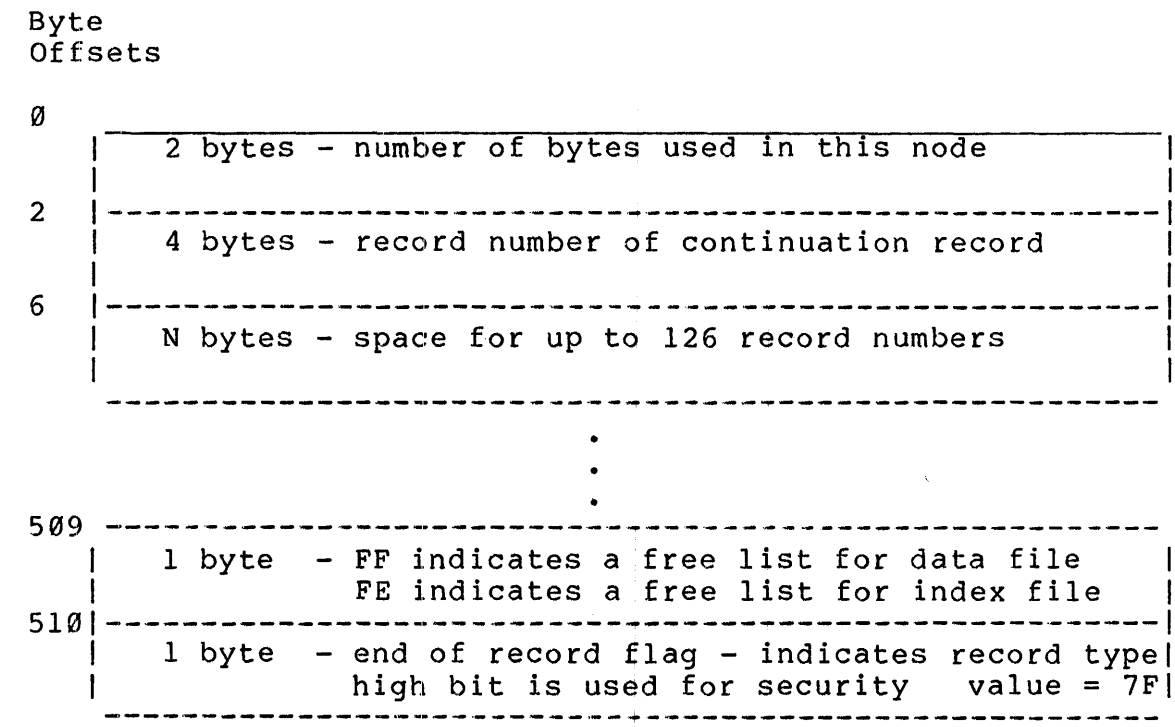
KEYDESCRIPTION FORMAT



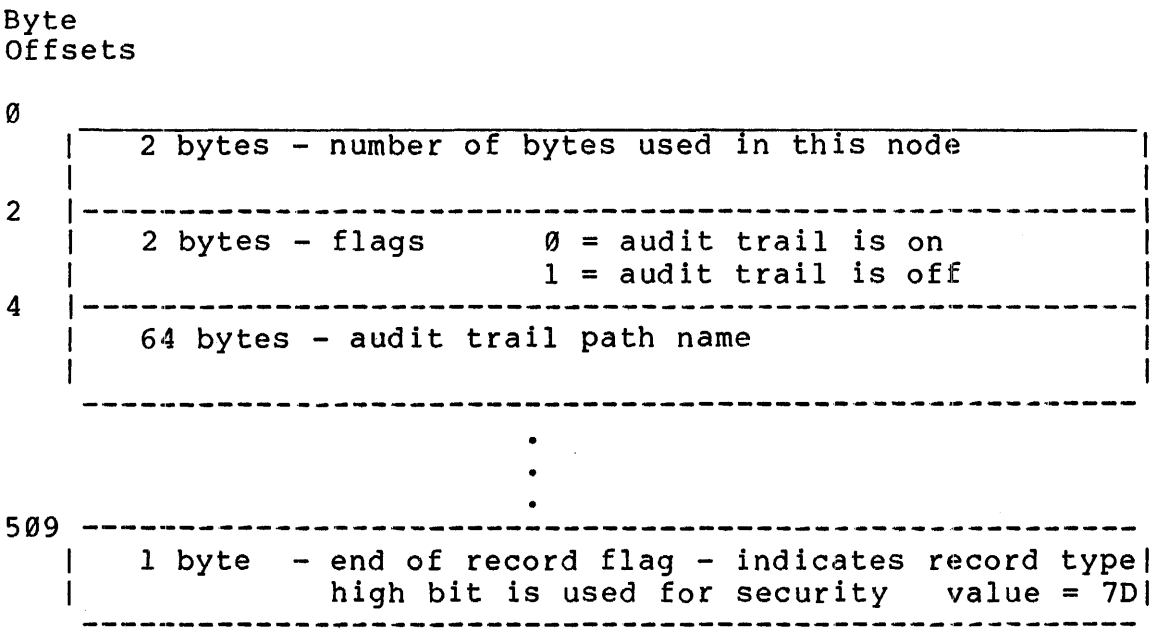
TREENODE FORMAT



FREELIST FORMAT



AUDITTRAIL NODE FORMAT



Screen Updating and Cursor Movement Optimization: A Library Package *

* This information is based on an article originally written by Kenneth C. R. C. Arnold

Preface

This document describes a package of C library functions which allow the user to:

(1) update a screen with reasonable optimization,

(2) get input from the terminal in a screen-oriented fashion, and

(3) independent from the above, move the cursor optimally from one point to another.

These routines all use the /etc/termcap database to describe the capabilities of the terminal.

Table of Contents

SECTION 1 SCREEN PACKAGE 1-1

1.1. Usage 1-1
1.1.1. Updating the Screen 1-2
1.1.2. Conventions 1-2
1.1.3. Terminal Environment 1-3
1.1.4. Screen Initialization 1-4
1.1.5. Screen Scrolling 1-4
1.1.6. Screen Updating 1-5
1.1.7. Screen Input 1-5
1.1.8. Exit Processing 1-5
1.1.9. Internal Description 1-6

SECTION 2 CURSES FUNCTIONS 2-1

APPENDIX A EXAMPLE A A-1

A.1. Variables Set By setterm() A-1
A.2. Variables Set By gettmode() A-2

APPENDIX B EXAMPLE B B-1

SECTION 1
SCREEN PACKAGE

With this package the C programmer can do the most common type of terminal dependent functions; that is, movement optimization and optimal screen updating.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

Screen updating and input can be done without using any programmer knowledge of motion optimization or the database itself. The motion optimization can be used without either of the other two.

In this document, the following terminology is used:

> window: An internal representation containing any image of what a section of the terminal screen can look like at some point in time. This subsection can encompass the entire terminal screen or any smaller portion down to a single character within that screen.
>
> terminal: Also called terminal screen, the current image on the terminal's screen.
>
> screen: A subset of windows as large as the terminal screen. One of these, stdscr, is automatically provided for the programmer.

1.1. Usage

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have:

```
#include <curses.h>
```

at the top of the source program. The header file **<curses.h>** includes **<sgtty.h>** and **<stdio.h>**. It is redundant (but harmless) for the programmer to define them again in the source program.

Compilations must have the following form:

cc [flags] file ... **-lcurses -ltermlib**

1.1.1. Updating the Screen: A data structure (WINDOW) describes a window image to the routines, including its starting position on the screen (the y, x of the upper left hand corner) and its size. One of these structures, called curscr (current screen), is a screen image of what is currently on the screen. Another structure, stdscr (standard screen) is provided by default for making changes.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal screen. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When a window describes what a part of a terminal should look like, the routine refresh() (or wrefresh()) makes the terminal, in the area covered by the window, look like that window. Changing something on a window does not change the terminal. Actual updates to the terminal screen are made only by calling refresh() or wrefresh(). This allows the programmer to maintain several different ideas of how a portion of the terminal screen should look. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this" and the package takes the best way to do it.

The routines can use several windows, but two are automatically given: curscr knows what the terminal looks like, and stdscr knows what the programmer wants the terminal to look like next. The user never accesses curscr directly. Changes are made to the appropriate screen, and then the routine refresh() (or wrefresh()) is called.

1.1.2. Conventions: Many functions are set up to deal with stdscr as a default screen. For example, to add a character to stdscr, call addch() with the desired character. If a different window is to be used, the routine waddch() (for window-specific addch()) is provided. The routine addch() is a "#define" macro with arguments using stdscr as a default. The convention of prepending function names with a "w" when applied to specific windows is consistent. The only routines that do not adhere to this convention are when a window must be specified.

To move the current (y, x) from one point to another, the routines move() and wmove() are provided. However, it is often desirable to first move and then perform the I/O operation. Most I/O routine names can be preceded by the

prefix "mv" and the desired (y, x) coordinates are added to the function arguments. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y  ,x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win,y  ,x  ,ch);
```

Note the window description pointer, win, comes before the added (y, x) coordinates. If win pointers are needed, they are always the first parameters passed.

1.1.3. Terminal Environment: Many variables to describe the terminal environment are available to the programmer. They are:

Type	Name	Description
WINDOW	*curscr	current version of the screen (terminal screen)
WINDOW	*stdscr	standard screen; most updates are usually done here
char	*Def_term	default terminal type if type cannot be determined
bool	My_term	use the terminal specifications in Def term as terminal, irrelevant of real terminal type
char	*ttytype	full name of current terminal
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal

int	ERR	error flag returned by routines on a fail
int	OK	error flag returned by routines when successful

There are also several "#define" constants and types which are useful:

reg	storage class "register" (for example, reg int;)
bool	boolean type, actually a "char" (for example, bool doneit;)
TRUE	boolean "true" flag (1)
FALSE	boolean "false" flag (Ø)

1.1.4. Screen Initialization: To use the screen package, the routines must know about terminal characteristics and the space for curscr and stdscr must be allocated. These functions are performed by initscr(). Since it must allocate space for the windows, it can overflow core when attempting to do so. When this occurs, initscr() returns ERR. The routine initscr() must be called before any routines which affect windows are used. If not, the program will core dump as soon as either curscr or stdscr are referenced. Terminal status changing routines, like nl() and crmode(), should be called after initscr().

1.1.5. Screen Scrolling: When the screen windows have been allocated, they can be set up for the run. To allow the window to scroll, use scrollok(). For the cursor to be left after the last change, use leaveok(). Otherwise, refresh moves the cursor to the window's current (y, x) coordinates after updating it. New windows are created by newwin() and subwin(). The routine delwin() gets rid of old windows. To change the official size of the terminal by hand, set the variables LINES and COLS, and then call initscr(). This is best done before the first call to initscr(), but can be done after, as initscr() deletes any existing stdscr and/or curscr before creating new ones.

1.1.6. Screen Updating: The basic functions to change what goes on a window are addch() and move(). The routine addch() adds a character at the current (y, x) coordinates returning ERR if it would cause the window to scroll illegally (print a character in the lower right hand corner of a terminal which automatically scrolls if scrolling is not allowed).

The routine move() changes the current (y, x) coordinates. If move() causes the new coordinates to move off the window when scrolling is not allowed, ERR is returned. As mentioned in section 1.1.2, the two can be combine into mvaddch() to do both in one function call.

The other output functions, such as addstr() and printw(), call addch() to add characters to the window.

After the window is modified as desired, call refresh() to display it. To optimize finding changes, refresh() assumes that any part of the window not changed since the last refresh of that window has not been changed on the terminal; that is, a portion of the terminal has not been refreshed with an overlapping window. Otherwise, the routine touchwin() is provided to cause the entire window to be changed, making refresh() check the whole subsection of the terminal for changes.

If wrefresh() is called with curscr, the current screen is displayed. This is useful for implementing a command to redraw the screen if necessary.

1.1.7. Screen Input: Input is essentially a mirror image of output. The complementary function to addch() is getch() which, if echo is set, calls addch() to echo the entire character. If the terminal is not in raw or cbreak mode, getch() sets it to cbreak, and reads in the character.

1.1.8. Exit Processing: To do certain optimizations, some things must be done before the screen routines start up. These functions are performed in gettmode() and setterm(), which are called by initscr(). The routine endwin() cleans up after the routines. It restores the terminal modes to what they were when initscr was first called. Thus, anytime after the call to initscr(), endwin() must be called before exiting.

1.1.9. Internal Description: The cursor optimization functions of this screen package can be used without the overhead and additional size of the screen updating functions. The screen updating functions are used where parts of the screen are changed but the overall image remains the same. Graphics programs, designed to run on character-oriented terminals find it difficult to use these functions without considerable unnecessary program overhead. A certain amount of familiarity with programming problems and some finer points of C is assumed to understand the following description of the happenings at the lower levels.

The /etc/termcap database describes a terminal's features, but a certain amount of decoding is necessary. The algorithm used is from vi. It reads the terminal capabilities from /etc/termcap in a tight loop into a set of variables whose names are two uppercase letters with some mnemonic value. For example, HO is a string which moves the cursor to the "home" position. See Appendix A for a complete list of those capabilities read and **termcap** (5) for a full description.

There are two routines to handle terminal setup in initscr(). The first, gettmode, sets some variables based upon the terminal modes accessed by gtty and stty (see ioctl(2)). The second, setterm(), reads in the descriptions from the /etc/termcap database. The following example shows how these routines are used.

```
if (isatty(Ø))
{
     gettmode();
     if (sp=getenv("TERM"))
          setterm(sp);
}
else
     setterm(Def_term);
_puts(TI);
_puts(VS);
```

isatty() determines if file descriptor Ø is a terminal. It does a gtty on the descriptor and checks the return value. gettmode() then sets the terminal modes from a gtty call. The routine getenv() is then called to get the name of the terminal. A pointer to a string containing the terminal name is returned, which is saved in the character pointer sp, and is passed to setterm(). The routine setterm() then reads in the capabilities associated with that terminal from /etc/termcap.

If isatty() returns false, the default terminal Def term is used. The TI and VS sequences initialize the terminal by calling puts; this macro uses tputs() (see termlib (3)) to put out a string. The routine endwin() undoes the previous operations.

The most difficult thing to do properly is motion optimization. When considering how many different features various terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, ...) it can be a decidedly non-trivial task to decide how to get from here to there. The editor **vi** uses many of these features and the routines it uses take up many pages of code. Fortunately, these routines are available here.

After using gettmode() and setterm() to get the terminal descriptions, the function mvcur() deals with this task. Its usage is simple: tell it where it is now and where to go. For example:

```
mvcur(Ø, Ø, LINES/2, COLS/2)
```

moves the cursor from the home position (Ø, Ø) to the middle of the screen. To force absolute addressing, use the function tgoto() from the termlib(3) routines or notify mvcur() that the cursor is positioned elsewhere. For example, to absolutely address the lower left hand corner of the screen from anywhere, just claim to be in the upper right hand corner:

```
mvcur(Ø,    COLS-1,  LINES-1,  Ø)
```

SECTION 2
CURSES FUNCTIONS

In the following definitions, **"[*]"** means that the function is really a **"#define"** macro with arguments (found in /usr/include/curses.h). This means it does not show up in stack traces in the debugger or, in the case of such functions as addch(), it shows up as its **"w"** counterpart. The arguments are given to show the order and type.

```
addch(ch) [*]
char ch;

waddch(win, ch)
WINDOW *win;
char ch;
```

adds the character ch on the window at the current (y, x) coordinates. If the character is a newline ('\n') and newline mapping is on, the line is cleared to the end and the current (y, x) coordinate is changed to the beginning of the next line. If newline mapping is off, the line is cleared to the end and the coordinate is changed. A return ('\r') moves to the beginning of the line on the window. Tabs ('\t') are expanded into spaces in the normal tabstop positions of every eight characters. This returns ERR if it would cause the screen to scroll illegally.

```
addstr(str) [*]
char *str;

waddrstr(win, str)
WINDOW *win;
char *str;
```

adds the string, str, on the window at the current (y, x) coordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, addstr puts on as much as it can.

```
box(win, vert, hor)
WINDOW *win;
char vert, how;
```

draws a box around the window using vert as the character

for drawing the vertical sides, and hor for drawing the horizontal lines. If scrolling is not allowed and the window encompasses the lower right hand corner of the terminal, the corners are left blank to avoid a scroll.

clear() [*]

wclear(win)
WINDOW *win;

resets the entire window to blanks. If win is a screen, this sets the clear flag which causes a clear-screen sequence to be sent on the next refresh call. This also moves the current (y, x) coordinates to (Ø, Ø).

clearok(scr, boolf) [*]
WINDOW *scr;
bool boolf;

sets the clear flag for the screen scr. If boolf is TRUE, this forces a clear-screen to be printed on the next refresh, or stops it from doing so if boolf is FALSE. This only works on screens, and, unlike clear, does not alter the contents of the screen. If scr is curscr, the next refresh call causes a clear-screen even if the window passed to refresh is not a screen.

clrtobot() [*]

wclrtobot(win)
WINDOW *win;

clears the window from the current (y, x) coordinates to the bottom. This does not force a clear-screen sequence on the next refresh. There is no associated "mv" command.

clrtoeol() [*]

wclrtoeol(win)
WINDOW *win;

clears the window from the current (y, x) coordinates to the end of the line. There is no associated "mv" command.

crmode() [*]

nocrmode() [*]

sets or unsets the terminal to/from cbreak mode.

delch()

wdelch(win)
WINDOW *win;

deletes the character at the current (y, x) coordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

deleteln()

wdeleteln(win)
WINDOW *win;

deletes the current line. Every line below the current one will move up, and the bottom line becomes blank. The current (y, x) coordinates remain unchanged.

delwin(win)
WINDOW *win;

deletes the window from existence. All resources are freed for future use by calloc (see malloc(3)). If a window has a subwin() allocated window inside it, and the outer window is deleted, the subwindow is not affected even though this does invalidate it. Therefore, subwindows must be deleted before their outer windows are.

echo() [*]

noecho() [*]

sets the terminal to echo or not echo characters.

endwin()

finishes up window routines before exits and restores the terminal to the state it was before initscr() (or gettmode() and setterm()) was called. It should always be called before exiting. This is especially useful for resetting

terminal status when trapping rubouts via signal(2).

erase() [*]

werase(win)
WINDOW *win;

erases the window to blanks without setting the clear flag. This is analogous to clear(), except that it never causes a clear-screen sequence to be generated on a refresh(). There is no associated "mv" command.

getch() [*]

wgetch(win)
WINDOW *win;

gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character is returned. If noecho is set, the window is left unaltered. In order to retain control of the terminal, it is necessary to have noecho, cbreak, or rawmode set. If not, whatever routine called to read characters sets cbreak mode and resets to the original mode when finished.

getstr(str) [*]
char *str;

wgetstr(win, str)
WINDOW *win;
char *str;

gets a string from the window and put it in the location pointed to by str. It sets terminal modes if necessary, and calls getch (or wgetch(win)) to get the characters needed to fill in the string until a newline or EOF is encountered. The newline is stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

gettmode()

gets the terminal modes. This is normally called by initscr.

getyx(win, y, x) [*]

```
WINDOW      *win;
int         y, x;
```

puts the current (y, x) coordinates of win in the variables y and x. Since it is a macro, not a function, the address of y and x is not passed.

inch() [*]

winch(win) [*]
WINDOW *win;

returns the character at the current (y, x) coordinates in the given window. This does not make any changes to the window. There is no associated "mv" command.

initscr()

initializes the screen routines. This must be called before any screen routines are used. It initializes the terminal-type data and without it, none of the routines can operate. If standard input is not a terminal, it sets the specifications to the terminal whose name is pointed to by Def term (initially "dumb"). If the Boolean My term is true, Def term is always used.

insch(c)
char c;

winsch(win, c)
WINDOW *win;

inserts c at the current (y, x) coordinates. Each character is shifted to the right and the last character disappears. This returns ERR if it would cause the screen to scroll illegally.

insertln()

winsertln(win)
WINDOW *win;

inserts a line above the current one. Every line below the current line is shifted down, and the bottom line disappears. The current line becomes blank, and the current (y, x) coordinates remain unchanged. This returns ERR if it would cause the screen to scroll illegally.

```
leaveok(win, boolf) [*]
WINDOW *win;
bool boolf:
```

sets the Boolean flag for leaving the cursor after the last change. If boolfdf is TRUE, the cursor is left after the last update on the terminal, and the current (y, x) coordinates for win are changed accordingly. If FALSE, the cursor is moved to the current (y, x) coordinates. This flag (initially FALSE) retains it value until changed by the user.

```
longname(termbuf, name)
char *termbuf, *name;
```

fills in name with the full name of the terminal described by the termcap entry in termbuf. This is available in the global variable ttytype. Termbuf is usually set via the termlib routine tgetent().

```
move(y, x) [*]
int y, x;

wmove(win, y, x)
WINDOW  *win;
int y, x;
```

changes the current (y, x) coordinates of the window to (y, x). This returns ERR if it would cause the screen to scroll illegally.

```
mvcur(lasty, lastx, newy, newx)
int lasty, lastx, newy, newx;
```

moves the terminal's cursor f from (lasty, lastx) to (newy, newx) in an approximation of optimal fashion. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. Move and refresh should be used to move the cursor position, so that the routines are aware of the movement. This routine uses the functions borrowed from the ex editor.

```
mvwin(win, y, x)
WINDOW *win;
int y, x;
```

moves the home position of the window win from its current

starting coordinates to (y, x). If that would put part or all of the window off the edge of the terminal screen, mvwin() returns ERR and does not change anything.

nl() [*]

nonl() [*]

sets or unsets the terminal to/from nl mode, i.e., start/stop the system from mapping **<RETURN>** to **<LINE-FEED>**. If the mapping is not done, refresh can do more optimization, so it is recommended, but not required, to turn it off.

overlay(win1, win2)
WINDOW *win1, *win2;

overlays win1 on win2. The contents of win1, (as much as will fit), are placed on win2 at their starting (y, x) coordinates. This is done non-destructively, i.e., blanks on win1 leave the contents of the space on win2 untouched.

overwrite(win1, win2)
WINDOW *win1, *win2;

overwrites win1 on win2. The contents of win1, (as much as will fit), are placed on win2 at their starting (y, x) coordinates. This is done destructively (blanks on win1 become blank on win2).

printw(fmt, arg1, arg2, ...)
char *fmt;

wprintw(win, fmt, arg1, arg2, ...)
WINDOW *win;
char *fmt;

performs a printf on the window starting at the current (y, x) coordinates. It uses addstr to add the string on the window. It is often advisable to use the field width options of printf to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

CURSES Zilog CURSES

WINDOW *

newwin(lines, cols, begin_y, begin_x)
int lines, cols, begin_y, begin_x;

creates a new window with lines lines and cols columns starting at position (begin_y, begin_x). If either lines or cols is Ø (zero), that dimension is set to (LINES - begin_y) or (COLS - begin_x) respectively. Thus, to get a new window of dimension LINES x COLS, use newwin(Ø, Ø, Ø, Ø).

WINDOW *

subwin(win, lines, cols, begin_y, begin_x)
subwin(win, lines, cols, begin_y, begin_x)
WINDOW *win;
int lines, cols, begin__y, begin__x;

creates a new window with lines lines and cols columns starting at position (begin_y, begin_x) in the middle of the window win. Any change made to either window in the area covered by the subwindow is made to both windows. The coordinates begin_y,begin_x are specified relative to the overall screen, not the relative (Ø, Ø) of win. If either lines or col is Ø (zero), that dimension is set to (LINES - begin_y) or (COLS - begin_x) respectively.

APPENDIX A
EXAMPLE A

The following is only a summary of the capabilities. For a full description of terminals, see termcap(5).

Capabilities from termcap are of three kinds: string valued options, numeric valued option, and Boolean options. The string valued options are the most complicated, since they can include padding information.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a **P** at the front of their comment. This is normally a number of milliseconds to pad the operation. In the current system, which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by PC)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen except when terminals have insert modes which will shift several lines). This is specified as, e.g. "12*", before the capability, to say 12 milliseconds per line. Capabilities where this makes sense have "P*" designated.

A.1. Variables Set By setterm()

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	Clear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		DOwn line sequence

Type	Name	Pad	Description
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ' '
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE
char *	LL		quick to Last Line, column Ø
char *	MA		ctrl character MAp for cmd mode
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r then eats Ø
char *	ND		Non-Destructive space
bool	OS		OverStrike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SO		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reverse
char *	TA	P	TAb (not I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		Underlining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

A.2. Variables Set By gettmode()

Type	Name	Description
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

If US and UE do not exist in the termcap entry, they are copied from SO and SE in setterm(). Names starting with **X** are reserved for unusual circumstances.

APPENDIX B
EXAMPLE B

The WINDOW structure is defined as follows:

```
    # define        WINDOW        struct        _win_st

struct _winst {
        short       _cury,_curx;
        short       _maxy,_maxx;
        short       _begy, begx;
        short       _flags;
        bool        _clear;
        bool        _leave;
        bool        _scroll;
        char        ** _y;
        short       * _firstch;
        short       *_lastch;
};
```

_cury and _curx are the current (y, x) coordinates for the window. New characters added to the screen are added at this point.

_maxy and _maxx are the maximum values allowed for (_cury, _curx).

_begy and _begx are the starting (y, x) coordinates on the terminal for the window, that is, the window's home.

Note that _cury, _curx, _maxy and _maxx are measured relative to (_begy, _begx), not the terminal's home.

_flags can have one or more of the following values "or'd" into it.

```
#define   -SUBWIN        Ø1
#define   _ENDLINE       Ø2
#define   _FULLWIN       Ø4
#define   _SCROLLWIN     Ø1Ø
#define   -STANDOUT      Ø2ØØ
```

_SUBWIN means that the window is a subwindow, which indicates to delwin() that the space for the lines is not to be freed.

_ENDLINE says that the space for the lines is not to be freed. _FULLWIN says that this window is a screen.

_SCROLLWIN indicates that the last character of this screen is at the lower right-hand corner of the terminal; that is, if a character is put there, the terminal will scroll.

_STANDOUT says that all characters added to the screen are in standout mode.

_clear tells if a clear-screen sequence is to be generated on the next refresh() call. This is only meaningful for screens. The initial clear-screen for the first refresh() call is generated by initially setting clear to be TRUE for cursor, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved.

_leave is TRUE if the current (y, x) coordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change.

_scroll is TRUE if scrolling is allowed.

_y is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the ith line.

_firstch represents the first character position in a line to be changed during a refresh(). This position is stored in

```
_firstch[i]
```

for the ith line.

_lastch represents the last character position in a line to be changed during a refresh(). This position is stored in

```
_lastch[i]
```

for the ith line.

starting coordinates to (y, x). If that would put part or all of the window off the edge of the terminal screen, mvwin() returns ERR and does not change anything.

nl() [*]

nonl() [*]

sets or unsets the terminal to/from nl mode, i.e., start/stop the system from mapping **<RETURN>** to **<LINE-FEED>**. If the mapping is not done, refresh can do more optimization, so it is recommended, but not required, to turn it off.

```
overlay(win1, win2)
WINDOW *win1, *win2;
```

overlays win1 on win2. The contents of win1, (as much as will fit), are placed on win2 at their starting (y, x) coordinates. This is done non-destructively, i.e., blanks on win1 leave the contents of the space on win2 untouched.

```
overwrite(win1, win2)
WINDOW *win1, *win2;
```

overwrites win1 on win2. The contents of win1, (as much as will fit), are placed on win2 at their starting (y, x) coordinates. This is done destructively (blanks on win1 become blank on win2).

```
printw(fmt, arg1, arg2, ...)
char *fmt;

wprintw(win, fmt, arg1, arg2, ...)
WINDOW *win;
char *fmt;
```

performs a printf on the window starting at the current (y, x) coordinates. It uses addstr to add the string on the window. It is often advisable to use the field width options of printf to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

raw() [*]

noraw() [*]

sets or unsets the terminal to/from raw mode. This also turns off newline mapping (see nl()).

refresh() [*]

wfresh(win)
WINDOW *win;

synchronizes the terminal screen with the desired window. If the window is not a screen, only the part covered is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it updates whatever it can without causing the scroll.

savetty() [*]

resetty() [*]

savetty() saves the current terminal characteristic flags. resetty() restores the flags to what savetty() stored. These functions are performed automatically by initscr() and endwin().

scanw(fmt, arg1, arg2, ...)
char *fmt;

wscanw(win, fmt, arg1, arg2, ...)
WINDOW *win;
char *fmt;

performs a scanf from the window using fmt. It does this using consecutive getch()'s (or wgetch(win)'s). This returns ERR if it would cause the screen to scroll illegally.

scroll(win)
WINDOW *win;

scrolls the window upward one line. This is normally not used by the user.

scrollok(win, boolf) [*]

```
WINDOW *win;
bool boolf;
```

sets the scroll flag for the given window. If boolf is FALSE, scrolling is not allowed. This is its default setting.

```
setterm(name)
char *name;
```

sets the terminal characteristics to be those of the terminal named name. This is normally called by initscr().

```
standout() [*]

wstandout(win)
WINDOW *win;

standend() [*]

wstandend(win)
WINDOW *win;
```

starts and stops putting characters onto win in standout mode. The routine standout() causes any characters added to the window to be put in standout mode on the terminal (if it has that capability) and standend() stops this. The sequences SO and SE (or US and UE if they are not defined) are used (see Appendix A).

```
touchwin(win)
WINDOW *win;
```

makes it appear that every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

WINDOW *

newwin(lines, cols, begin_y, begin_x)
int lines, cols, begin_y, begin_x;

creates a new window with lines lines and cols columns starting at position (begin y, begin x). If either lines or cols is Ø (zero), that dimension is set to (LINES - begin y) or (COLS - begin x) respectively. Thus, to get a new window of dimension LINES x COLS, use newwin(Ø, Ø, Ø, Ø).

WINDOW *

subwin(win, lines, cols, begin_y, begin_x)
subwin(win, lines, cols, begin_y, begin_x)
WINDOW *win;
int lines, cols, begin__y, begin__x;

creates a new window with lines lines and cols columns starting at position (begin y, begin_x) in the middle of the window win. Any change made to either window in the area covered by the subwindow is made to both windows. The coordinates begin y,begin_x are specified relative to the overall screen, not the relative (Ø, Ø) of win. If either lines or col is Ø (zero), that dimension is set to (LINES - begin_y) or (COLS - begin_x) respectively.

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 1-1

SECTION 2 LEX SOURCE 2-1

SECTION 3 LEX REGULAR EXPRESSIONS 3-1

3.1. Introduction 3-1
3.2. Operators 3-1
3.3. Character Classes 3-2
3.4. Arbitrary Character 3-3
3.5. Optional Expressions 3-3
3.6. Repeated Expressions 3-3
3.7. Alternation and Grouping 3-4
3.8. Context Recognition 3-4
3.9. Repetitions and Definitions 3-5
3.10. Segment Separator 3-5

SECTION 4 LEX ACTIONS 4-1

4.1. Introduction 4-1
4.2. Regular Routines 4-1
4.3. Input/Output Routines 4-4
4.4. Library Routines 4-4

SECTION 5 AMBIGUOUS SOURCE RULES 5-1

SECTION 6 LEX SOURCE DEFINITIONS 6-1

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 regularexpressions (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-zØ-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,

```
[-+Ø-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-zØ-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 cdk. 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 Ø. 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 Ø 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 integers, it is taken as an identifier, because [a-z]+ matches eight characters while integer matches only seven. If the input is integer, both rules match seven characters, and the keyword rule is selected because it is given first. Anything shorter (such as int) does not match the expression integer, 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][a-z]     {digram[yytext[Ø]][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                       [Ø-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

```
[Ø-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 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;
 [Ø-9]+ {
        sscanf(yytext, "%d", &k);
        if (k%7 == Ø)
            printf("%d", k+3);
        else
            printf("%d",k);
        }
```

The rule [Ø-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;
-?[Ø-9]+                  {
                          sscanf(yytext, "%d", &k);
                          printf("%d", k%7 == Ø ? k+3 : k);
                          }
-?[Ø-9.]+                 ECHO;
[A-Za-z][A-Za-zØ-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 =  Ø ; 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 Ø;
```

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 Ø;}
<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*	Ø,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.

Table of Contents

SECTION 1 INTRODUCTION 1-1

SECTION 2 LEX SOURCE 2-1

SECTION 3 LEX REGULAR EXPRESSIONS 3-1

3.1. Introduction 3-1
3.2. Operators 3-1
3.3. Character Classes 3-2
3.4. Arbitrary Character 3-3
3.5. Optional Expressions 3-3
3.6. Repeated Expressions 3-3
3.7. Alternation and Grouping 3-4
3.8. Context Recognition 3-4
3.9. Repetitions and Definitions 3-5
3.10. Segment Separator 3-5

SECTION 4 LEX ACTIONS 4-1

4.1. Introduction 4-1
4.2. Regular Routines 4-1
4.3. Input/Output Routines 4-4
4.4. Library Routines 4-4

SECTION 5 AMBIGUOUS SOURCE RULES 5-1

SECTION 6 LEX SOURCE DEFINITIONS 6-1

SECTION 7 COMPILING LEX 7-1

SECTION 8 LEX AND YACC 8-1

SECTION 9 EXAMPLES 9-1

9.1. Copy with Simple Arithmetic Changes 9-1
9.2. Statistical Accumulations 9-1

SECTION 10 LEFT CONTEXT SENSITIVITY 10-1

SECTION 11 CHARACTER SET 11-1

SECTION 12 SUMMARY OF SOURCE FORMAT 12-1

SECTION 13 CAUTIONS 13-1

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 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.

LINT -- A C PROGRAM CHECKER *

* This information is based on an article originally written by S.C. Johnson, Bell Laboratories.

Table of Contents

SECTION 1 GENERAL 1-1

1.1. A Word About Philosophy 1-1
1.2. Unused Variables and Functions 1-2
1.3. Set/Used Information 1-3
1.4. Flow of Control 1-3
1.5. Function Values 1-4
1.6. Type Checking 1-5
1.7. Type Casts 1-6
1.8. Nonportable Character Use 1-6
1.9. Assignments of longs to ints 1-7
1.10. Strange Constructions 1-7
1.11. Ancient History 1-8
1.12. Pointer Alignment 1-9
1.13. Multiple Uses and Side Effects 1-9
1.14. Implementation 1-10
1.15. Portability 1-11
1.16. Shutting Lint Up 1-12
1.17. Library Declaration Files 1-14
1.18. Bugs, etc. 1-14

APPENDIX A CURRENT LINT OPTIONS A-1

SECTION 1
GENERAL

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 by will produce messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying 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.

1.1. 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.

1.2. 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 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 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 option is available to suppress the printing of complaints about unused arguments. When 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 flag may be used to suppress the spurious messages which might otherwise appear.

1.3. 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.

1.4. 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 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 statements. The 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 option.

1.5. 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!

1.6. 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.

1.7. 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 flag controls the printing of comments about casts. When 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.

1.8. 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 cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned.**

1.9. 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.

1.10. 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&Ø77 == Ø ) ...
```

or

```
x<<2 + 4Ø
```

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.

1.11. 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.

1.12. 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.

1.13. 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
```

1.14. 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.

1.15. 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.

1.16. 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.

1.17. 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 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 flag can be used to suppress all library checking.

1.18. 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 A
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 statements.

x Report unused external declarations

a Report assignments of to or shorter.

c Complain about questionable casts

n No library checking is done

s Same as (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 1-1

1.1. Using Make 1-1

SECTION 2 BASIC FEATURES 2-1

2.1. Program Operation 2-1
2.2. Programming Example 2-1
2.3. File Generation and Macro Substitution 2-2
2.4. Description Files 2-3

SECTION 3 COMMAND USAGE 3-1

3.1. Arguments 3-1
3.2. Implicit Rules 3-2
3.3. Suffixes and Transformation Rules 3-3
3.4. Sample Program 3-4
3.5. Suggestions and Warnings 3-6

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.

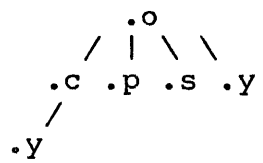
The remaining arguments are assumed to be the names of targets to be made; they are done in left-to-right order. If there are no such arguments, the first name in the description files that does not begin with a period is made.

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.

The M4 MACRO PROCESSOR*

* This information is based on an article originally written by Brian W. Kernighan and Dennis M. Ritchie.

Table of Contents

SECTION 1 The M4 Macro Processor 1-1

1.1. Introduction 1-1
1.2. Usage 1-2
1.3. Defining Macros 1-2
1.4. Quoting 1-4
1.5. Arguments 1-5
1.6. Arithmetic Built-ins 1-7
1.7. File Manipulation 1-8
1.8. System Command 1-9
1.9. Conditionals 1-9
1.10. String Manipulation 1-10
1.11. Printing 1-11
1.12. Summary of Built-ins 1-11
1.13. Acknowledgements 1-12
1.14. References 1-12

SECTION 1
The M4 Macro Processor

M4 is a macro processor available on ZEUS. Its primary use has been as a front end for Ratfor for those cases where parameterless macros are not adequately powerful. It has also been used for languages as disparate as C and Cobol. M4 is particularly suited for functional languages like Fortran, PL/I and C since macros are specified in a functional notation.

M4 provides features seldom found even in much larger macro processors, including

- arguments
- condition testing
- arithmetic capabilities
- string and substring functions
- file manipulation

This paper is a user's manual for M4.

1.1. Introduction

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The #define statement in C and the analogous define in Ratfor are examples of the basic facility provided by any macro processor - replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1]. Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string

processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is rescanned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

1.2. Usage

On ZEUS use

```
m4 [files]
```

Each argument file is processed in order; if there are no arguments, or if an argument is `-', the standard input is read at that point. The processed text is written on the standard output, which may be captured for subsequent processing with

```
m4 [files] >outputfile
```

1.3. Defining Macros

The primary built-in function of M4 is define, which is used to define new macros. The input

```
define(name, stuff)
```

causes the string name to be defined as **stuff**. All subsequent occurrences of **name** will be replaced by **stuff**. **name** must be alphanumeric and must begin with a letter (the underscore _ counts as a letter). **stuff** is any text that contains balanced parentheses; it can stretch over multiple lines.

Thus, as a typical example,

```
define(N, 100)
...
if (i > N)
```

defines **N** to be 100, and uses this "symbolic constant" in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by `(', it is assumed to have no arguments. This is the situation for **N** above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
...
if (NNN > 100)
```

the variable **NNN** is absolutely unrelated to the defined macro **N**, even though it contains a lot of **N's**.

Things may be defined in terms of other things. For example,

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if **N** is redefined? Or, to say it another way, is **M** defined as **N** or as 100? In M4, the latter is true - **M** is 100, so even if **N** subsequently changes, **M** does not.

This behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string **N** is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is

to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now M is defined to be the string **N**, so when you ask for **M** later, you'll always get the value of **N** at that time (because the M will be replaced by **N** which will be replaced by 100).

1.4. Quoting

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by the single quotes ` and ' is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N')
```

the quotes around the **N** are stripped off as the argument is being collected, but they have served their purpose, and **M** is defined as the string **N**, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word **define** to appear in the output, you have to quote it in the input, as in

```
`define' = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining **N**:

```
define(N, 100)
  ...
define(N, 200)
```

Perhaps regrettably, the **N** in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine **N**, you must delay the evaluation by quoting:

```
define(N, 100)
  ...
define(`N', 200)
```

In M4, it is often wise to quote the first argument of a macro.

If ` and ' are not convenient for some reason, the quote characters can be changed with the built-in **changequote:**

```
changequote([, ])
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to **define. undefine** removes the definition of some macro or built-in:

```
undefine(`N')
```

removes the definition of **N.** (Why are the quotes absolutely necessary?) Built-ins can be removed with **undefine,** as in

```
undefine(`define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has pre-defined the names **unix** and **gcos** on the corresponding systems, so you can tell which one you're using:

```
ifdef(`unix', `define(wordsize,16)' )
ifdef(`gcos', `define(wordsize,36)' )
```

makes a definition appropriate for the particular machine. Don't forget the quotes!

ifdef actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef(`unix', on UNIX, not on UNIX)
```

1.5. Arguments

So far we have discussed the simplest form of macro processing - replacing one string by another (fixed) string.

User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$n** will be replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump**, defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is

```
x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through **$1** to **$9**. (The macro name itself is **$0**, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro **cat** which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

$4 through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

```
define(a,   b   c)
```

defines **a** to be **b c**.

Arguments are separated by commas, but parentheses are counted properly, so a comma "protected" by parentheses does not terminate an argument. That is, in

```
define(a, (b,c))
```

there are only two arguments; the second is literally **(b,c)**. And of course a bare comma or parenthesis can be inserted by quoting it.

1.6. Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as "one more than N", write

```
define(N, 100)
define(N1, `incr(N)')
```

Then **N1** is defined as one more than the current value of **N**.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the operators (in decreasing order of precedence)

```
unary + and -
** or ^ (exponentiation)
*  /  % (modulus)
+  -
==  !=  <  <=  >  >=
!                  (not)
& or && (logical and)
|or||   (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is 32 bits on ZEUS.

As a simple example, suppose we want **M** to be **2******N+1**. Then

```
define(N, 3)
define(M, `eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

1.7. File Manipulation

You can include a new file in the input at any time by the built-in function **include:**

```
include(filename)
```

inserts the contents of **filename** in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** ("silent include") says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as **n.** Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(Ø)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between Ø and 9 inclusive.

The value of **undivert** is <u>not</u> the diverted stuff. Furthermore, the diverted material is <u>not</u> rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

1.8. System Command

You can run any program in the local operating system with the **syscmd** built-in. For example,

```
syscmd(date)
```

on ZEUS runs the **date** command. Normally **syscmd** would be used to create a file for a subsequent **include.**

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function mktemp: a string of XXXXX in the argument is replaced by the process id of the current process.

1.9. Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

```
ifelse(a, b, c, d)
```

compares the two strings **a** and **b.** If these are identical, **ifelse** returns the string **c;** otherwise it returns **d.** Thus we might define a macro called **compare** which compares two strings and returns "yes" or "no" if they are the same or different.

```
define(compare, `ifelse($1, $2, yes, no)')
```

Note the quotes, which prevent too-early evaluation of **ifelse.**

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

```
ifelse(a, b, c, d, e, f, g)
```

if the string **a** matches the string **b,** the result is **c.** Otherwise, if **d** is the same as **e,** the result is **f.** Otherwise the result is **g.** If the final argument is omitted, the result is null, so

```
ifelse(a, b, c)
```

is **c** if **a** matches **b,** and null otherwise.

1.1Ø. String Manipulation

The built-in **len** returns the length of the string that makes up its argument. Thus

```
len(abcdef)
```

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. **substr(s, i, n)** returns the substring of **s** that starts at the **i**th position (origin zero), and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

```
substr(`now is the time', 1)
```

is

```
ow is the time
```

If **i** or **n** are out of range, various sensible things happen.

index(s1, s2) returns the index (position) in **s1** where the string **s2** occurs, or -1 if it doesn't occur. As with **substr**, the origin for strings is Ø.

The built-in **translit** performs character transliteration.

```
translit(s, f, t)
```

modifies **s** by replacing any character found in **f** by the corresponding character of **t**. That is,

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters which don't have an entry in **t** are deleted; as a limiting case, if **t** is not present at all, characters from **f** are deleted from **s**. So

```
translit(s, aeiou)
```

deletes vowels from **s**.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, due to J. E. Weythman, is

```
divert(-1)
        define(...)
        ...
divert
```

1.11. Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus you can say

```
errprint(`fatal error')
```

dumpdef is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

1.12. Summary of Built-ins

Each entry is preceded by the page number where it is described.

```
3  changequote(L, R)
1  define(name, replacement)
4  divert(number)
4  divnum
5  dnl
5  dumpdef(`name', `name', ...)
5  errprint(s, s, ...)
4  eval(numeric expression)
3  ifdef(`name', this if true, this if false)
5  ifelse(a, b, c, d)
4  include(file)
3  incr(number)
5  index(s1, s2)
5  len(string)
4  maketemp(...XXXXX...)
4  sinclude(file)
5  substr(string, position, number)
4  syscmd(s)
5  translit(str, from, to)
3  undefine(`name')
4  undivert(number,number,...)
```

1.13. Acknowledgements

We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially Jim Weythman, whose pioneering use of M4 has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code.

1.14. References

[1] B. W. Kernighan and P. J. Plauger, Software Tools, Addison-Wesley, Inc., 1976.

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 1-1

1.1. Program Arguments 1-1
1.2. The Standard Input and Output 1-1

SECTION 2 The Standard I/O Library 2-1

2.1. Introduction 2-1
2.2. File Access 2-1
2.3. Error Handling 2-4
2.4. Miscellaneous I/O Functions 2-5
2.5. General Usage 2-5
2.6. Calls 2-6
2.7. Macros 2-16

SECTION 3 LOW-LEVEL I/O 3-1

3.1. General 3-1
3.2. File Descriptors 3-1
3.3. Read and Write 3-2
3.4. Open, Creat, Close, Unlink 3-4
3.5. Random Access With lseek 3-6
3.6. Error Processing 3-7

SECTION 4 PROCESSES 4-1

4.1. System Function 4-1
4.2. Low-Level Process Creation 4-1
4.3. Control of Processes 4-2
4.4. Pipes 4-4

SECTION 5 SIGNALS 5-1

5.1. General 5-1
5.2. Signal Routine 5-1
5.3. Interrupts 5-2

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[Ø] is the command name itself, so argc is always greater than Ø. 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 \Ø, 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 < Ø177) || c == '\t' || c == '\n')
      putchar(c);
   exit(Ø);
}
```

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 Ø) 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 standalone 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 ? " %sØ\n" : "Ø\n", argv[i]);
            fclose(fp);
            tlinect += linect;
            twordct += wordct;
            tcharct += charct;
        } while (++i < argc);
        if (argc > 2)
        printf("%71d %71d %71d totalØ\n", tlinect, twordct, tchar
        exit(Ø);
}
```

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 Ø 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 \Ø 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) charc; 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) charc; 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) charc;

This call is identical to putc(c, stdout).

char *fgets(s, n, ioptr) char *s; intn; 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) charc; 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, al, ...) char *format;
fprintf(ioptr, format, al, ...) FILE *ioptr; char *format;
sprintf(s, format, al, ...)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, al, ...) char *format;
fscanf(ioptr, format, al, ...) FILE *ioptr; char *format;
sscanf(s, format, al, ...) 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 Ø, 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; intnitems; 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; intnitems; 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 8ØØØ uses 16-bit words.)

putw(w, ioptr) intw; 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; intptrname;

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) intuid; 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); intnum;

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); intnum, size;

This call allocates space for num items, each of size size. The space is guaranteed to be set to Ø 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 Ø, 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 Ø 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 Ø 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 Ø 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 anything, 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)) > Ø)
                write(stdout, buf, n);
        exit(Ø);
}
```

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   Ø377    /* for making char's > Ø */
getchar()       /* unbuffered single character input */
{
        char c;
        return((read(Ø, &c, 1) > Ø) ? c & CMASK : EOF);
}
```

The variable c must be declared char, because read accepts a character pointer. The character being returned must be masked with Ø377 (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, Ø755 (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 Ø
#define BUFSIZE 512
#define PMODE Ø644 /* 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], Ø)) == -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)) > Ø)
                if (write(f2, buf, n) != n)
                        error("cp: write error", NULL);
        exit(Ø);
}

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 command-line 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), Oproc 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() == Ø)
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() == Ø)
        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 Ø 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 (Ø, 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[Ø] 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 Ø, 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 Ø. 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 occured 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("ØnterruptØ);
        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);
}
```

ZEUS PLZ/ASM ASSMEBLER

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. A description of its exact invocation is contained in the ZEUS Reference Manual (as(1).

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 1-1

1.1. General Description 1-1
1.2. Relocatability 1-1
1.3. Assembler Abort Conditions 1-1
1.4. User Input 1-1
1.5. Assembler Output 1-2
1.6. Command Line 1-2
1.7. Options 1-2

SECTION 2 LISTING FORMAT 2-1

2.1. Format Description 2-1
2.2. Sample Listing 2-2

SECTION 3 MINIMAL PROGRAM REQUIREMENTS 3-1

SECTION 4 IMPLEMENTATION FEATURES AND LIMITATIONS 4-1

SECTION 5 OBJECT CODE 5-1

SECTION 6 PLZ/ASM ERROR MESSAGES 6-1

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.

1.4. 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 1.6.

1.5. 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 5). 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 6 for explanations of error messages.

1.6. 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.

1.7. 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 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 5).

-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 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 5).
-^	Turns on the pass1 trace facility.

SECTION 2
LISTING FORMAT

2.1. 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 2.2.

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 1.6).

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.

2.2. 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 execut-
able 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  pro-
cedure !
                   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 3
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 := %1Ø

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 4
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 5
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 1.6 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		
ØØ	NOP	Null operation
Ø1	SEGMODULE	Segmented module definition
Ø2	NONSEGMODULE	Nonsegmented module definition
Ø3	ENDMODULE	End module
Ø4	SECTION	Section definition
Ø5	GLOB	Global symbol definition
Ø6	ABSGLOB	Global symbol definition with absolute offset
Ø7	EXTERN	External symbol definition
Ø8	ENTRYPT	Entry point with relocatable offset
Ø9	ABSENTRYPT	Entry point with absolute offset
ØA	DEBUGSYMBOL	Debug symbol
ØB	DEBUGINFO	Debug information
ØC	MESSAGE	Variable length message
ØD	SETDATA	Set current data section
ØE	SETPROG	Set current program section
ØF	BEGSEC	Begin section
1Ø	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		
2Ø	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 6
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
	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
	PROCEDURE DECLARATION ERRORS
70	Invalid procedure declaration
71	ENTRY expected
72	Procedure name expected after END
	INITIALIZATION ERRORS
80	Invalid initial value
81	Too many initialization elements for declared variables
82	Invalid initialization
83	Array [*] given single noncharacter_sequence initializer
84	Attempt to initialize an uninitialized data area
	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

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.

ZEUS 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
- ZEUS Reference Manual, Zilog part number 03-3255 (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 1-1

SECTION 2 PLZ/SYS RUNNING UNDER ZEUS 2-1

2.1. Overview 2-1
2.2. Limitations 2-1
2.3. Run-Time Conventions 2-1

SECTION 3 PLZ/SYS COMPILER 3-1

3.1. Overview 3-1
3.2. Plzsys Command Line 3-1
3.3. PLZ/SYS Version 3.1 Features and Limitations 3-2
3.3.1. Character Conventions 3-2
3.3.2. Character Sequence and Identifier Length . 3-3
3.3.3. Source Line Length 3-3
3.3.4. Procedure, Data, and Program Size Limitations 3-3
3.3.5. Error Recovery 3-3
3.3.6. Compiler Evaluation of Constant Expressions 3-4
3.3.7. Literal Constants or Compile-Time Constant Expressions 3-4
3.3.8. Constant Type Determination 3-5
3.3.9. Structured Return Parameters 3-6

SECTION 4 CODE GENERATOR 4-1

4.1. Overview 4-1
4.2. Plzcg Command Line 4-1

SECTION 5 PLZ/SYS IMPLEMENTATION CONVENTIONS FOR THE Z8000 5-1

5.1. Overview 5-1
5.2. Data Representation 5-1
5.2.1. Primitive Data Type Representation 5-1
5.2.2. Structured Data Type Representation 5-3

5.3. Data Alignment 5-3
5.4. Data Access Methods 5-5
5.5. Run-Time Storage Administration 5-6
5.5.1. Nonsegmented Code 5-6
5.5.2. Segmented Code 5-8
5.6. Register Conventions 5-10
5.6.1. Nonsegmented Code 5-10
5.6.2. Segmented Code 5-11
5.7. Execution Preparation 5-11
5.7.1. Nonsegmented Code 5-11
5.7.2. Segmented Code 5-11

SECTION 6 PLZ/SYS - PLZ/ASM INTERFACE EXAMPLE 6-1

6.1. Purpose 6-1
6.2. Nonsegmented Code 6-3
6.3. Segmented Code 6-6

APPENDIX A PLZ/SYS ERROR MESSAGES A-1

APPENDIX B PLZCG ERROR NUMBERS AND EXPLANATIONS B-1

List of Illustrations

Figure

1-1 Linking of PLZ/SYS and PLZ/ASM Source Code 1-2

5-1 Nonsegmented Run-Time Stack--General Layout 5-7

5-2 Segmented Run-Time Stacks--General Layout 5-9

6-1 Example 2: PLZ/ASM Module for the Nonsegmented S8000 6-1

6-2 Nonsegmented Run-Time Stack Detail After Entry Sequence 6-4

6-3 Nonsegmented Run-Time Stack Detail Before Recursive Call 6-5

6-4 Example 3: PLZ/ASM Module for the Segmented S8000 6-8

6-5 Segmented Run-Time Stack Detail After Entry Sequence 6-9

6-6 Segmented Run-Time Stack Detail Before Recursive Call 6-10

List of Tables

Table
3-1 Evaluation of Constant Expressions 3-5

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.

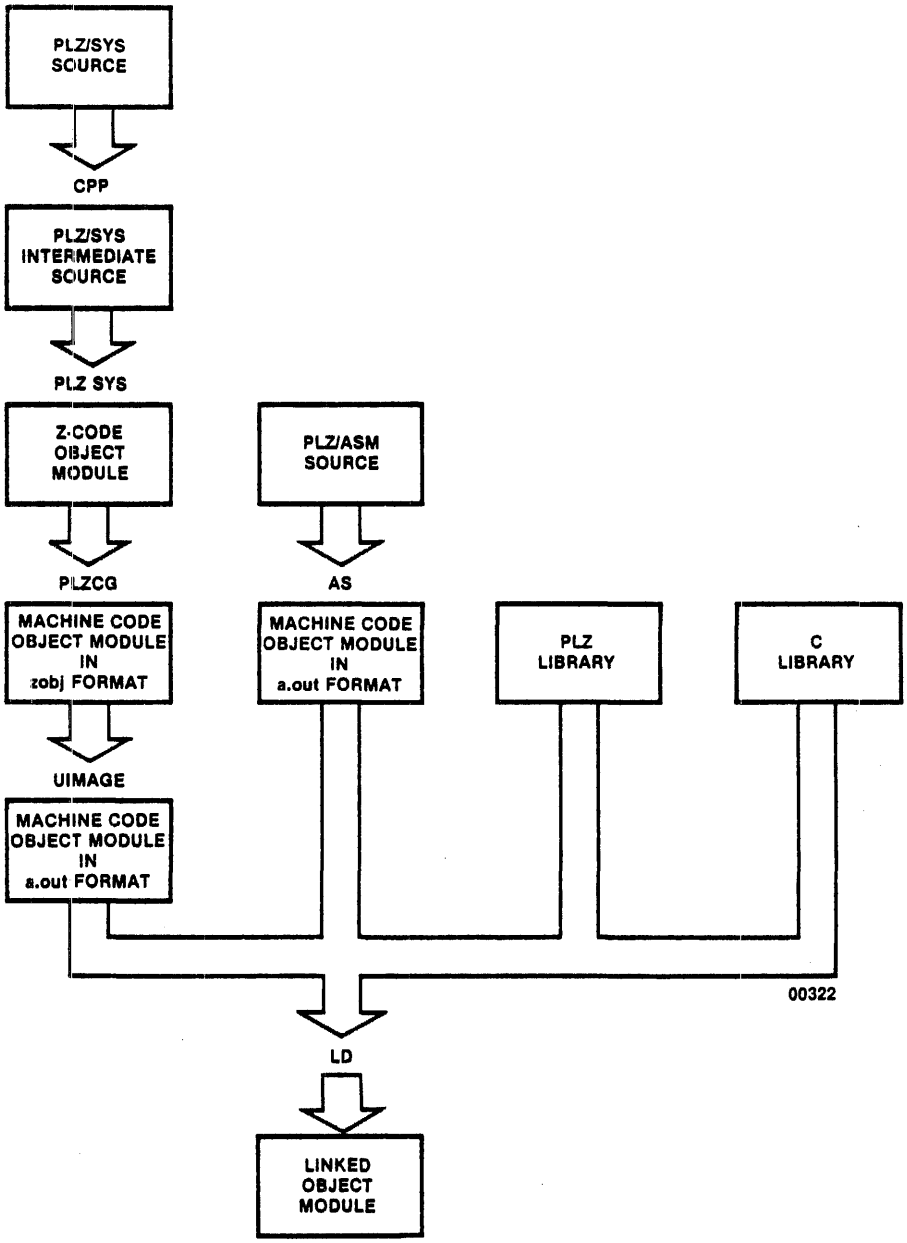


Figure 1-1 Linking of PLZ/SYS and PLZ/ASM Source Code

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 Z8Ø	Generates output suitable for the Z8Ø. Does not allow extensions to plzsys such as long variables and structure comparison and assignment. The output can run on MCZ only.
-t Z8ØØØs	Generates output suitable for the segmented Z8ØØØ. 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 Z8ØØØns	Generates output suitable for the nonsegmented Z8ØØØ. Allows long variables and structure comparison and assignment. This is the default.

3.3. PLZ/SYS Version 3.1 Features and Limitations

The following Z8ØØØ 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 12Ø 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 1ØØØ 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 Ø, 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 Z8ØØØ) cannot transport easily to other machines or translators.

3.3.7. Literal Constants or Compile-Time Constant Expressions: Error 24Ø 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 Z8ØØØ), 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 Ø 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 Ø.

Table 3-1. Evaluation of Constant Expressions

Compile-Time Constant Expression	Value	Run-Time Constant Expression	Value
L LONG := -1	%ØØØØFFFF	L := -1	%FFFFFFFF (*)
LI LONG_INTEGER := -1	%FFFFFFFF	LI := -1	%FFFFFFFF (*)
L LONG := %FFFF	%ØØØØFFFF	L := %FFFF	%ØØØØFFFF
LI LONG_INTEGER := %FFFF	%FFFFFFFF	LI := %FFFF	%ØØØØFFFF
L LONG := -%FFFF	%ØØØØØØØ1	L := -%FFFF	%FFFFØØØ1 (*)
LI LONG_INTEGER := -%FFFF	%ØØØØØØØ1	LI := -%FFFF	%FFFFØØØ1 (*)
L LONG := %AAAA*(%FFFF+1) + %BBBB	%ØØØØBBBB	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 (Ø<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>Ø instead of Ø<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 [1Ø 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
```

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] filename1

where filename1 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 Ø to 65535 (decimal) and is represented on the Z8ØØØ as an unsigned 16-bit word.

INTEGER

An INTEGER value is an integer in the range -32768 to 32767 and is represented on the Z8ØØØ as a signed 16-bit word in twos-complement notation.

LONG

A LONG value is a nonnegative integer in the range Ø to 4,294,967,295 (decimal) and is represented on the Z8ØØØ 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 Z8ØØØ as a signed 32-bit long word in twos-complement notation.

Pointer

Nonsegmented code:

A pointer value on the nonsegmented Z8ØØØ is a storage address represented as a 16-bit word. The distinguished value NIL is represented by the value zero (Ø).

Segmented code:

A pointer value on the segmented Z8ØØØ 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 Ø): 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[Ø]
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 Z8ØØØ, 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 Z8ØØØ. However, Based Addressing is available on only a restricted set of machine instructions. On the nonsegmented Z8ØØØ, Indexed mode is equivalent to Based mode, and can be used in most instructions to achieve the effect of Based Addressing. On the segmented Z8ØØØ, 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.

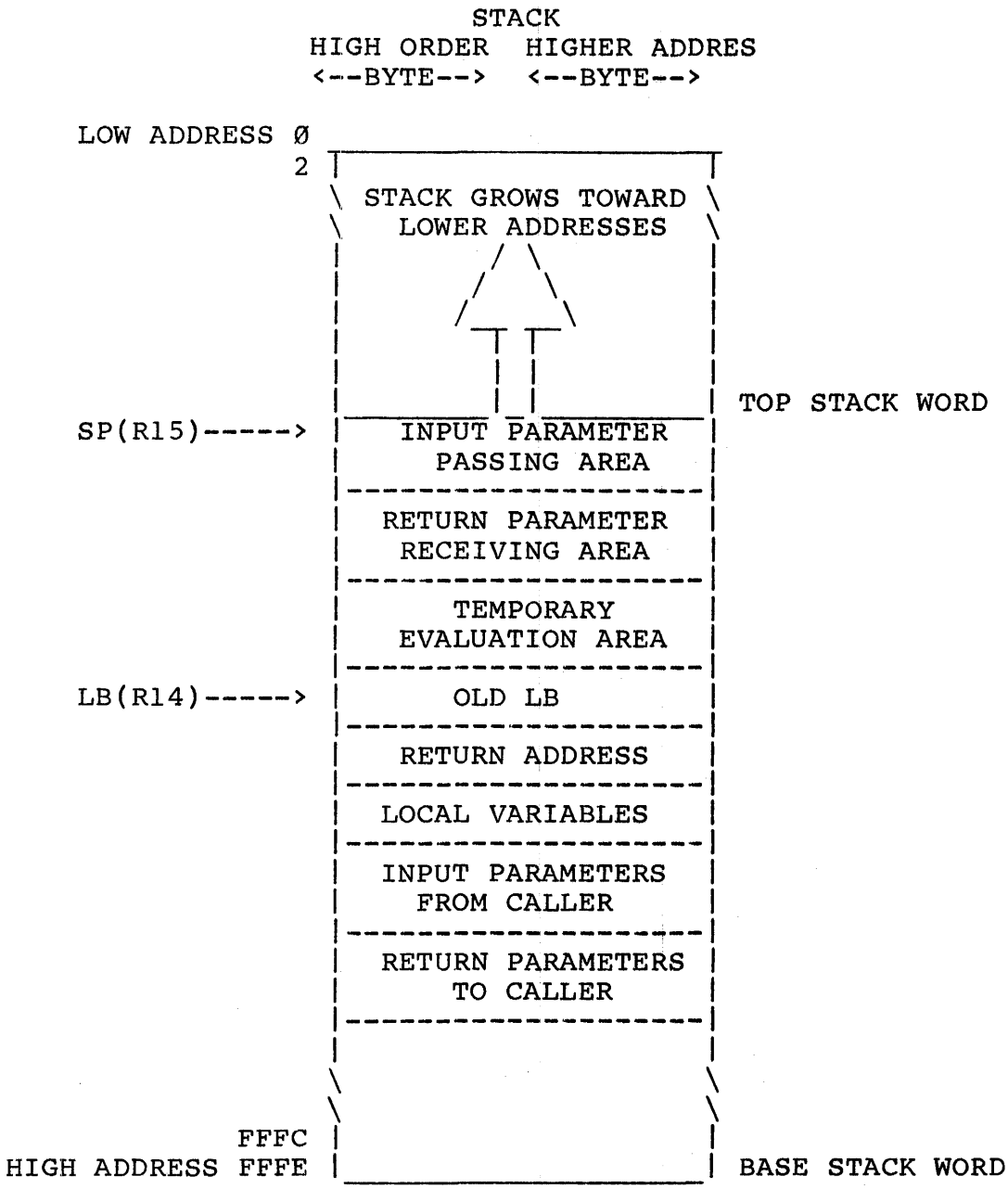


Figure 5-1 Nonsegmented Run-Time Stack--General Layout

Immediately above the input 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.

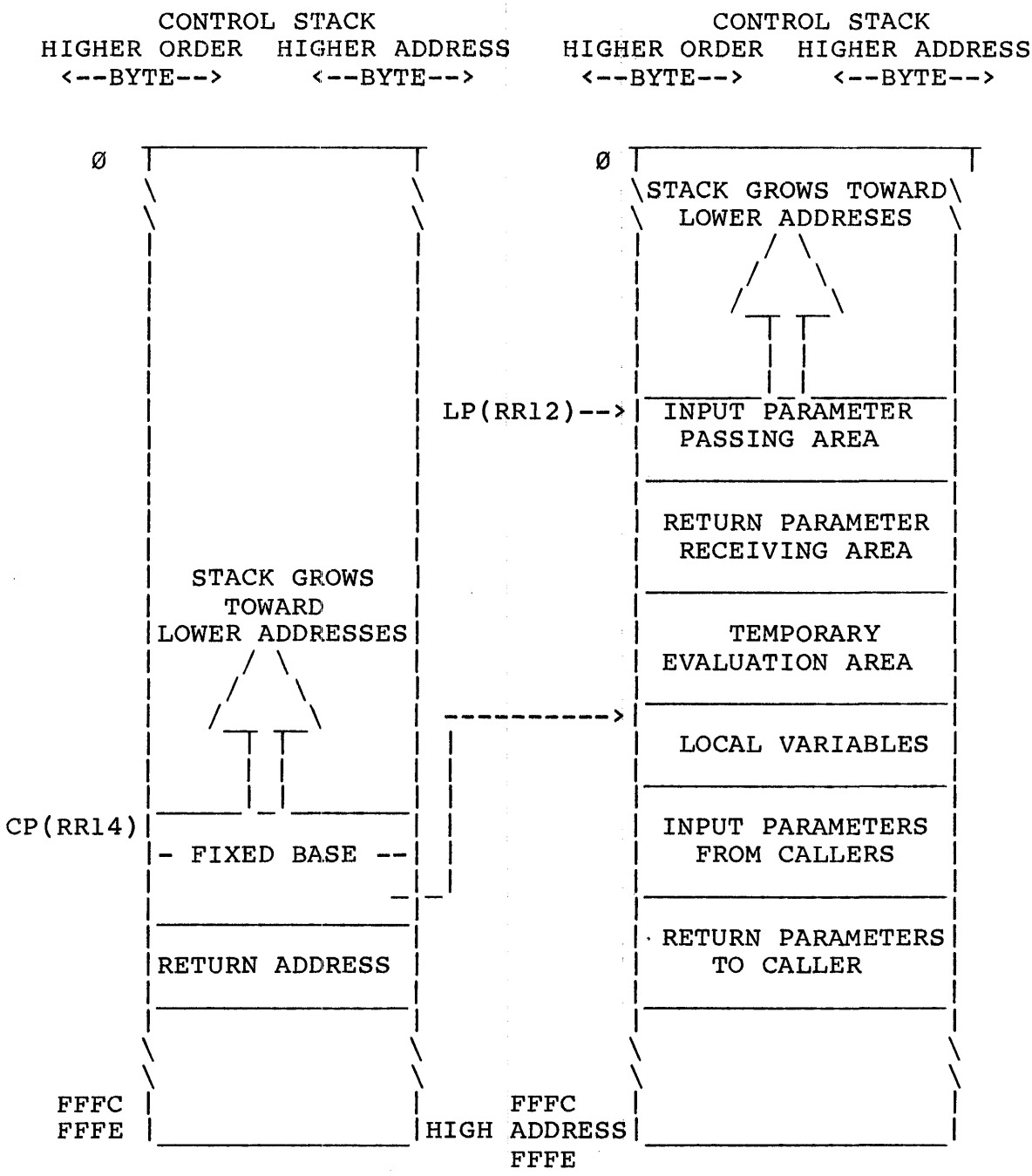


Figure 5-2 Segmented Run-Time Stacks--General Layout

The location of parameters is such that input 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)
RØ-R13	Unassigned

Procedures use registers Ø 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)
RØ-R11	Unassigned

Procedures use Registers Ø 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 Z8ØØØ, 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 8ØØØ. 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 (in1:  BYTE;  in2:  ^BYTE)
  10            RETURNS   (out1: BYTE;  out2:  WORD)
  11            LOCAL
  12                local1, local2, local3: BYTE
  13            ENTRY
  14    1           out1, 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
   Ø error
Assembly complete
```

Figure 6-1 Example 2: PLZ/ASM Module for the Nonsegmented S8ØØØ

6.2. Nonsegmented Code

An equivalent module written in PLZ/ASM for the nonsegmented Z8ØØØ 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 local1 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 in1 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.

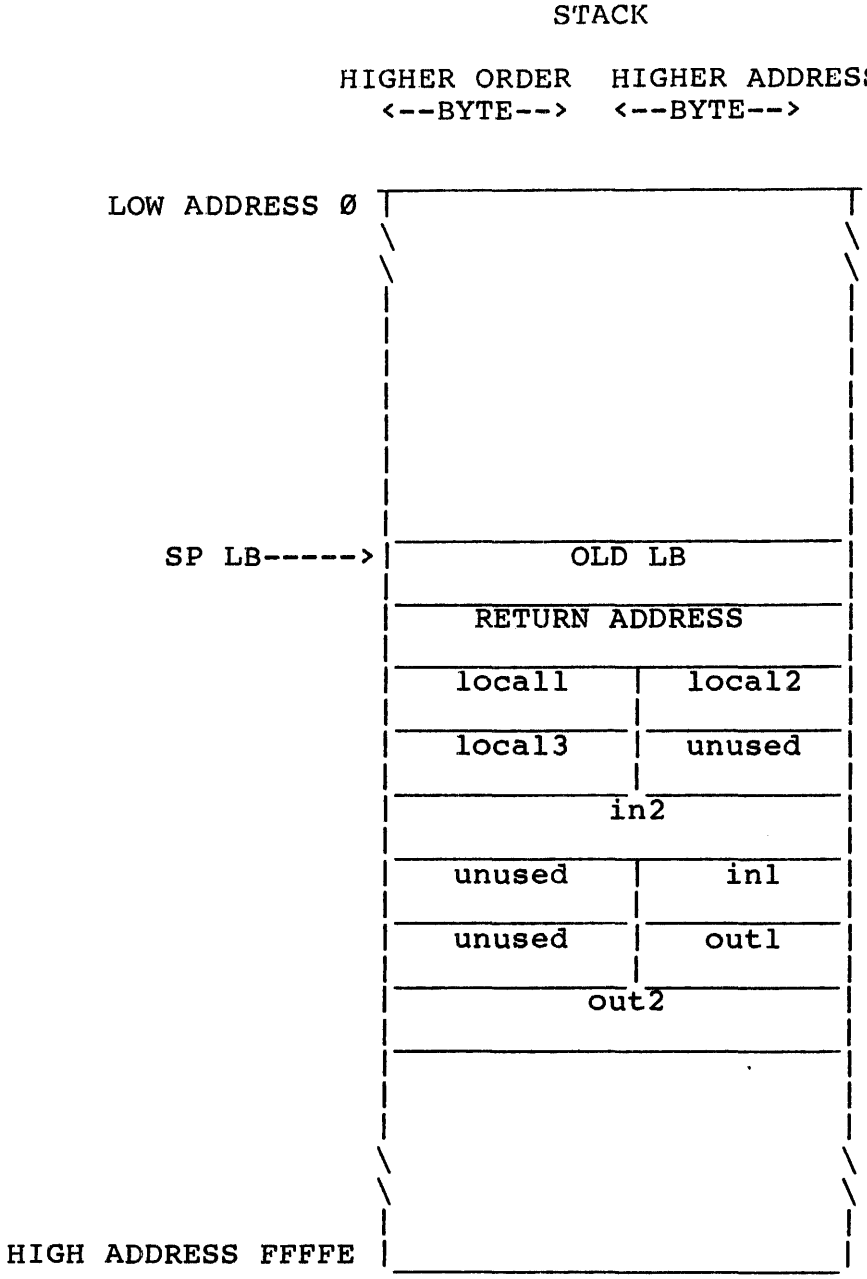


Figure 6-2 Nonsegmented Run-Time Stack Detail After Entry Sequence (Before Line 29 in Example #2)

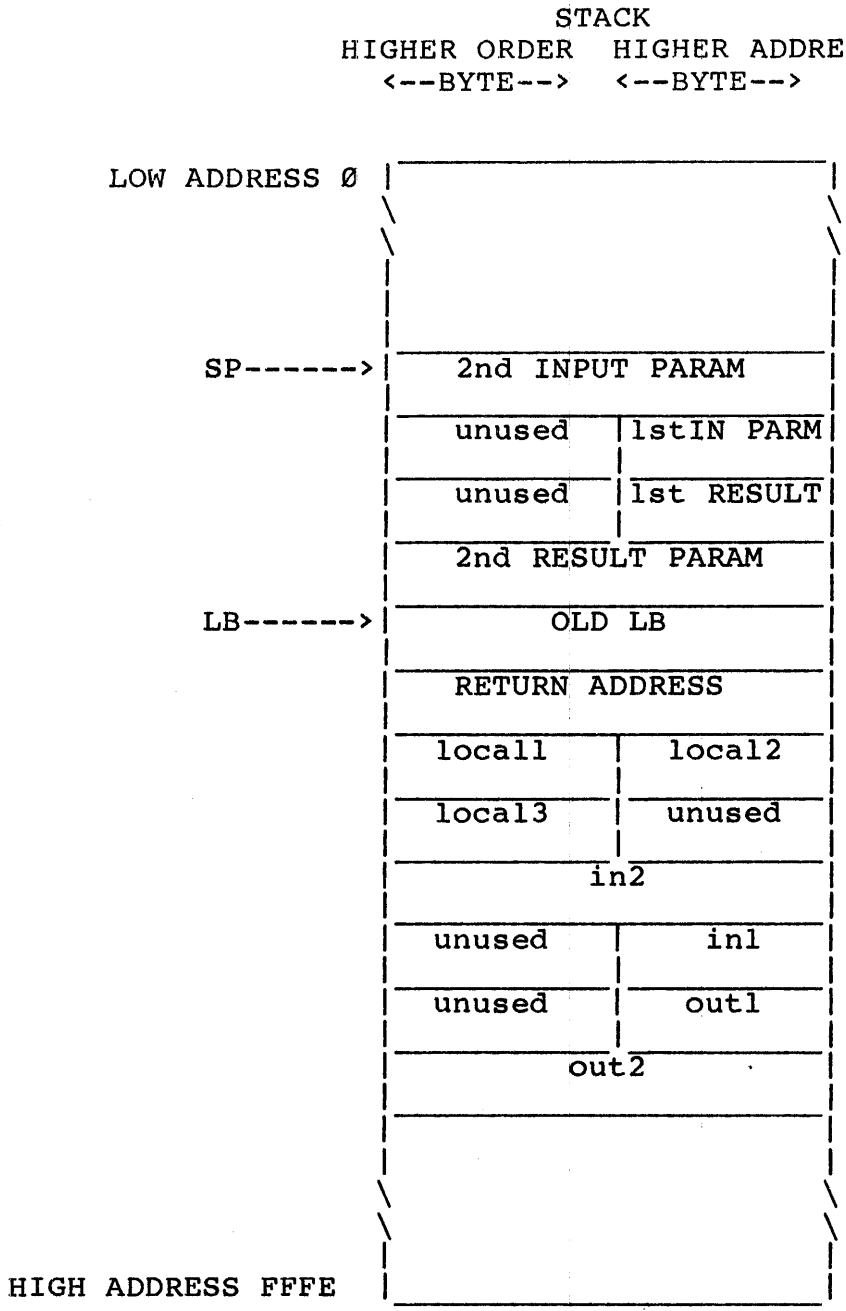


Figure 6-3 Nonsegmented Run-Time Stack Detail Before Recursive Call (Before Line 36 in Example #2)

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 resu
                   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
                5Ø    END Example
   Ø errors
Assembly complete
```

Figure 6-4 Example 3: PLZ/ASM Module for the Segmented S8ØØØ

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 3Ø 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.

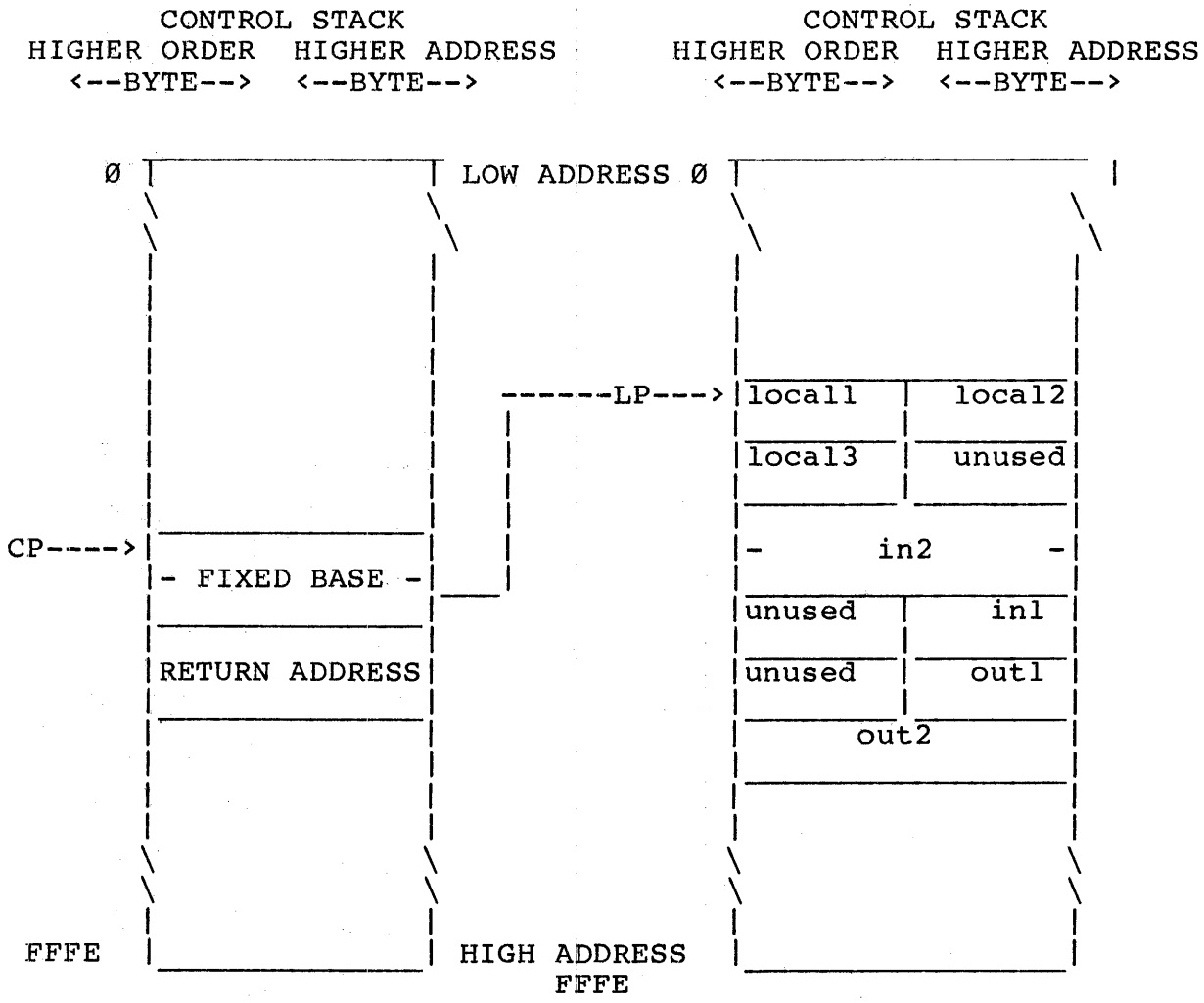


Figure 6-5 Segmented Run-Time Stack Detail After Entry Sequence (Before Line 30 in Example #3)

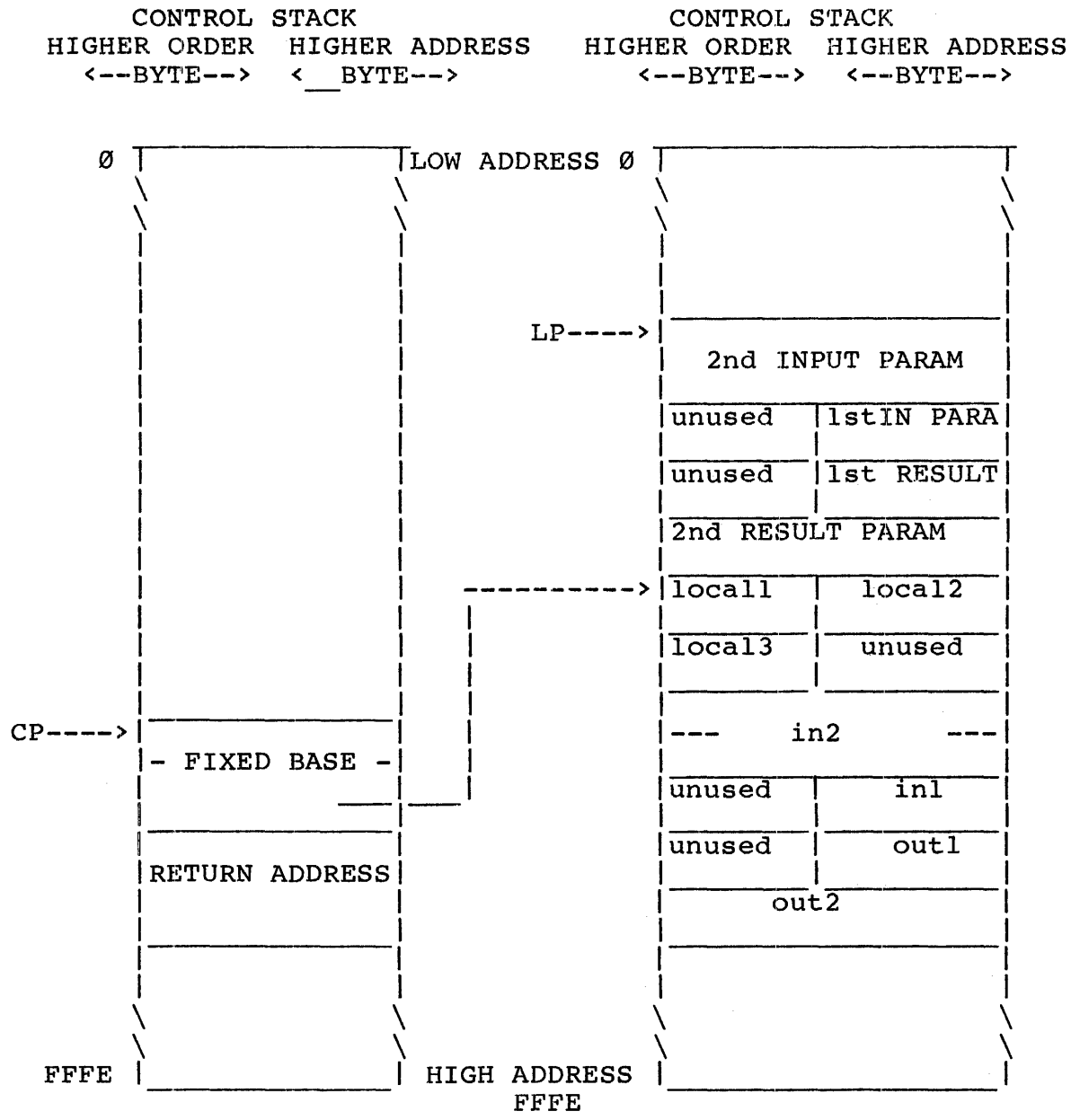


Figure 6-6 Segmented Run-Time Stack Detail Before Recursive Call (Before Line 38 in Example #3)

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 ZEUS PLZ/ASM User Guide (PLZ/ASM).

ERROR	EXPLANATION
	Warnings
Ø	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 8ØØØ
	Token Errors
1Ø	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
2Ø	Unmatched OD
21	OD expected
22	Invalid repeat statement
23	Invalid exit statement
24	Invalid FROM label
	IF Statement Errors
3Ø	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.

The Screen Interface Library

Preface

This document describes a package of C library functions which allow the user to:

(1) update a screen with reasonable optimization,

(2) get input from the terminal in a screen-oriented fashion, and

(3) independent from the above, move the cursor optimally from one point to another.

These routines all use the /etc/termcap database to describe the capabilities of the manual.

Table of Contents

SECTION 1 INTRODUCTION 1-1

SECTION 2 DESCRIPTION 2-1

SECTION 3 INVOCATION AND OPERATION 3-1

3.1. Programming 3-1
3.2. Normal Termination 3-1
3.3. Abnormal Termination 3-1

APPENDIX A SUMMARY OF LIBRARY ROUTINES A-1

APPENDIX B PROGRAMMING INFORMATION B-1

APPENDIX C USER GUIDE C-1

SECTION 1
INTRODUCTION

An effective method to convey information to a computer user is to output displays on the terminal screen. When presented with concise displays, users are able to respond quickly and easily. Visualizing the activities occurring on a computer system allows both user and machine to become more effective.

On the ZEUS system, the most commonly-used screen utility is the visual editor, vi. With vi, pages of text are displayed to allow more efficient editing and movement within a file. The flexibility of vi, the screen editor, is immediately evident when compared with the line-oriented editor ed. In addition, vi is terminal-independent; it uses /etc/termcap, the terminal capability data base, to interface to a variety of terminal types.

The Screen Interface Library is a library of functions ("routines") for developing software with this screen-oriented or display approach. The routines in the library use /etc/termcap for terminal-independency and the Curses library (/usr/lib/libcurses.a) because of its screen updating capabilities. See the Curses entry in this manual.

The Screen Interface Library includes routines that provide terminal setup, cursor manipulation using the arrow keys, single character input, highlighting of the cursored item, paging, scrolling, saving and restoring displays, obtaining the word on which the cursor lies, handling help files, and general last line handling. The individual routines are described in the ZEUS Reference Manual; the remainder of this document describes their use.

SECTION 2
DESCRIPTION

The Screen Interface Library provides the programmer with a set of routines that allow cursor and screen manipulation for terminal displays. These terminal displays are columns of information (files names or a list) which are created using the Curses routines.

The Screen Interface Library is considered to be an extension of the Curses package since the routines themselves use the Curses package. This implies that the Screen Interface Library is terminal-independent which means a call to the Curses routine, initscr(), must be made for a screen program. This routine finds the terminal type and performs terminal initialization.

Another feature of the Screen Interface Library is that the routines have adopted the window concept introduced in the Curses package. A window is defined as an internal representation containing an image of a section of the terminal screen. New windows are defined by using the following Curses call:

```
win = newwin(lines, cols, begin y, begin_x);
```

where win is a pointer to the structure WINDOW (this is defined in /usr/include/curses.h).

As in the Curses package, most Screen Interface routines have two versions - a full screen version and a window version; these are defined in /usr/include/screen.h. This means that when the full screen version of the routine is used, the WINDOW pointer, stdscr, is assumed. For example, the routine getword gets a word from the full screen

```
getword(string);
```

and wgetword gets a word from a window

```
wgetword(win, string);
```

Here is a list of the Screen Interface Library routines:

1) goraw()
   sets up standard output for cbreak mode

2) gonormal()
   resets the terminal back to its normal state

3) getkey()
   gets a single key typed on the keyboard

4) wgetword(win, string)
   get a word from the display

5) wmesg(win, string, data)
   output a message in standout mode on the last line of win

6) wmvcursor(win, c, top, bottom)
   use the character c and move the cursor appropriately on win

7) wforword(win, top, bottom)
   move to the beginning of the next word

8) wbackword(win, top, bottom)
   move to the beginning of the previous word

9) wforspace(win, top, bottom)
   move to the next space

10) wbackspace(win, top, bottom)
   move to the previous space

11) wright(win, top, bottom)
   move cursor to the right

12) wleft(win, top, bottom)
   move cursor to the left

13) whighlight(win, flag)
   highlight or unhighlight the word at the current position

14) wsavescrn(win)
   save the contents of win

15) wresscrn(win)
   restore a previously saved window and overwrite win

16) wscrolf(win)
   scroll forward on win

17) wscrolb(win)
   scroll backward on win

18) wpagefor(win, fp, top)
page forward on win using the file associated with the pointer, fp

19) wpageback(win)
page backward on win

20) wcolon(win)
handle colon commands (for example, :q for quit)

21) whelp(win, file)
handle help commands using the help file, file

As an optional feature during cursor movement, the cursored item can be displayed in standout mode to highlight the item. This is accomplished by setting the global flag, hilite, to TRUE. Also, note that all error and informational messages are displayed in standout mode on the last line of the window.

A more detailed explanation for each of the Screen Interface Library routines is included in the Appendix A.

SECTION 3
INVOCATION AND OPERATION

3.1. Programming

When using the Screen Interface Library, the programmer must ensure that the following lines are included in the source program:

```
#include <curses.h>

#include <screen.h>
```

These header files contain the global definitions and variables used by both the Curses and Screen Interface libraries. In addition, pseudo functions for the standard screen are defined; this means that the routine parameter, win, is assumed to be equivalent to the Curses variable, stdscr. For example, the routine call

```
help(file)
```

is defined as

```
whelp(stdscr, file)
```

When compiling the screen program, the following command line should be used:

```
cc [flags] file.c -lscreen -lcurses -ltermlib
```

3.2. Normal Termination

Upon normal termination of a routine in the Screen Interface Library, the value OK (=1) or a specific value is returned (OK is defined in /usr/include/curses.h). Specific values (if any) returned by each routine are outlined in Appendix A.

3.3. Abnormal Termination

Upon abnormal termination of a routine in the Screen Interface Library the value ERR (=Ø) is returned (ERR is defined in /usr/include/curses.h).

APPENDIX A
SUMMARY OF LIBRARY ROUTINES

This appendix contains descriptions of the routines available in the Screen Interface Library. If applicable, there are two calling sequences available (one for a window and one for stdscr) where the w version of the routine can be applied to a window defined by the user. The alternate version of the routine assumes the standard or full terminal screen.

goraw()
: This routine sets standard output for CBREAK mode; in addition, it sets the Curses flag, _rawmode to TRUE (=1).

gonormal()
: This routine resets standard output back to its normal mode and resets _rawmode back to FALSE (=Ø).

getkey()
: This routine gets a single character input from the keyboard.

  If any of the arrow keys is typed, the standard definition found in /usr/include/screen.h (i.e. left, down, up, and right) is returned. The following list contains the aliases for the arrow keys:

```
left - h, CTRL-h, backspace
down - j, CTRL-j
up - k, CTRL-k
right - l, CTRL-l, space
```

  Alternatively, if none of the arrow keys is typed, the character typed is returned. In addition, if a carriage return is typed, the character \r is returned. This is because some terminals generate a line feed or \n character for the down arrow; therefore, a distinction must be made between the RETURN key (which is mapped to line feed) and the down arrow key.

```
wgetword(win, str)
WINDOW      *win;
char        *str;
```

or

```
getword(str)
char        *str;
```

This routine gets a word from the display and puts the string in str; if the word is highlighted in the display, the standout mode bit in each character is masked out and returned in str.

```
wmesg(win, str, data)
WINDOW      *win;
char        *str;
char        *data;
```

or

```
mesg(str, data)
char        *str;
char        *data;
```

This routine outputs the printf-formatted message, str, on the last line of win or stdscr. It is noted here that a newline or \n is not required since the message is output on the last line of the window. Any additional data to be output (for example, for a %s in mesg), is stored in the variable, data. If there is no additional data (that is, str is a simple informational message), data should contain NULL. After outputting the message, the cursor is returned to the current position.

```
wmvcursor(win, c, top, bottom)
WINDOW      *win;
char        c;
int         top, bottom;
```

or

```
mvcursor(c, top, bottom)
char        c;
int         top, bottom;
```

This routine uses the given character, c, to move the cursor appropriately about win within the display limits of the top and bottom line. The valid values for c (and therefore, valid cursor movements) are down, up, forward (or word) or backward as defined in /usr/include/screen.h (see Appendix B). If the bottom line limit is exceeded, the cursor will be moved to the top of the next column to the right or to the top of the leftmost column; therefore, there is cursor wraparound. After the movement is performed, the routine

returns OK. If c represents an invalid cursor movement, the routine returns the value ERR. If the global flag, hilite, is set, highlighting of the word at the current cursor position is handled automatically.

```
wforword(win, top, bottom)
WINDOW      *win;
int         top, bottom;
```

or

```
forword(top, bottom)
int         top, bottom;
```

This routine moves the cursor to the beginning of the next word (to the right) on the display. If the cursor is at the bottom line of the rightmost column, the cursor is wrapped around to the top of the leftmost column.

```
wbackword(win, top, bottom)
WINDOW      *win;
int         top, bottom;
```

or

```
backword(top, bottom)
int         top, bottom;
```

This routine moves the cursor to the beginning of the previous word (to the left) on the display. If the cursor is at the leftmost position of the top line, the cursor is wrapped around to the last word of the rightmost column.

```
wforspace(win, top, bottom)
WINDOW      *win;
int         top, bottom;
```

or

```
forspace(top, bottom)
int         top, bottom;
```

This routine moves the cursor to the right until a space is reached on the display. If the cursor is at the bottom line of the rightmost column, the cursor is wrapped around to the top of the leftmost column.

```
wbackspace(win, top, bottom)
WINDOW      *win;
```

```
int         top, bottom;
```

or

```
backspace(top, bottom)
int         top, bottom;
```

This routine moves the cursor to the left until a space is reached on the display. If the cursor is at the leftmost position of the top line, the cursor is wrapped around to the last word of the rightmost column.

```
wright(win, top, bottom)
WINDOW      *win;
int         top, bottom;
```

or

```
right(top, bottom)
int         top, bottom;
```

This routine moves the cursor one position to the right. If the cursor is at the bottom line of the rightmost column, the cursor is wrapped around to the top of the leftmost column.

```
wleft(win, top, bottom)
WINDOW      *win;
int         top, bottom;
```

or

```
left(top, bottom)
int         top, bottom;
```

This routine moves the cursor one position to the left. If the cursor is at the leftmost position of the top line, the cursor is wrapped around to the last word of the rightmost column.

```
whighlight(win, flag)
WINDOW      *win;
int         flag;
```

or

```
highlight(flag)
```

This routine puts the word at the current cursor position in standout mode (thus, highlighting the word) if

flag is TRUE. If flag is FALSE, standout mode is turned off for the word at the current position.

```
wsavescrn(win)
WINDOW      *win;
```

or

```
savescrn()
```

This routine saves the contents of win in the global WINDOW, scrn (found in /usr/include/screen.h), which is allocated memory in this routine. If there is a problem with the allocation, this routine returns ERR. Otherwise, the global flag scrnflg is set to TRUE, the contents of win is saved, and the routine returns OK.

```
wresscrn(win)
WINDOW      *win;
```

or

```
resscrn()
```

This routine tests the global flag, scrnflg (set to TRUE in wsavescrn) and checks whether the contents of the WINDOW scrn will fit on win. If so, the contents of scrn is overwritten onto win and the routine returns OK; otherwise, the routine returns ERR.

```
wscrolf(win)
WINDOW      *win;
```

or

```
scrolf()
```

This routine performs scrolling forward on win; this has not been implemented yet.

```
wscrolb(win)
WINDOW      *win;
```

or

```
scrolb()
```

This routine performs scrolling backward on win; this has not been implemented yet.

```
wpagefor(win, fp, top)
WINDOW      *win;
FILE        *fp;
int         top;
```

or

```
pagefor(fp, top)
FILE        *fp;
int         top;
```

This routine outputs a page of the file associated with the pointer, fp. If the number of lines in win is exceeded, the prompt

Type ^f for next page

is output. If ^f is not typed, the routine returns; otherwise, the next page of data is output.

```
wpageback(win)
WINDOW      *win;
FILE        *fp;
```

or

```
pageback()
FILE        *fp;
```

This routine outputs the previous page of the file associated with the pointer, fp; this has not been implemented yet.

```
wcolon(win)
WINDOW      *win;
```

or

```
colon()
```

This routine handles the colon commands (the colon is echoed on the last line of win). A character followed by a carriage return is the expected typein; the routine returns the character typed.

```
whelp(win, file)
WINDOW      *win;
char        *file;
```

or

```
help(file)
char        *file;
```

This routine opens the given help file, file, and displays its contents; following the display, the original screen is restored. If the help file cannot be opened or if there was a problem restoring the original screen, the routine returns ERR; otherwise, the routine returns OK.

APPENDIX B
PROGRAMMING INFORMATION

This appendix contains a description of the contents of the header file, /usr/include/screen.h and the global messages available in the Screen Interface Library.

These are simply definitions for the file descriptors for standard input, standard output, and standard error.

```
#define     INFILE  Ø
#define     OUTFILE 1
#define     ERRFILE 2
```

The following macro is used by the library routines to interpret CTRL characters.

```
#define     CTRL(c)         ('c' & Øx1f)
```

These definitions are the standard keys used in the Screen Interface package.

```
#define     LEFT            'h'
#define     RIGHT           'l'
#define     UP              'k'
#define     DOWN            'j'
#define     BACKWARD        'b'
#define     FORWARD         'f'
#define     WORD            'w'
#define     COLON           ':'
#define     HELP            '?'
#define     PAGEFOR         CTRL(f)
#define     PAGEBACK        CTRL(b)
```

The following list contains the aliases for the functions using stdscr; these definitions simply assume the variable win equals stdscr when writing programs for the standard terminal screen.

NOTE

The appearance of the following list has been modified due to space limitatons. Please see /usr/include/screen.h for the original text.

```
/*
 * pseudo functions for standard screen
 */
```

```
#define   mesg(s, d)
             wmesg(stdscr, s, d)
#define   mvcursor(c, top, bottom)
             wmvcursor(stdscr, c, top, bottom)
#define   forword(top, bottom)
             wforword(stdscr, top, bottom)
#define   backword(top, bottom)
             wbackword(stdscr, top, bottom)
#define   forspace(top, bottom)
             wforspace(stdscr, top, bottom)
#define   backspace(top, bottom)
             wbackspace(stdscr, top, bottom)
#define   right(top, bottom)
             wright(stdscr, top, bottom)
#define   left(top, bottom)
             wleft(stdscr, top, bottom)
#define   highlight(flag)
             whighlight(stdscr, flag)
#define   getword(str)
             wgetword(stdscr, str)
#define   colon()
             wcolon(stdscr)
#define   help(file)
             whelp(stdscr, file)
#define   savescrn()
             wsavescrn(stdscr)
#define   resscrn()
             wresscrn(stdscr)
#define   scrolf()
             wscrolf(stdscr)
#define   scrolb()
             wscrolb(stdscr)
#define   pagefor()
             wpagefor(stdscr)
#define   pageback()
             wpageback(stdscr)
```

These global variables are used by the Screen Interface Library. If the variable, hilite is set, the library routines assume that the cursored item is in standout mode. The variable, scrnflg, is set by the routine wsavescrn whenever storage has been allocated for the WINDOW scrn. The user program can use this variable in order to clean up any allocated storage upon exit.

```
int         hilite;
int         scrnflg;
```

The variable, scrn, is used by the routines wsavescrn and wresscrn for saving and restoring screens.

```
WINDOW      *scrn;
```

The following global messages are also defined in the library; to use these messages, simply declared the message variable as extern (for example, extern char *crmsg).

```
char        *crmsg = "Type <CR> to continue";

char        *qmsg  =  "Type ?<CR> for help";
```

APPENDIX C
USER GUIDE

This appendix is intended to be a user's guide for using the Screen Interface Library routines. The following is an outline of what should be included in a typical screen program which outputs a display and allows cursor movement; in this example, stdscr is assumed.

```
#include <curses.h>
#include <screen.h>
#include <signal.h>

     :
     :

/* top line of the display */
#define   TOP 2

/* name of help file */
#define HELPFILE "/usr/lib/screen/helpfile"

#define SOMELENGTH 14

     :
     :

/* bottom line of the display */
int  bottom;

     :

extern  char *qmsg, *crmsg;

     :

main(argc, argv)
int  argc;
char **argv;
{
     /*
      * parse options
      */

          :
          :

     /*
```

```
        * set or reset hilite as default
        */

            :
            :

       /*
        * initialize for the Screen Interface Library
        */
       initialize();

       /*
        * get data for display
        */
       getdata();

       /*
        * handle keyboard input
        */
       keyinput();

}

/*
 * basic initialize routine
 */
initialize()
{
       /* interrupt routine */
       int  done();

       /*
        * set up terminal for Curses
        */
       initscr();

       /*
        * set up interrupt signal
        */
       signal(SIGINT, done);

       /*
        * go to cbreak mode
        */
       goraw();
}

/*
 * this routine merely gets the display information and
 * stores it in some internal buffer
```

```
 */
getdata()
{

     :
     :
}

/*
 * handle keyboard input (in this routine, assume that
 * only 'right arrow',
 * ':q', and '?<CR>' are the valid commands)
 */
keyinput()
{
     char c;

     /*
      * output display
      */
     dsp();
     move(TOP, Ø);
     refresh();

     /*
      * loop until ':q' is typed
      */
     do
     {
          if (hilite)
               highlight(TRUE);

          while (TRUE)
          {
               c = getkey();
               if c = (RIGHT || c == COLON || c == HELP)
                    break;
               /*
                * if invalid cursor movement,
                * output '?<CR>' message
                */
               if (mvcursor(c, TOP, bottom) == ERR)
                    mesg(qmsg, NULL);
          }

          /*
           * perform command
           */
          c = cmd(c);
     }
     while (c != 'q');
```

```
	/*
	 * clean up before exit
	 */
	done();
}

/*
 * output the display
 */
dsp()
{
	/*
	 * set up the screen for the display
	 */
	erase();
	refresh();
	move(0, 0);
	printw("Some Title\n\n");
	refresh();

	/*
	 * print out the internal buffer of information
	 * (either printw or addstr can be used)
	 */
	printw("here\n");
	printw("there\n");
	printw("everywhere\n");

		:
		:

	/*
	 * set up bottom of the display
	 */
	bottom = stdscr->_cury - 1;
	refresh();
}

/*
 * perform command after 'right arrow', ':' or '?' is typed
 */
cmd(c)
char c;
{
	int  i, j;
	char str[SOMELENGTH];

	/*
	* get the cursored item
	*/
	getword(str);
```

```
        /*
         * perform command
         */
        switch (c)
        {
          case RIGHT:
          /*
           * this code performs the command on
          the cursored item
           */
          i = stdscr->_cury;
          j = stdscr->_curx;
          mesg("here it is - type <CR>", NULL);
          while (getkey() != '\r');

             :
             :

          /*
           * redisplay
           */
          dsp();
          move(i, j);
          break;

         case COLON:
          if (colon() == 'q')
                  return('q');
          break;

         case HELP:
          if (help(HELPFILE) == ERR)
               mesg ("Cannot display help file", NULL);
          break;
        }
        return(c);
}

/*
 * clean up routine
 */
done()
{
        signal(SIGINT, done);
        if (hilite)
             highlight(FALSE);
        move(stdscr->_maxy - 1, 0);
        refresh();
        if (scrnflg)
             delwin(scrn);
        gonormal();
```

```
        endwin();
        exit(Ø);
}
```

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 1-1

SECTION 2 BASIC SPECIFICATIONS 2-1

SECTION 3 ACTIONS 3-1

SECTION 4 LEXICAL ANALYSIS 4-1

SECTION 5 HOW THE PARSER WORKS 5-1

SECTION 6 AMBIGUITY AND CONFLICTS 6-1

SECTION 7 PRECEDENCE 7-1

SECTION 8 ERROR HANDLING 8-1

SECTION 9 THE YACC ENVIRONMENT 9-1

SECTION 10 HINTS FOR PREPARING SPECIFICATIONS 10-1

10.1. General 10-1
10.2. Input Style 10-1
10.3. Left Recursion 10-1
10.4. Lexical Tie-Ins 10-2
10.5. Reserved Words 10-3

SECTION 11 ADVANCED TOPICS 11-1

11.1. Simulating Error and Accept in Actions 11-1
11.2. Accessing Values in Enclosing Rules 11-1
11.3. Support for Arbitrary Value Types 11-2

APPENDIX A YACC INPUT SYNTAX A-1

APPENDIX B A SIMPLE EXAMPLE B-1

APPENDIX C AN ADVANCED EXAMPLE C-1

APPENDIX D OLD FEATURES D-1

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 ('\Ø' or Ø) 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 Ø or a negative number. This token number cannot be redefined by the user; thus, all lexical analyzers must be prepared to return Ø 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 Ø, the stack contains only state Ø, 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 Ø
        $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 Ø. The parser refers to the input to select among the actions available in state Ø, so the first token (DING) is read, becoming the lookahead token. The action in state Ø 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 Ø, 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 Ø. Consulting the description of state Ø, 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 Ø, 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 Ø, 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 Ø again. In state Ø, 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 expr (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 Ø.

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 Ø. 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 reserved 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 Ø. 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
                        {           $$ = DOG;  }
        |       CRONE
                        {           if( $Ø == YOUNG ){
                                        printf( "what?\n" );
                                        }
                                    $$ = 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 $Ø 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>Ø );  }
        ;
```

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 Ø (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  -  'Ø';
        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.

The 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
%Ø 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.

Systems Publications

Reader's Comments

Your feedback about this document helps us ascertain your needs and fulfill them in the future. Ple take the time to fill out this questionaire and return it to us. this information will be helpful to us and, 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