

NEW SUBSCRIBERS: See Section A for details of bringing up UCSD Pascal on your machine.

DISCLAIMER: These documents and/or the software they describe are subject to change and/or correction without notice. The UCSD Pascal Project cannot be held responsible for implementations on processors where the implementation work was not done at UCSD. Users with systems obtained from sources other than UCSD must contact their supplier for support.

ACKNOWLEDGEMENTS:

The work described in these notes has been supported significantly by the following organizations:

United States Navy Personnel Research and Development Center, Sperry Univac Minicomputer Operations, EDUCOM, Digital Equipment Corporation, Processor Technology Inc., Springer-Verlag, Terak Corporation, General Automation Corporation, The UCSD Computer Center, grants from the University of California Instructional Improvement Program, Tektronix Corporation, Micropolis Inc., Computer Power and Light, Phillips Research Labs, Lawrence Livermore Labs, Pascal Computing.

The work described in these notes has been made possible by the drive and direction of the Director of the IIS:

Kenneth L. Bowles

Documentation Authors:

Gillian M. Ackland, S. Dale Ander, Lucia A. Bennett, Raymond S. Causey, Charles "Chip" Chapin, Gary J. Dismukes, Julie E. Erwin, Shawn M. Fanning, Mary K. Landauer, J. Raoul Ludwig, Joel J. McCormack, Mark D. Overgaard, Keith A. Shillington, David A. Smith, Roger T. Sumner, Dennis J. Volper.

Software Authors:

S. Dale Ander, Marc Bernard, Charles "Chip" Chapin, J. Greg Davidson, Barry Demchak, William P. Franks, C. Richard Grunsky, Robert J. Hopkin, Albert A. Hoffman, Richard S. Kaufmann, Peter A. Lawrence, Joel J. McCormack, Mark D. Overgaard, David A. Reisner, Keith A. Shillington, David M. Steingre, Roger T. Sumner, Steven S. Thompson, David B. Wollner.

Collected and Edited by:

Keith Allan Shillington and Gillian M. Ackland.

 * TABLE OF CONTENTS *

Version I.5 September 1978

SECTION	PAGE
1 THE UCSD PASCAL SYSTEM	
1 INTRODUCTION AND OVERVIEW	1
2 FILE HANDLER	7
3 SCREEN ORIENTED EDITOR	
1 INTRODUCTION	31
2 GETTING STARTED	32
3 DETAILED DESCRIPTION OF COMMANDS	36
4 REFERENCE	51
5 EXPERIMENTAL LARGE FILE VERSION (L2)	53
4 YET ANOTHER LINE ORIENTED EDITOR - YALOE	59
5 DEBUGGER	71
6 PASCAL COMPILER	81
7 BASIC COMPILER	89
8 LINKER	95
9 ASSEMBLER	99
2 THE UCSD PASCAL LANGUAGE	
1 INTRINSICS	117
1 STRING	119
2 INPUT/OUTPUT	123
3	
4 LOW LEVEL GRAPHICS	129
5 CHARACTER ARRAY MANIPULATION	131
6 MISCELLANEOUS	133
2 DIFFERENCES BETWEEN UCSD'S PASCAL AND STANDARD PASCAL	135
1 CASE STATEMENTS	135
2 COMMENTS	136
3 DYNAMIC MEMORY ALLOCATION	136
4 EOF	138
5 EOLN	138
6 FILES	149
7 GOTO AND EXIT STATEMENTS	142
8 PACKED VARIABLES	144
9 PARAMETRIC PROCEDURES AND FUNCTIONS	147
10 PROGRAM HEADINGS	147
11 READ AND READLN	148
12 RESET	150
13 REWRITE	150
14 SEGMENT PROCEDURES	150
15 SETS	151
16 STRINGS	152
17 WRITE AND WRITELN	156
18 IMPLEMENTATION SIZE LIMITATIONS	156
19 EXTENDED COMPARISONS	156
20 LONG INTEGERS	156
21 UNITS	156
22 TABLE OF UCSD INTRINSICS	156

3 IMPLEMENTORS' GUIDES

1	DRAWLINE	159
2	FILE FORMATS	163
3	SPECIAL UCSD PASCAL SYNTAX (USE OF)	
1	SEGMENT PROCEDURES	165
2	UNITS	167
3	LONG INTEGERS	179
4	INTERPRETER NOTES	183
5	INTRODUCTION TO THE PASCAL PSEUDO-MACHINE	201
6	BYTE SWAPPING	213

4 UTILITY PROGRAMS

1	CALCULATOR	215
2	LIBRARIAN	217
3	SETUP - SYSTEM RECONFIGURATION	221
4	BOOTSTRAP COPIER	227
5	PATCH/DUMP	229
6	RT11 TO PASCAL CONVERSION KIT	233
7	GOTOXY PROCEDURE BINDER	235
8	DUPLICATE DIRECTORY	237
9	P-CODE DISASSEMBLER	239
10	LIBRARY MAP	245

5 TABLES

1	EXECUTION ERRORS	249
2	IORESULTS	251
3	UNITNUMBERS	253
4	PENSTATES	255
5	SYNTAX ERRORS	257
6	ASSEMBLER SYNTAX ERRORS	261
7	AMERICAN STANDARD CODE for INFORMATION INTERCHANGE	265
8	P-MACHINE OP-CODES	267
9	UCSD PASCAL SYNTAX DIAGRAMS	268A

A ADDENDA, ERRATA AND NOTES

1	NOTES ON OTHER MATERIALS AVAILABLE	269
2	BRINGING UP THE PASCAL SYSTEM	
1	ON PDP-11	271
2	ON 8080/Z80 SYSTEM WITH CP/M AND 3740 DISKS	273
3	DIFFERENCES AMONG IMPLEMENTATIONS FOR DIFFERENT PROCESSORS.	277
4	CHANGES MADE IN I. 5 FROM (I. 4, I. 4b) SYSTEMS	279

B	INDEX	285
---	-----------------	-----

* INTRODUCTION AND OVERVIEW * * Section 1.1 *

Version I.5 September 1978

The UCSD Pascal system described in the following document is a system intended to run on stand alone micro- and mini-computers. This system is highly machine independent since it runs on a pseudo-machine interpreter commonly referred to as the "P-machine". All the system software is written in Pascal, except for the P-machine interpreter and a few run-time support routines written in assembler for efficiency, resulting in relatively straightforward software maintenance and enhancement.

The system is designed to be used primarily with a CRT terminal acting as the CONSOLE device; however, the system is flexible enough to be reconfigured for slower hard-copy terminals. For further information regarding compatibility between various types of equipment and this system see the "SETUP" document in Section 4.3. This document is intended for programmers who are familiar with the Pascal programming language and have some experience in writing computer programs.

The following is a tutorial book on PASCAL:

Kenneth L. Bowles,
(Microcomputer) Problem Solving Using PASCAL
Springer-Verlag, New York, (c)1977

We suggest the following book as a PASCAL reference guide:

Kathleen Jensen and Niklaus Wirth,
PASCAL User Manual and Report
Springer-Verlag, New York, (c)1975

For documentation concerning the differences between UCSD Pascal and Standard Pascal see Section 2.2.

1.1.1 THE UCSD PASCAL SYSTEM: AN OVERVIEW

The structure of the UCSD Pascal system is best conceptualized in terms of the "tree-like" structure diagram figure 0.1 at the end of this sub-section.

The diagram in figure 0.1 depicts the outermost level of the system. In terms of a "tree" or structure diagram, the "root" corresponds to the outermost level, while the "leaves" (i.e. the boxes with no branches to lower levels) correspond to the lower levels of the system. While a user is in a particular level, the system displays a list of available commands called the "prompt-line". If the system is running on a CRT screen type terminal, then the prompt-line will usually appear at the top of the screen. Commands are usually invoked by typing a single character from the CONSOLE device. For example, the prompt-line for the outermost level of the system is:

Command: E(edit, R(un, F(file, C(omp, L(link, X(ecute, A(ssem, D(ebug, ? [I.5]

By typing "F" the user will "descend" a level within the structure diagram into a level called the "Filer". Upon entering the Filer, another prompt-line detailing the set of commands available at the Filer level of the system is displayed. The Q(uit command causes the user to exit from the Filer and "ascend" back to the outermost command level of the system. Now the user is back at the level in the system from which he started after bootstrapping the machine. Some commands within the system prompt the user for the name of some disk file. In these cases, the user enters the name of the file followed by a carriage return. If an error is made in typing a portion of the file name, the backspace key (or equivalent key depending upon the system configuration) may be used to "back over" and erase the erroneous part. The line delete key (rubout key) may be used to erase the entire file name, thereby allowing the user to completely start over. If the user decides not to accept any file name whatsoever, "escape" from this command is by entering a file name of zero characters, i.e. type <cr>.

Note that due to a limited amount of room on the prompt-line, some of the infrequently used commands may not appear on the prompt-line.

A concept central to the design of the entire UCSD Pascal system command structure is the concept of the "workfile". A workfile can be thought of as a "scratch-pad" area used for development of programs and only one workfile is allowed at any one time. If a user wishes to begin a new workfile, the contents of the old one can be saved, under a separate file name, for later reference by using the S(ave command in the Filer level of the system. When that file is later retrieved for further work on the contents, it is possible that a number of files (usually source and code) will be retrieved together and in total they comprise the work-file.

1.1.2 OUTERMOST LEVEL COMMANDS: AN OVERVIEW

A. E(edit)

Typing "E" while at the outermost command level of the system causes the editor program to be brought into memory from disk. The user may, while in the editor, insert or delete text inside his workfile or any textfile, along with many other powerful commands. See Section 1.3 for details. The workfile text (if present) is read into the editor buffer, otherwise the Editor prompts for a file.

B. F(iler)

"F" places the user in a level of the system called the Filer. This section of the system contains commands used primarily for maintenance of the files stored on the disk. The L(dir command allows the user to list the titles and the last modification date, as well as determine the number of blocks occupied by each file on the disk. The T(transfer command is used to copy from either one disk to another, or from one area on a particular disk to another area on the same disk. For more documentation on the Filer level including commands associated with the "getting", "saving", and "clearing" of the user's workfile see Section 1.2.

C. C(omp)

This command initiates the system compiler to compile the users work-file. If there is no work-file currently the user is asked for a source text file name. If a syntax error within the source is detected, the compiler will stop and display the error number and the surrounding text of the program. By typing a space, the user can cause the compiler to continue the compilation. Typing an <esc> causes the compiler to abort & return to Command level. Typing 'E' will, if the system editor is the screen editor, call the editor placing the cursor near the offending symbol. If the compilation is successful, (i.e. no syntax errors were encountered) a codefile called *SYSTEM.WRK.CODE is written out onto the user's disk and becomes part of the workfile. For more documentation on the use of the UCSD Pascal compiler see Section 1.6.

D. R(un)

This command causes the codefile associated with the current workfile to be executed. If no such code file currently exists, the compiler is called in the same manner as described in C above. If the compilation requires linkage to separately compiled code the linker will automatically be invoked and will assume the use of the file *SYSTEM.LIBRARY. After a successful compilation, the program is executed.

E. X(ecute

This command prompts the user for the filename of a previously compiled codefile. If the file exists, the codefile is executed; otherwise the message "can't find file" is returned. (Note: the ".CODE" suffix on such a file is implicit.) If all code necessary to execute the codefile has not been linked in, the message "file <fileid> not linked in" is returned. It is convenient to X(ecute other programs which have already been compiled because otherwise the user would have to enter the Filer, G(et the file, G(uit the Filer, and then R(un the program.

F. A(ssem

Just like C(omp except the system assembler is invoked rather than the system compiler.

G. D(ebug

This command causes the current workfile to be executed. If the program in the workfile has not been compiled, the compiler will be called as in the case of the R(un command. However if a run-time error occurs, or a user-defined break-point or halt is encountered, the Debugger program is called. The Debugger is a program which allows the user to examine the contents of variables within the program. See section 1.5 Debugger for more details.

H. L(link

This command starts the system linker program explicitly to allow users to link routines from libraries other than *SYSTEM.LIBRARY. See section 1.8 for more information on the Linker.

1.1.3 UTILITY PROGRAMS

There are many functions needed by users of any operating system. To attempt to make all these functions system functions would result in a terrible proliferation of command letters as the base node level. In order to keep the COMMAND line simple, we have restricted the functions available on it to what we feel is the bare minimum for program and text development. The other useful, but much less often used functions are available through the X(ecute command. The sort of functions which are available are the desk calculator, the patch/dump utility, the terminal configuration setup program, a bootstrap mover, a librarian and many others. For a complete list of the utility programs now available with the UCSD Pascal system, reference Section 4 in the Table of Contents. Any programs which you write and feel would be a useful addition to our library of utilities will be welcome contributions.

1.1.4 AN INTRODUCTION TO THE UCSD PASCAL SYSTEM

1.5 is the first release which contains the fully intergrated and implemented concept of separate compilation and assembly. 1.4b was the first to support multiple types of processors.

The great bulk of the system software is written in Pascal and runs on a relatively simple pseudo-machine. If this pseudo-machine is emulated by a machine language program on a new real machine, the Pascal software will also run on that new real machine.

One class of differences among versions of the system is due to aspects of the pseudo-machine that are not identically emulated by the implementations for different types of processors. A subsection in section A contains a chart of differences between processors the system currently runs on.

Another class of differences stems from variations in the system I/O environments rather than in the host processor. Included here are difference in system console terminal types (a.e. hard-copy vs CRT vs storage tube) or command conventions and capabilities (eg. "intelligent" vs "dumb" CRT's). The system is intended to be able to cope with this sort of variation. Version 1.4 had some troubles with terminals that generate/require two-character sequences for some controls, and single-character sequences for others. The utility program "SETUP" has been completely regenerated for 1.5 (see section 4.3).

In the PDP-11 world these mass storage variations are not too serious, primarily because there is considerable motivation to be compatible with DEC devices and media. We have written and support drivers for a few DEC incompatible devices but make no claim to support users who want to develop their own such drivers. See section A for warnings about problems you might encounter.

The situation in the BOBO/Z80 world is much more chaotic. Since it would not be practical for the Project to write and support drivers for the vast multitude of BOBO/Z80 I/O environments that exist, we have chosen to take advantage of the widespread implementation of Digital Research's CP/M operating system by structuring the pseudo-machine's I/O operations as calls on CP/M's Basic I/O Subsystem (BIOS) primitives. Therefore, any I/O configuration on which CP/M has been implemented should also be able to support the Pascal system. We do not guarantee this. For example, Intel MDS disk controllers cannot read disks generated here and some BIOS's we have encountered do not completely meet all the requirements specified for CP/M. UCSD plans to support some of the larger distribution BOBO-based machines directly.

Our dominant mode of distribution for BOBO/Z80 systems will be on 3740 compatible diskettes. One of the distribution diskettes will be CP/M oriented. This disk will be used, via a somewhat awkward two-step process, to bring up UCSD Pascal on a particular CP/M configuration. Look to section A for details on this process. It also describes the configuration of a modified BIOS, which will better support the needs of the Pascal system. Finally, directions are given for making it possible to boot directly to Pascal rather than indirectly through a CP/M program.

A number of files on the disk start with 'SYSTEM.' specifically:

SYSTEM.PDP-11
SYSTEM.MICRO
SYSTEM.PASCAL
SYSTEM.FILER
SYSTEM.COMPIILER
SYSTEM.SYNTAX
SYSTEM.EDITOR
SYSTEM.LINKER
SYSTEM.ASSMBLER
SYSTEM.SWAPDISK
SYSTEM.CHARSET
SYSTEM.LIBRARY
SYSTEM.WRK.TEXT
SYSTEM.WRK.CODE
SYSTEM.STARTUP

In most cases these files contain the system segment of the name they carry. That is to say that the EDITOR, FILER, LINKER, COMPILER, ASSEMBLER are the files that are invoked by the text editor when 'E', 'F', etc. is typed. Some of the files are machine specific. INTERP and MICRO are the files which contain the interpreters for the particular machine being used. CHARSET is a file which appears on disks meant for TERA computers only and contains the definition for the soft character set, and the data for the Triton logo prompt. LIBRARY is a file containing separately assembled or compiled routines for use by the Linker in producing executable code files. PASCAL contains the operating system, and the Debugger. SWAPDISK is a file used by some of the system segments during compilation of "include" files if a memory shortage exists. It is a 2048 byte file which gets a portion of memory swapped to it when a directory needs to be read into core. When the directory work is complete, the memory is restored to its original state. STARTUP is a file which can be created at the user's option. If it exists on a disk, the operating system considers it a runnable code-file, and executes it at initialize time. This allows the user to have a program that runs before the main command prompt comes up, and will run anytime the I(nitalize command is typed. WRK.TEXT and WRK.CODE are the current work-file after some action has occurred to the work-file. They appear after having done some text editing on a work-file (SYSTEM.WRK.TEXT) or compiling a work-file (SYSTEM.WRK.CODE).

All other files on the disk are user generated (in one fashion or another). The other important parts of a disk are relatively invisible to the user. The directory resides at block 2 on the disk and extends for 4 blocks if it is a single directory, 8 blocks if it is a duplicated (backed-up) directory. The bootstrap can reside at any of a number of places on the disk, depending on the host machine. In most cases, blocks 0 and 1 are reserved for the bootstrap.

* FILEHANDLER * * Section 1.2 *

Version 1.5 September 1978

1.2.1 FILES

A file is a discrete 'chunk' of information which is stored on the disk and referenced by a filename. Each disk has a directory which contains the filenames and locations of each file on the disk. The Filehandler, or Filer, uses the information contained in the disk directory to manipulate files.

One of the attributes of a file is its type. The type of the file determines the way in which it can be used. File types are assigned based on the file name.

Reserved type suffixes for filenames are:

.TEXT	Human readable text.
.CODE	Machine executable code.
.DATA	Data file.
.FOTO	A file containing one TERA screen-image.
.GRAF	Intended to be a file containing a vector list of a graphic image. Currently unused.
.BAD	An unmovable file covering a physically damaged area of a disk.

1.2.2 VOLUMES

A volume is any I/O device, such as the printer, the keyboard, or a disk. A "block-structured" device is one that can have a directory and files, usually a disk of some sort. A non-block-structured device does not have internal structure; it simply produces or consumes a stream of characters. The printer and the keyboard, for example, are non-block-structured. The table below illustrates the reserved volume names used to refer to non-block-structured devices, the 'unit number' associated with each device, and the unit numbers associated with the system (booted) disk and any alternate disks.

Unit Number	Volume ID	Description
1	CONSOLE:	screen and keyboard with echo
2	SYSTEM:	screen and keyboard without echo
3	GRAPHIC:	the graphic 'side' of the screen
4	<volume name>:	the system disk
5	<volume name>:	the alternate disk
6	PRINTER:	the line printer
8	REMOTE:	additional peripherals
9-12	<volume name>:	additional disk drives

FIGURE 1

1.2.3 THE 'WORKFILE'

The workfile is a temporary copy of the file being modified. It is used by the Filer, in the Editor, and by the Compiler. When the text part of a workfile is changed, the system stores it on disk under the name '*SYSTEM.WRK.TEXT', and when a code version is first created, it is named '*SYSTEM.WRK.CODE'.

1.2.4 FILE SPECIFICATION

Many Filer commands require the user to respond with at least one file specification. The diagram below illustrates the syntax of file specification.

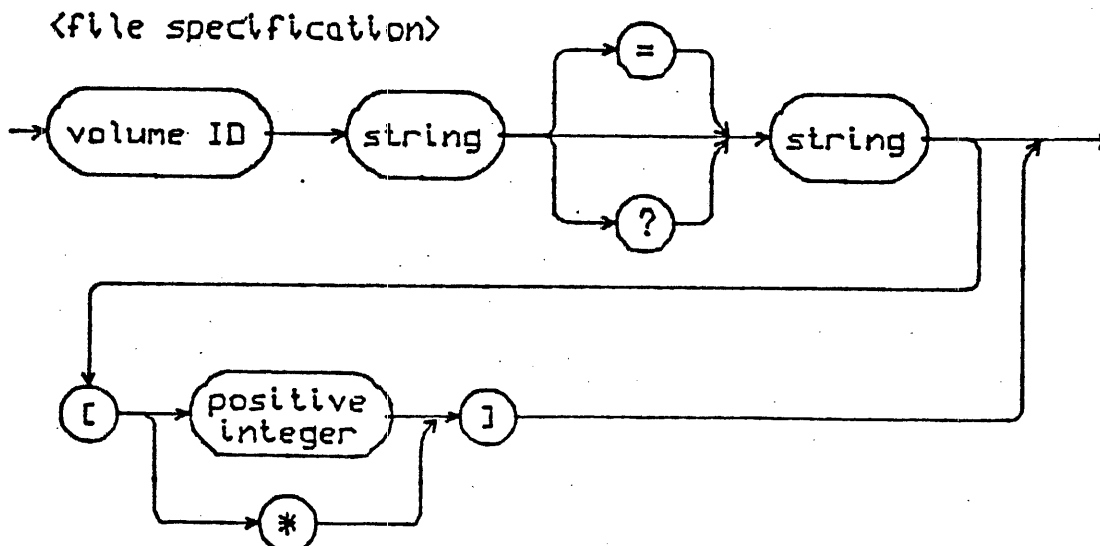


FIGURE 2

Volume i. d. syntax can be expanded thusly:

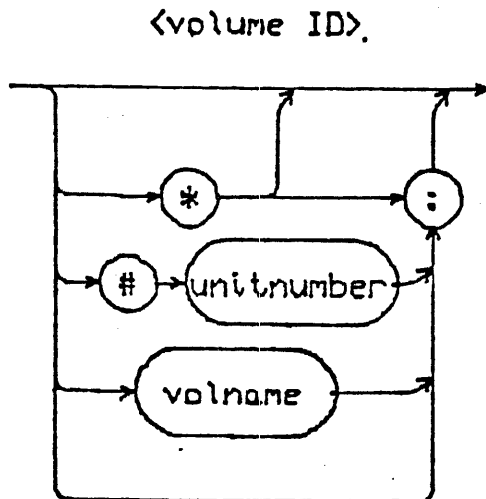


FIGURE 3

Volume names for block-structured volumes can be arbitrarily assigned by the user. A volume name must be 7 or less characters long and may not contain '=', '\$', '?' or ','. Reserved volume names for non-block-structured devices are given in Figure 1. The character '*' is the volume ID of the 'system disk', the disk upon which the system was booted. The character ':', when used alone, is the volume ID of the 'default disk'. The system disk and default disk are equivalent unless the default prefix (see material on P(refix) has been changed. '#Cunit

number>' is equivalent to the name of the volume in the drive at that time.

A legal filename can consist of up to 15 characters. In order for the file to be run the last 5 characters must be .TEXT, .CODE, OR .DATA. Without these suffixes the file may be executed but not put in the workfile to be run. Lower-case letters will be translated to upper-case, and blanks and non-printing characters will be removed from the filename. Legal characters for filenames are the alphanumerics and the special characters '-', '/', '\', '_', and '.'. These special characters may be used to indicate hierarchic relationships among files and/or to distinguish several related files of different types.

WARNING: The I.5 Filer will not be able to access filenames containing the characters '\$', ':', '=', '?', and '.'. If files from previous versions of the system contain these characters, then they should be removed before attempting to use those files with the I.5 System.

The wildcard characters, '*' and '?', are used to specify subsets of the directory. The Filer performs the requested action on all files meeting the specifications. A file specification containing the subset-specifying string 'DOC=TEXT' notifies the Filer to perform the requested action on all files whose names begin with the string 'DOC' and end with the string 'TEXT'. If a '?' is used in place of an '*', the Filer requests verification before affecting each file meeting the specified criteria. Either or both strings may be empty. For example, a subset specification of the form '*<string>' or '<string>=' or even '*' is valid. This last case, where both subset-specifying strings are empty, is interpreted by the Filer to specify every file on the volume, so typing '*' or '?' alone causes the Filer to perform the appropriate action on every file in the directory.

Given an example directory for volume MYDISK:

NAUGHTYBITS	6	23-Jun-54
MOLD.TEXT	4	29-Jun-54
USELESS.CODE	10	19-May-54
MOLD.CODE	4	29-Jun-54
NEVERMORE.TEXT	12	5-Apr-54
GOONS	5	10-Sep-52

EXAMPLE:

Prompt: Remove what file?

Response: Typing 'N=' generates the message:

```
MYDISK: NAUGHTYBITS      removed
MYDISK: NEVERMORE.TEXT  removed
Update directory?
```

(At this point the user can type 'Y' to remove or type 'N', in which case the files will not be removed. The Filer always requests verification on any wildcard removes.)

Typing 'N?' generates the message:

Remove NAUGHTYBITS: ?

After the user types a response, the Filer asks:

Remove NEVERMORE.TEXT: ?

EXAMPLE:

Prompt: Dir listing of what vol ?

Response: Typing '=TEXT' causes the Filer to list

```
MOLD.TEXT      4  29-Jun-54
NEVERMORE.TEXT 12  5-Apr-54
```

The subset-specifying strings may not 'overlap'. For example, GOON=NS would not specify the file GOONS, whereas GOON=S would be a valid (although pointless) specification.

The size specification information is predominantly useful in the commands T(transfer section 1.2.5.11 and M(ake section 1.2.5.17.

1.2.5 COMMANDS AND USE

Type "F" at the Command level to enter the Filer and the following prompt is displayed:

Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

Typing '?' in response to this prompt displays more Filer commands:

Filer: B(ad-blks, E(xt-dir, K(rnch, M(ake, P(refix, V(ols, X(amine, Z(ero

The individual Filer commands are invoked by typing the letter found to the left of the parenthesis. For example, 'S' would invoke the Save command.

In the Filer, answering a Yes/No question with any character other than 'Y' constitutes a 'No' answer. Typing an <esc> will return the user to the outer level of the Filer.

For each command requiring a file specification, refer to the file specification diagram (Figure 2). In many cases, the entire file specification is not necessary, and in some cases, certain parts of the file specification are not valid. See the required command in the following section.

Whenever a Filer command requests a file specification, the user may specify as many files as desired, by separating the file specifications with commas, and terminating this 'file list' with a carriage return. Commands operating on single filenames will keep reading filenames from the file list and operating on them until there are none left. Commands operating on two filenames (such as C(hange and T(rans) will take file names in pairs and operate on each pair until only one or none remains. If one filename remains, the Filer will prompt for the second member of the pair. If an error is detected in the list, the rest of the list will be flushed.

~~TOP SECRET~~

Loads the designated file into the workfile.

The entire file specification is not necessary. If the volume ID is not given, the default disk is assumed. Wildcards are not allowed, and the size specification option is ignored.

Given the example directory:

```
FILERDOC2.TEXT
A. OUT. CODE
F5. TEXT
ABSURD. TEXT
HYTYPER. CODE
STASIS. TEXT
LETTER1. TEXT
ASSEM. DOC. TEXT
FILER. DOC. TEXT
STASIS. CODE
```

EXAMPLE:

Prompt: Get what file?

Response: STASIS

The Filer responds with the message

'Text and Code file loaded'

since both text and code file exist. Had the user typed 'STASIS.TEXT' or 'STASIS.CODE', the result would have been the same - both text and code versions would have been loaded. In the event that only one of the versions exists, as in the case of A.OUT, then that version would be loaded, regardless of whether text or code was requested. Typing 'A.OUT.TEXT' in response to the prompt would generate the message: 'Code file loaded'.

~~XXXXXXXXXX~~
Saves the workfile under the filename specified by the user.

The entire file specification is not necessary. If the volume ID is not given, the default disk is assumed. Wildcards are not allowed, and the size specification option is ignored.

EXAMPLE:

Prompt: Save as what file?

Response: Type a filename of 10 or less characters, observing the filename conventions in section 1.2.4 'FILES'. This causes the FILER to automatically remove any old file having the given name, and to save the workfile under that name. For example, typing "X" in response to the prompt causes the workfile to be saved on the default disk as X.TEXT. If a codefile has been compiled since the last update of the workfile, that codefile will be saved as X.CODE.

The FILER automatically appends the suffixes .TEXT and .CODE to files of the appropriate type. Explicitly typing AFILE.TEXT in response to the prompt will cause the FILER to save this file as AFILE.TEXT.TEXT. Any illegal characters in the filename will be ignored, with the exception of ':'. If the file specification includes volume id, the Filer assumes that the user wishes to save the workfile on another volume. For example, typing:

RED:EYE

in response to 'Save as what file?' will generate

Prompt: Would you like EYE.TEXT written to RED: ?

RED:EYE constitutes a file specification, and a 'Y' answer to this prompt will cause the Filer to attempt a transfer of the workfile to the specified volume and file. (see section 1.2.5.11 T(transfer.))

~~3) N(aw)~~

Clears the workspace (workfile).

No file specifications allowed.

If there is already a workfile present, the user is prompted:

Prompt: Throw away current workfile?

Response: 'Y' will clear the workfile while 'N' returns the user to the outer level of the FILER.

If <workfile name>.BACK exists, then the user is prompted:

Prompt: Remove <workfile name>.BACK ?

~~4) UN(aw)~~

Returns the user to the outermost command level.

No file specification allowed.

~~5) W(aw)~~

Identifies the name and state (saved or not) of the workfile.

No file specification allowed.

~~6) V(aw)~~

Lists volumes currently on-line, with their associated unit (device) numbers.

No file specification allowed.

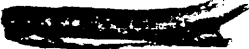
A typical display would be:

Volumes on-line:

1 CONSOLE:
2 SYSTEM:
3 GRAPHIC:
4 * MYDISK:
6 PRINTER:
8 REMOTE:
9 * BIG:

Prefix is - MYDISK:

The system or "boot-disk" volume's name is preceded by a '*'.

The system volume is the default volume unless the prefix (see P(refix) has been changed. Block-structured devices are indicated by '*' or '#'.


Lists a disk directory, or some subset thereof, to the volume and file specified (default is CONSOLE:).

The user may list any subset of the directory, using the 'wildcard' option, and may also write the directory, or any subset thereof, to a volume or filename other than CONSOLE. File specification will therefore be discussed in terms of source file specification and destination file specification.

Source file specification consists of a mandatory volume ID, and optional subset-specifying strings, which may be empty. If subset-specifying strings are used, then one of the wildcard characters must be used. A string (for example, the full filename STASIS.TEXT) may not be used as part of the source file specification unless a wildcard character is used!

Source file information is separated from destination file information by a comma (',').

Destination file specification consists of a volume ID, and, if the volume is a block-structured device, a filename. File size specifications will be ignored.

The most frequent use of this command is to list the entire directory of a volume. The following display, which represents a complete directory listing for the example disk MYDISK, would be generated by typing any valid volume ID for MYDISK (see Figure 2) in response to the prompt.

Dir listing of what vol?

```
MYDISK:
FILERDDC2.TEXT    28    1-Sep-78
A.OUT CODE        10    1-Sep-78
F5.TEXT           8     1-Sep-78
ABSURD            4     1-Sep-78
HYTYPER.CODE     12    1-Sep-78
STASIS TEXT       8     1-Sep-78
LETTER1.TEXT     18    1-Sep-78
ASSEMDDC.TEXT    20    1-Sep-78
FILERDDC1.TEXT   24    1-Sep-78
STASIS.CODE       6     1-Sep-78
10/10 files <listed/in-dir>, 130 blocks used, 364 unused
```

(The bottom line of the display informs the user that 10 files out of 10 files on the disk have been listed, that 130 disk blocks have been used, and that 364 disk blocks remain unused.)

EXAMPLE:

L(dir transaction involving wildcards:

Prompt: Dir listing of what vol ?

User response: #4:FIL=TEXT

generates the following display:

```
MYDISK:
FILERDDC2.TEXT    28    1-Sep-78
FILERDDC1.TEXT    24    1-Sep-78
2/10 files <listed/in-dir>, 52 blocks used, 364 unused
```

EXAMPLE:

L(dir transaction involving writing the directory subset to a device other than CONSOLE:

Prompt: Dir listing of what vol ?

User response: *FIL=TEXT,PRINTER: causes

```
MYDISK:
FILERDDC2.TEXT    28    1-Sep-78
FILERDDC1.TEXT    24    1-Sep-78
2/10 files <listed/in-dir>, 52 blocks used, 364 unused
```

to be written to the Printer.

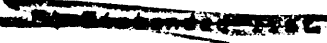
EXAMPLE:

L(dir transaction involving writing the directory subset to a block-structured device:

Prompt: Dir listing of what vol ?

User response: #4: FIL=TEXT, #5: TRASH creates the file TRASH on the volume associated with unit 5. TRASH would contain:

MYDISK:
FILERDOC2.TEXT 28 1-Sep-78
FILERDOC1.TEXT 24 1-Sep-78
2/10 files <listed/in-dir>, 52 blocks used, 364 unused



Lists the directory in more detail than the L(dir command.

All files and unused areas are listed along with (in this order) their block length, last modification date, the starting block address, the number of bytes in the last block of the file, and the filekind. All wildcard options and prompts are as in the L(dir command. An example display is shown below.

MYDISK:
FILERDOC2.TEXT 28 1-Sep-78 6 512 Textfile
A. OUT. CODE 10 1-Sep-78 34 512 Codefile
F5.TEXT 8 1-Sep-78 44 512 Textfile
<UNUSED> 10 52
ABSURD 4 1-Sep-78 62 512 Datafile
HYTYPER.CODE 12 1-Sep-78 66 512 Codefile
STASIS.TEXT 8 1-Sep-78 78 512 Textfile
LETTER1.TEXT 18 1-Sep-78 86 512 Textfile
ASSEMDOC.TEXT 20 1-Sep-78 104 512 Textfile
FILERDOC1.TEXT 24 1-Sep-78 124 512 Textfile
STASIS.CODE 6 1-Sep-78 148 512 Codefile
<UNUSED> 354 154
10/10 files <listed/in-dir>, 130 blocks used, 364 unused, 354 in largest area



Changes file or volume name.

This command requires two file specifications. The first of these specifies the file to be changed, the second, to what it will be changed. The first specification is separated from the second specification by either a <ret> or a comma (','). Any volume ID information in the second file specification is ignored, since obviously the 'old file' and the 'new file' are on the same volume! Size specification information is ignored.

Given the example file F5.TEXT, residing on the volume occupying unit 5:

Prompt : Change what file?

User Response: #5:F5.TEXT,HOOHAAH

changes the name in the directory from 'F5.TEXT' to 'HOOHAAH'. Although filekinds are originally determined by the filename, the C(hange command does not affect the filekind. In the above case, HOOHAAH would still be a text file. However, since the G(et command searches for the suffix '.TEXT' in order to load a text file into the workfile, HOOHAAH would need to be renamed HOOHAAH.TEXT in order to be loaded into the workfile.

Wildcard specifications are legal in the C(hange command. If a wildcard character is used in the first file specification, then a wildcard must be used in the second file specification. The subset-specifying strings in the first file specification are replaced by the analogous strings (henceforward called replacement strings) given in the second file specification. The Filer will not change the filename if the change would have the effect of making the filename too long (>15 characters). Given a directory of example disk NOTSANE: containing the files:

POEMS.TEXT
MAUNDER.TEXT
MALPRACTICE
MAKELISTS.TEXT

EXAMPLE:

Prompt : Change what file?

User response: NOTSANE:MA=TEXT,XX=GAACK
causes the Filer to report

NOTSANE:MAUNDER.TEXT changed to XXUNDER.GAACK
NOTSANE:MAKELISTS.TEXT changed to XXKELISTS.GAACK

The subset-specifying strings may be empty, as may the replacement strings. The Filer considers the file specification '=' (where both subset-specifying strings are empty) to specify every file on the disk. Responding to the C(hange prompt with '=,Z=Z' would cause every filename on the disk to have a 'Z' added at front and back. Responding to the prompt with 'Z=Z,=' would replace each terminal and initial 'Z' with nothing. Given the filenames:

THIS.TEXT
THAT.TEXT

EXAMPLE:

Prompt : Change what file?

User Response: T=T,=

The result would be to change 'THIS.TEXT' to 'HIS.TEX', and 'THAT.TEXT' to 'HAT.TEX'.

The volume name may also be changed by specifying a volume ID to be changed, and a volume ID to change to.

EXAMPLE:

Prompt : Change what file?

User Response: NOTSANE:,WRKDISK:

generates the message, NOTSANE: changed to WRKDISK:

Removes file entries from the directory.

This command requires one file specification for each file the user wishes to remove. Wildcards are legal. Size specification information is ignored. Given the example files (assuming that they are on the default volume):

AARDVARK.TEXT
ANDROID.CODE
QUINT.TEXT
AMAZING.CODE

EXAMPLE:

Prompt: Remove what file?

User Response: AMAZING.CODE

removes the file AMAZING.CODE from the volume directory. Note: To remove SYSTEM.WRK.TEXT and/or SYSTEM.WRK.CODE the N(ew command should be used, or the system may get confused.

As noted before, wildcard removes are legal.

EXAMPLE:

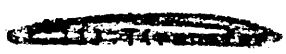
Prompt: Remove what file?

User Response: A=CODE

causes the Filer to remove AMAZING.CODE and ANDROID.CODE.
WARNING: Remember that the Filer considers the file specification '=' (where both subset-specifying strings are empty) to specify every file on the volume. Typing an '=' alone will cause the Filer to remove every file on your directory!! Fortunately, before finalizing any wildcard removes, the Filer prompts the user with

Prompt: Update directory?

Response: 'Y' causes all specified files to be removed. 'N' returns the user to the outer level of the Filer without any removes having occurred.



Copies the specified file to the given destination.

This command requires the user to type two file specifications, one for the source file, and one for the destination file, separated with either a comma or <ret>. Wildcards are permitted, and size specification information is recognized for the destination file.

Assume that the user wishes to transfer the file FARKLE.TEXT from the disk MYDISK to the disk BACKUP.

EXAMPLE:

Prompt: Transfer what file ?

User Response: MYDISK:FARKLE.TEXT

Prompt: To where?

(Note: On a one-drive machine, do NOT remove your source disk until you are prompted to insert the destination disk)

User Response: BACKUP:NAME.TEXT

Prompt: Put in BACKUP:
Type <space> to continue

The user should remove the source disk, insert the destination disk and type a <space>. The Filer then notifies the user:

MYDISK:FARKLE.TEXT transferred to BACKUP:NAME.TEXT

The Filer has made a copy of FARKLE and has written it to the disk BACKUP giving it the name NAME.TEXT. If the specified file is large, the user may be prompted to alternately insert the source and destination disks until the transfer is completed.

It is often convenient to transfer a file without changing the name, and without retyping the file name. The Filer enables the user to do this by allowing the character '\$' to replace the filename in the destination file specification. In the above example, had the user wished to save the file FARKLE.TEXT on BACKUP under the name FARKLE.TEXT, she could have typed:

MYDISK:FARKLE.TEXT, BACKUP:\$

WARNING: Please try to avoid typing the second file specification with the filename completely omitted! For example, a response to the Transfer prompt of the form:

MYDISK:FARKLE.TEXT, BACKUP:

generates the message:

Possibly destroy directory of BACKUP: ?

'Y' answer causes the directory of BACKUP to be wiped out!

Files may be transferred to volumes that are not block structured, such as CONSOLE: and PRINTER:, by specifying the appropriate volume ID (see Figure 1) in the destination file specification. A file name on a non-block-structured device is ignored. It is generally a good idea to make certain that the destination volume is on-line.

EXAMPLE:

Prompt: Transfer what file?

User Response: FARKLE.TEXT

Prompt: To where?

User Response: PRINTER:

causes FARKLE.TEXT to be written to the printer.

The user may also transfer from non-block-structured devices, providing they are input devices. Filenames accompanying a non-block-structured device ID are ignored.

The wildcard capability is allowed for T(transfer. If the source file specification contains a wildcard character, and the destination file specification involves a block-structured device, then the destination file specification must also contain a wildcard character. The subset-specifying strings in the source file specification will be replaced by the analogous strings in the destination file specification (henceforward known as replacement strings). Any of the subset-specifying or replacement strings may be empty. Remember that the Filer considers the file specification '=' to specify every file on the volume.

EXAMPLE:

Given the volume MYDISK containing the files PAUCITY, PARITY and PENALTY, and the destination ODDNAMZ:

Prompt: Transfer what file?

User Response: P=TY,ODDNAMZ:V=S

would cause the Filer to reply:

MYDISK:PAUCITY	transferred to ODDNAMZ:VAUCIS
MYDISK:PARITY	transferred to ODDNAMZ:VARIS
MYDISK:PENALTY	transferred to ODDNAMZ:VENALS

Using '=' as the source filename specification will cause the Filer to attempt to transfer every file on the disk. This will probably overflow the outputbuffer. (There are easier ways to transfer whole disks. If you wish to do this, please refer to the material in this section on volume- to- volume transfers.)

Using '=' as the destination filename specification will have the effect of replacing the subset-specifying strings in the source specification with nothing. A brief reminder: '?' may be used in place of '='. The only difference is that '?' causes the user to be asked for verification before the operation is performed.

A file can be transferred from a volume to the same volume by specifying the same volume ID for both source and destination file specifications. This is frequently useful when the user wishes to relocate a file on the disk. Specifying the number of blocks desired will cause the Filer to copy the file in the first-^{fit} area of at least that size. If no size specification is given, the file is written in the largest unused area.

If the user specifies the same filename for both source and destination on a same-disk transfer, then the Filer rewrites the file to the size-specified area, and removes the older copy.

EXAMPLE:

Prompt: Transfer what file?

User Response: #4: QUIZZES.TEXT, #4: QUIZZES.TEXT[20]

causes the Filer to rewrite QUIZZES.TEXT in the first 20-block area encountered (counting up from block 0) and to remove the previous version of QUIZZES.TEXT.

WARNING: Wildcard-type specifications do not always work very well on same-disk transfers. The results tend to be unpredictable, so these operations are not recommended.

It is also possible to do entire volume-to-volume transfers. The file specifications for both source and destination should consist of volume ID only. Transferring a block-structured volume to another block-structured volume causes the destination volume to be 'wiped out' so that it becomes an exact copy of the source volume.

Assume that the user desires an extra copy of the disk MYDISK: and is willing to sacrifice disk EXTRA:

EXAMPLE:

Prompt: Transfer what file?

User Response: MYDISK:, EXTRA:

Prompt: Possibly destroy directory of EXTRA: ?

WARNING: There's no 'possibly' about this! If the user types 'Y', the directory of EXTRA: will be destroyed! An 'N' response will return the user to the outer level of the Filer, and a 'Y' will cause EXTRA to become an exact copy of MYDISK. Often this is desirable for backup purposes, since it is relatively easy to copy a disk this way, and the volume name can be changed (see C(hng) if desired.

Although it is certainly possible to transfer a volume (disk) to another using a single disk-drive, it is a fairly tedious process, since the in-core transfer reads up the information in rather small chunks, and a great deal of disk juggling is necessary for the complete transfer to take place.

~~DATE~~

Lists current system date, and enables the user to change the date.

```
Prompt: Date Set: <1..31>-<JAN..DEC>-<00..99> DR <CR>
          Today is 19-Aug-78
          New date?
```

The user may enter the correct date in the format given. After typing <ret>, the new date will be displayed. Typing only a return does not affect the current date. The hyphens are delimiters for the day, month and year fields, and it is possible to affect only one or two of these fields. For example, the year could be changed by typing '--79', the month by typing '-Sep', etc. The entire month-name can be entered, but will be truncated by the Filer. Slash ('/') is also acceptable as a delimiter. The most common input will be a single number, which will be interpreted as a new day. For example, if yesterday was the 19th of August, the user would want to type D20<ret>, which would have the desired effect of changing the date to the 20th of August. The day-month-year order is inviolate, however.

This date will be associated with any files saved during the current session and will be the date displayed for those files when the directory is listed.

~~TOPTION~~

Changes the current default to the volume specified.

This command requires the user to type a volume ID. An entire file specification may be entered, but only the volume ID will be used. It is not necessary for the specified volume to be on-line.

To determine the current default volume, the user may respond to the prompt with ':'.

~~Bad Blocks~~

Scans the disk and detects bad blocks.

This command requires the user to type a volume ID. The specified volume must be on-line.

Prompt: Bad blocks scan of what vol?

Response: <volume ID>

Checks each block on the indicated volume for errors and lists the number of each bad block. Bad blocks can often be fixed or marked (see eX(amine)).

~~Fix Bad Blocks~~

Attempts to physically recover suspected bad blocks.

This command requires the user to type a volume ID. The volume must be on-line.

EXAMPLE:

Prompt : Examine blocks on what volume?

Response : <volume ID> generates the

Prompt: Block number-range ?

The user should have just done a bad block scan, and should enter the block number(s) returned by the bad block scan. If any files are endangered, the following prompt should appear:

Prompt: File(s) endangered:
<filename>
Try to fix them?

Response: 'Y' will cause the FILER to examine the blocks and return either of the messages:

Block <block-number> may be ok

in which case the bad block has probably been fixed, or
Block <block-number> is bad

in which case the FILER will offer the user the option of marking the block(s) BAD. Blocks which are marked BAD will not be shifted during a K(runch, and will be rendered effectively harmless.

An 'N' response to the 'fix them?' prompt returns the user to the outer level of the FILER.

WARNING: A block which is 'fixed' may contain garbage. 'May be ok' should be translated as 'is probably physically ok'. Fixing a block means that the block is read, is written back out to the block and is read again. If the two reads are the same, the message is 'may be ok'. In the event that the reads are different, the block is declared bad and may be marked as such if so desired.

K(runch

Moves the files on the specified volume so that unused blocks are combined at the 'end' of the disk.

This command requires the user to type a volume ID. The specified volume must be on-line. It is strongly recommended that the user perform a bad block scan of the volume before K(runching in order to avoid writing files over bad areas of the disk. If bad blocks are encountered, they must be either fixed or marked before the K(runch (see eX(amine).

As each file is moved, its name is reported to the console. If SYSTEM.PASCAL is moved, the system must be reinitialized by bootstrapping. Do not touch the disk, the boot-switch or the disk-drive door until K(runch tells you it has completed its task.

EXAMPLE:

Prompt : Crunch what vol?

Response : <volume ID>

causes Filer to prompt with:

Prompt : Are you sure you want to crunch <volume ID>?

Response: 'Y' initiates the K(runch. Typing an 'N' will return the user to the outer level of the FILER.

~~XXXXXXXXXX~~
Creates a directory entry with the specified filename.

This command requires the user to type a file specification. Wildcard characters are not allowed. The file size specification option is extremely helpful, since, if it is omitted, the Filer creates the specified file by consuming the largest unused area of the disk. The file size is determined by following the filename with the desired number of blocks, enclosed in square brackets '[' and ']'. Some special cases are:

[O] - equivalent to omitting the size specification. The file is created in the largest unused area.

[*] - the file is created in the second largest area, or half the largest area, whichever is larger.

EXAMPLE:

Prompt : Make what file?

Response : MYDISK:FARKLE.TEXT[28]

Creates the file FARKLE.TEXT on the volume MYDISK: in the first unused 28-block area encountered.

~~XXXXXXXXXX~~
Reformats the specified volume. The previous directory is rendered irretrievable.

EXAMPLE:

Prompt: Zero dir of what vol ?

Response: <volume ID>

Prompt: Destroy <volume name> ?

Response: A 'Y' response generates

Prompt: Duplicate dir ?

Response: If a 'Y' is typed, then a duplicate directory will be maintained. This is advisable because, in the event that the disk directory is destroyed, a utility program called COPYDUPDIR can use the duplicate directory to restore the disk.

Prompt: <current number of blocks on disk> blocks ?

Response: 'N' generates

Prompt: # of blocks ?

Response: User will type number of blocks desired. The table following this section gives the correct number of blocks for several types of disks.

'Y' generates

Prompt: New vol name ?

Response: User types any valid volume name.

Prompt: <new volume name> correct ?

Response: 'Y' causes the Filer to respond with the message:

<new volume name> zeroed

MACHINE	DISK TYPE	# OF BLOCKS
Terak	Single-density soft-sectored 8" floppy	494
Northwest Micro	Double-density soft-sectored 8" floppy	1102
<i>Shugart</i> SA-400		600 <u>594</u> (6 blocks for directory)
Zilog	Single-density hard-sectored 8" floppy	608
North Star	Double-density hard-sectored 5 1/4" floppy	167
DEC	RK05 - per platter	4872

- Notes -

To find X)ecute PRINT + then follow further
commands

move 1 bit on cursor
right 2 bits on

* SCREEN ORIENTED EDITOR * * Section 1.3.1 *

Version I.5 September 1978

This introduction, which describes the idea behind the Editor, is the first of four sections. The second section is a tutorial for the novice. While the Editor is designed to handle any files, the tutorial section uses a sample program to demonstrate how to use the most basic commands to modify a file. The third section contains a detailed description of each command, with examples, and the fourth is for quick reference.

THE CONCEPT OF A 'WINDOW' INTO THE FILE

The Screen Oriented Editor is specifically designed for use with Video Display Terminals. On entering any file, the Editor displays the start of the file in the upper left hand corner of the screen. If the file is too long for the screen, only the first portion is displayed. This is the concept of a 'window'. The whole file is there and is accessible by Editor commands, but only a portion of it can be seen through the 'window' of the screen. When any Editor command takes the user to a position in the file which is not displayed, the "window" is updated to show that portion of the file.

THE CONCEPT OF A CURSOR

The cursor represents the exact position in the file and can be used to move to any position. The window shows that portion of the file near the cursor. To see another portion of the file, move the cursor. Action always takes place at the cursor. Some of the commands permit additions, changes or deletions of such length that the screen cannot hold the whole portion of the text that has been changed. In those cases, the portion of the screen where the cursor stopped is displayed. In no case is it necessary for the user to operate on portions of the text not seen on the screen, but in some cases it is optional.

THE CONCEPT OF A PROMPT LINE

The Editor displays a prompt line as the top line of the screen in order to remind the user of the current mode and the options available for that mode. Only the most commonly used options appear on the prompt line as the following display shows:

>Edit: A(djust C(py D(lete F(ind I(nsert J(mp R(place G(uit X(chng Z(ap [E.6]

NOTATION

The notation used in this section corresponds to the notation used to prompt the user in the editor. Any input that is enclosed between a < and > is requesting that a particular key be used, not that the particular word be typed out. For example, <RET> means that the return key should be typed at that point. When a particular sequence of key strokes is required they will be contained within quotes. For example, "FILENAME", <RET> refers to the typed sequence "FILENAME" followed by typing the return key. Lower or upper case may be used when typing Editor commands

* GETTING STARTED * * Section 1.3.2 *

ENTERING THE WORKFILE AND GETTING A PROGRAM.

On entering the Editor :

No workfile is present. File? (<ret> for no file) appears.

There are two ways to answer this question :

1) With a name, for example "STRING1 <ret>". The file named STRING1 will now be retrieved. The file STRING1 could contain a program, also called STRING1, as in Fig. 2.1. After typing the name, a copy of the text of the first part of the file appears on the screen. Figure 2.1

```
PROGRAM STRING1;  
BEGIN  
  WRITE('TOO WISE');  
  WRITE('YOU ARE');  
  WRITELN(',');  
  WRITELN('TOO WISE');  
  WRITELN('YOU BE')  
END.
```

2) With a <return>. This implies that a new file is to be started. The only thing visible on the screen after doing this is the editor prompt line. A new workfile is opened and currently has nothing in it. Type "I" to begin inserting a program or text.

Workfiles: No questions are asked if a workfile already exists. The workfile is displayed and can be modified or can be cleared, in order to start a file, by using the N)ew command in the Filer.

In order to edit, it is necessary to move the cursor. On the keyboard are four keys with arrows, (which may look like triangles), which move the cursor. The <up-arrow> moves the cursor up one line, the <right-arrow> moves the cursor right one space and so forth.

The cursor does not like to be outside of the text of the program. For example, after the "N" in "BEGIN" in Fig. 2.2, push the <right-arrow> and the cursor moves to the "W" in "WRITE". Similarly at the "W" in "WRITE('TOO WISE ');", use <left-arrow> to move to after the "N" in "BEGIN".

Figure 2.2

```
BEGIN_
WRITE('TOO WISE ');

BEGIN
WRITE('TOO WISE ');
```

If it is necessary to change the "WRITE('TOO WISE ');" found in the third line to a "WRITE('TOO SMART ');", the cursor must first be moved to the right spot.

For example: if the cursor is at the "P" in "PROGRAM STRING1;", go down two lines by pressing the down arrow 2 times. To mark the positions the cursor occupies, labels a,b,c are used in Fig. 2.3. "a" is the initial position of the cursor; "b" is where the cursor is after the first <down-arrow>; "c", after the second <down-arrow>.

Figure 2.3

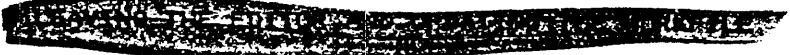
```
aPROGRAM STRING1
bBEGIN
cWRITE('TOO WISE ');
```

Now, using the right arrow, move until the cursor sits on the "W" of "WISE". Note that with the use of <down-arrow> the cursor appears to be outside the text. Actually it is at the "W" in "WRITE", so do not be surprised when on typing the first <left-arrow> the cursor jumps to the "R" in "WRITE".

The Edit level prompt line shows that to I(nsert) an item, type "I". The cursor must be in the correct position before typing "I". Earlier, the cursor was moved to the "W" in "TOO WISE"; now, on typing "I", an insertion will be made before the "W". The rest of the line from the point of insertion will be moved to the right hand side of the screen. In the event that the insertion is lengthy, that

It is legal to delete a carriage return. At the end of the line, enter DELETE mode, and <space> until the cursor moves to the beginning of the next line.

These are sufficient commands to edit any file desired. The next section describes many more commands in the Editor which make editing easier.

 When all the changes and additions have been made, exit the Editor and "save" a copy of the modified program. This is done by typing "G" which will cause the prompting display shown in Fig. 2.7.

by going into the Filer or by doing a Write directly.

Figure 2.7

>Quit:

U(pdate the workfile and leave
E(xit without updating
R(eturn to the editor without updating
W(rite to a file name and return

The most elementary way to save a copy of the modified file on disk is to type "U" for U(pdate which causes the workfile to be saved as SYSTEM.WRK.TEXT. With the workfile thus saved, it is possible to use the R(un command, provided of course the file is a program. It is also possible to use the S(ave option in the Filer to save the modified file in the library before using the Editor to modify or create another file.

Miscellaneous commands, in the next section, explains in greater detail the options available at >Quit.

* DETAILED DESCRIPTION OF COMMANDS * * Section 1.3.3 *

At the Edit level there are many options, some of which are referred to as commands and some as modes depending upon the appearance of the prompt. If an option executes a task and returns control to the Edit level, that option is called a command. If an option issues a prompt and gives the user another level of options, it is called a mode. On entering or returning to the Edit level, the Editor redisplay the "Edit:" prompt line.

Many of the commands allow repeat-factors. A repeat-factor is applied to a command by typing a number immediately before issuing the command which is then repeated for the number of times indicated by the repeat-factor. For example: typing "2 <down-arrow>" will cause the <down-arrow> command to be executed twice, moving the cursor down two lines. Commands which allow a repeat-factor assume the repeat-factor to be 1 if no number is typed before the command. A '/' typed before the command implies an infinite number.

It should be pointed out that the cursor is never really "at" a character. The cursor is only allowed to be "between" characters. For instance, if the cursor looks as though it is at the letter "R", it is actually between the letter "R" and the letter in front of it. This is noticed most clearly on the insert command as it inserts in front of the character the cursor was "at". On the screen the cursor is placed "at" "R" to make it easier to display.

Certain commands are affected by direction. If the direction is forward, then they operate forward through the file, that being the standard direction of reading English. Backwards is the reverse direction. When direction affects the command it is specifically noted.

<down-arrow>	Moves down
<up-arrow>	Moves up
<right-arrow>	Moves right
<left-arrow>	Moves left
"<" or "," or "--"	Changes the direction to backward
">" or " ." or "+"	Changes the direction to forward
<space>	Moves direction
<back-space>	Moves left
<tab>	Moves direction to the next position which is a multiple of 8 spaces from the left side of the screen
<return>	Moves to the beginning of the next line

The arrow, "<" or ">", in front of the prompt line always indicates direction; "<" for backward and ">" for forward. On entering the Editor, the direction is forward. The direction can be changed by typing the appropriate command whenever the "Edit:" prompt line is present. The period and the comma can also be used because on many standard keyboards, "." is lower-case for ">" and "," is the lower-case for "<".

Repeat-factors can be used with any of the above commands.

For user convenience, the Editor maintains the column position of the cursor when using <up-arrow> and <down-arrow>. When the cursor is outside the text, the Editor treats the cursor as though it were immediately after the last character, or before the first, in the line.

JUMP mode is reached by typing "J" for J(mp while at the Edit level. On entering JUMP mode the following prompt line appears:

ⓀJUMP: B(eginning E(nd M(arker <esc>

Typing "B" (or "E") moves the cursor to the beginning (or the end) of the file, displays the edit prompt line and the first (or last) page of the file. Typing "M" causes the Editor to display the prompt line:

Jump to what marker?

The name of the marker must be entered followed by a <return>. The Editor will then move the cursor to the place in the file with that name. If the marker is not in the file the Editor will display:

ERROR: Marker not there. Please press <space bar> to continue.

The instructions for setting a marker are detailed in SET under Miscellaneous commands.

PAGE command is executed by typing "P" while at the Edit level. Depending on the direction of the arrow at the beginning of the prompt line, PAGE command moves the cursor one whole screenful up or down. The cursor always moves to the start of the line. A <repeat-factor> may be used before this command for moving several pages.

EQUALS command is executed by typing "=" while at the Edit level. It causes the cursor to jump to the beginning of the last section of text which was inserted, found or replaced from anywhere in the file. Equals works from anywhere in the file and is not direction sensitive. An INSERT, FIND or REPLACE cause the absolute position of the beginning of the insertion, find or replacement to be saved. Typing "=" causes the cursor to jump to that position. If a copy or a deletion has been made between the beginning of the file and that absolute position, the cursor will not jump to the start of the insertion as that absolute position will no longer be correct.

TEXT CHANGING COMMANDS

~~mode~~ mode
INSERT mode is reached by typing "I" for "I(nsert)" while at the Edit level. On entering INSERT mode the following prompt line appears:

>Insert: Text (<bs> a char, a line) [*<etx>* accepts, <esc> escapes]

One of the options here is to type in text followed by <esc> or <etx>. It is possible to delete a character without leaving the INSERT mode by back-spacing over it. To delete the entire line just typed, type . The INSERT prompt line indicates these by "<bs> a char" and " a line".

Typing <return> INSERT starts a new line at the level of indentation specified by the options turned on in Environment section of the SET mode. See the section on the SET mode in order to set these options.

~~option~~ option
If Auto-indent is True, a <return> causes the cursor to start the next line with an indentation equal to the indentation of the line above. If Auto-indent is False, a <return> returns the cursor to the first position in the next line. Note: if Filling is True, the first position is the Left-margin.

~~option~~ option
If Filling is True, the Editor forces all insertions to be between the right and left margins by automatically inserting <return>'s between "words" whenever the right margin would have been exceeded and by indenting to the Left-margin whenever a new line is started. The Editor considers anything between two spaces or between a space and a hyphen to be a word.

If both Auto-indent and Filling are True, Auto-indent controls the Left-margin while Filling controls the Right-margin. The level of indentation may be changed by using the <space> and <backspace> keys immediately after a <return>. Important: This can only be done immediately after a <return>.

Example 1: With Auto-indent true, the following sequence creates the indentation shown in Figure 3.1.

"ONE", <return>, <space>, <space>, "TWO",
<return>, "THREE", <return>, <backspace>, "FOUR".

Figure 3.1

ONE	Original indentation
TWO	Indentation changed by <space> <space>
THREE	<return> causes auto-indentation to level of line above
FOUR	<backspace> changes indentation from level of line above

Example 2: With Filling True (and Auto-indent False) the following sequence creates the indentation shown in Figure 3.2:

"ONCE UPON A TIME THERE- WERE".

(Very narrow margins have been used for simplicity.)

Figure 3.2

ONCE UPON A	Auto-returned when next word would exceed margin
TIME THERE-	Auto-returned at hyphen
WERE	

^
Level of left margin

Filling also causes the Editor to adjust the margins on the portion of the paragraph following the insertion. Any line beginning with the Command character (see SET mode) is not touched when filling does this adjustment and that line is considered to terminate the paragraph.

The direction does not affect the INSERT mode, but is indicated by the direction of the arrow on the prompt line.

If an insertion is made and accepted, that insertion is available for use in the COPY mode. However, if <esc> is used, there is no string available for COPY.

DELETE mode is reached by typing "D" for "D(lete)" while at the Edit level. On entering DELETE mode the following prompt line appears:

>Delete: < > <Moving commands> {<etx> to delete, ,<esc> to abort}

In order to delete, the cursor must be in position at the first character to be deleted. On typing "D" and entering DELETE, the Editor remembers where the cursor is. That position is called the anchor. As the cursor is moved from the anchor position using the normal moving commands, text in its path will disappear. To accept the deletion, type <etx>; to escape, type <esc>.

When <etx> is typed, the Editor saves everything which was deleted for COPY to use; but if <esc> is typed, the copy buffer is empty.

Example:

In Figure 3.3:

- 1) Move the cursor to the "E" in END.
- 2) Type "<" (This changes the direction to backward)
- 3) Type "D" to enter DELETE mode.
- 4) Type <ret> <ret>. After the first return the cursor moves to before the "W" in WRITELN and "WRITELN('TO BE. ');" disappears. After the second return the cursor is before the "W" in WRITE and that line has disappeared.
- 5) Now press <etx>. The program after deletion appears as is shown in Figure 3.4.

The two deleted lines have been stored in the copy buffer and the cursor has returned to the anchor position. Now use the COPY routine to copy the two deleted lines at any place to which the cursor is moved.

Figure 3.3

```
-----  
PROGRAM STRING2;  
BEGIN  
  WRITE('TOO WISE ');  
  WRITELN('TO BE. ');  
END.  
-----
```

Figure 3.4

```
-----  
PROGRAM STRING2;  
BEGIN  
END.  
-----
```

The <repeat-factor> may also be used to delete several lines as once by prefacing a <return> or any other of the moving commands with a <repeat-factor> while in delete mode.

The ZAP command is executed by typing "Z" for Zap while at the Edit level. This command deletes all text between the start of what was previously found, replaced or inserted and the current position of the cursor. This command is designed to be used immediately after one of the FIND, REPLACE or INSERT commands. If more than 80 characters are being zapped the editor will ask for verification.

Repeat-factors and Zap: If a FIND or a REPLACE is made with a repeat factor and then ZAP, only the last find or replacement will be zapped. All others will be left as found or replaced.

Whatever was deleted by using the ZAP command is available for use with the COPY command.

The COPY command is executed by typing "C" for C(py while at the Edit level.

On entering the Copy mode the following prompt line is displayed:

>COPY: B(uffer F(ile <esc>

To copy text from another file, type "F" and another prompt will appear:

{ >COPY: FROM WHAT FILE[MARKER, MARKER]?

Any file may now be specified, .text is assumed. In order to copy part of a file, two markers can be set to bracket the desired text. If [,marker] or [marker,] is used, the file will be copied from the start to the marker or from the marker to the end. On completion of the copy command (from file), the cursor returns to the beginning of the text just copied from the file. Use of the copy command does not change the contents of the file being copied from.

To copy the text in the copy buffer, type "B" and the Editor immediately copies the contents of the copy buffer into the file at the location of the cursor when "C" was typed. On the completion of the copy command the cursor returns to immediately before the text which was copied. Use of the copy command does not change the contents of the copy buffer.

The copy buffer is affected by the following commands:

1)DELETE: On accepting a deletion, the buffer is loaded with the deletion; on escaping from a deletion the buffer is loaded with what would have been deleted.

2)INSERT: On accepting an insertion the buffer is loaded with the insertion; on escaping from an insertion the copy buffer is empty.

3)ZAP: If the ZAP command is used the buffer is loaded with the deletion.

The copy buffer is of limited size. Whenever the deletion is greater than the buffer available, the Editor will issue a warning upon typing <etx> with the line:

There is no room to copy the deletion. Do you wish to delete anyway? (y/n)

EXCHANGE mode is reached by typing "X" while at the Edit level. On entering EXCHANGE mode the following prompt line appears:

>eXchange: TEXT {<bs> a char} [<esc> escapes; <etx> accepts]

EXCHANGE mode replaces one character in the file for each character of text typed. For example in the file in Figure 3.5 with the cursor at the "W" in WISE, typing "X", followed by typing "SM" will replace the "W" with the "S" and then the "I" with the "M" leaving the line as shown in Figure 3.6 with the cursor before the second "S".

Figure 3.5


WRITE('TOO WISE ');

Figure 3.6

WRITE('TOO SMSE ');

Typing a <back-space> (<bs>) will back the cursor one character and cause the original character in that position to reappear. As with most other commands, when in EXCHANGE mode, <esc> leaves the mode without making any of the changes indicated since entering the mode, while <etx> makes the changes part of the file.

Note: Exchange does not allow typing past the end of the line or typing in a carriage return.

 In both modes the use of a <repeat-factor> is valid and must be typed before typing "F" or "R". The <repeat-factor> appears in brackets on the prompt line.

Strings: Both modes operate on delimited strings. The Editor has two string storage variables. One, called <targ> by the prompt lines, is the target string and is referred to by both commands while the other, called <sub> by the prompt line, is the substitute and is used only by REPLACE. The following rules apply to both these strings.

Delimiters: Both delimiters of the string will be the same. For example: When in REPLACE mode the following command is valid and will replace the first occurrence of the character "[" with the character "]": "<[<]>". Here "<" and ">" are the delimiters.

The Editor considers any character which is not a letter or a number to be a delimiter. <space> is a particularly common delimiter.

Direction: Both modes operate from the position of the cursor to scan the text in the direction indicated by the arrow on the prompt line. The target pattern can only be found if it appears in that section of the text. See the section on direction on order to change the arrow.

Literal and Token mode: In Literal mode, the Editor will look for any occurrences of the target string. If you are in Token mode the Editor will look for isolated occurrences of the target string. The Editor considers a string isolated if it is surrounded by any combination of delimiters. For example, in the sentence "Put the book in the bookcase.", using the target string "book", literal mode will find two occurrences of "book" while token mode will find only one, the word "book" isolated by the delimiters <space> <space>.

To use token mode, type "T" after the prompt line and before the target string; to use literal mode, type "L". The default value found in the Environment may be over-ridden by typing "L" or "T" as appropriate. Token mode ignores spaces within strings so that both "(, ')" and "((, ')" are considered to be the same string.

The Same option: In both commands typing "S" indicates to the Editor that it is to use the same string as used previously. For example, typing "RS/<any-string>/" causes the REPLACE mode to use the previous target string, while typing "R/<any-string>/S" causes the previous substitute string to be used.

FIND mode is reached by typing "F" while at the Edit level. On entering Find mode one of the prompt lines in Figure 3.7 appears.

Figure 3.7

```
-----  
>Find[1]: L(it <target> =>
```

```
>Find[1]: T(ok <target> =>  
-----
```

The FIND mode finds the n-th occurrence of the <target> string starting with the current position and moving in the direction shown by the arrow at the beginning of the prompt line. The number "n" is the <repeat-factor> and is shown on the prompt line in the brackets "[]".

Example 1: In the STRING1 program with the cursor at the first "P" in PROGRAM STRING1 type "F". When the prompt appears type "'WRITE'". The single quote marks MUST be typed. The prompt line should now appear as:

```
>Find[1]: L)it <target> =>'WRITE'
```

After typing the last quote mark the cursor jumps to immediately after the "E" in the first WRITE.

Example 2: In the STRING1 program with the cursor at the "E" of "END." type "<" "3" "F". This will find the 3rd ("3") pattern in the reverse ("<") direction. When the prompt line appears type /WRITELN/. The prompt line should read:

```
<Find[3]: L)it <target> =>/WRITELN/
```

The cursor will move to immediately after the "N" in WRITELN.

Figure 3.8

```
PROGRAM STRING1;  
BEGIN  
  WRITE('TOO WISE ');  
  WRITE('YOU ARE');  
  WRITELN(',');          (*CURSOR FINISHES IN THIS LINE*)  
  WRITELN('TOO WISE ');  
  WRITELN('YOU BE. ');  
END.                      (*CURSOR STARTS IN THIS LINE*)
```

Example 3: On the first find we type "F/WRITE/". This locates the first "WRITE". Now typing "FS" will make the prompt line flash:

```
>Find[1]: L)it <target> =>S
```

and the cursor will appear at the second WRITE.

REPLACE mode is reached by typing "R" while at the Edit level. On entering REPLACE mode one of the two prompt lines in Figure 3.9 appears. In this example, a <repeat-factor> of four is assumed.

Figure 3.9

```
>Replace[4]: L(it V(fy <targ> <sub> =>  
>Replace[4]: T(ok V(fy <targ> <sub> =>
```

Example 1: Type "RL/QX//YZ/" which make the prompt line appear as:

```
>Replace[1]: L)it V)fy <targ> <sub> =>L/QX//YZ/
```

This command will change: "VAR SIZEGX: INTEGER;" to "VAR SIZEYZ: INTEGER;". Literal mode is necessary because the string GX is not a token but is part of the token SIZEGX.

Example 2: In Token mode REPLACE ignores spaces between tokens when looking for patterns to replace. For example, using the lines on the left hand side of Figure 3.10 and typing: "2RT/(', ')/.LN." The prompt line should appear as:

```
>Replace: L)it V)fy <targ> <sub> =>/(', ')/.LN.
```

Immediately after the last period was typed those two lines would change to those on the right hand side.

Figure 3.10

```
WRITE(' ');  
WRITE(' ');
```

```
WRITELN;  
WRITELN;
```

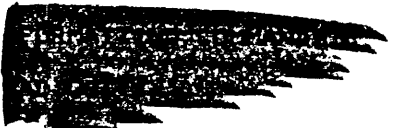
V)fy: The verify option permits examination of the <targ> string (up to the limit set by the repeat factor) and deciding if it is to be replaced. The following prompt line appears whenever REPLACE mode has found the <targ> pattern in the file and verification has been requested:

```
>Replace: <esc> aborts, 'R' replaces, ' ' doesn't
```

Typing an "R" at this point will cause a replacement while typing a space will cause the REPLACE mode to search for the next occurrence provided the <repeat-factor> has not been reached. The <repeat-factor> counts the number of times an occurrence is found, not the number of times you actually type "R". Use "/" as a <repeat-factor> in order to replace every occurrence of the target string. Once the Editor can no longer find the target string, the prompt:

```
ERROR: Pattern not in the file Please press <spacebar> to continue.
```

appears.



ADJUST mode is reached by typing "A" while at the Edit level of Command. On entering ADJUST mode the following prompt line appears:

```
>Adjust: L(just R(just C(enter <left,right,up,down-arrows> (<etx> to leave)
```

The ADJUST mode is designed to make it easy to adjust the indentation. On any line the <right-arrow> and <left-arrow> commands move the whole line. Each time a <right-arrow> is typed the whole line moves one space to the right. Each <left-arrow> moves it one to the left. When the line is adjusted to the desired indentation press <etx>. After pressing <etx>, <esc> cannot be used.

In order to adjust a whole sequence of lines, adjust one line, then use <up-arrow> (<down-arrow>) commands and the line above (below) will be automatically adjusted by the same amount.

Repeat-factors are valid when used before any of the <arrow> commands while in ADJUST mode.

ADJUST mode can also center or justify text. Typing "L" while in ADJUST mode will cause the line to be left-justified to the margin set in the Environment. Similarly typing "R" right-justifies to the set margin and typing "C" will cause the line to be centered between the set margins. Typing <up-arrow> (or <down-arrow>) will cause the line above (below) to be adjusted to the same specification (left-justified, right-justified or centered) as the previously adjusted line.

MARGIN command is executed by typing "M" while at the Edit level. MARGIN is an Environment dependent command, that is, it may only be executed when Filling is set to True and Auto-indent is set to False. The prompt for the MARGIN command does not appear on the ">Edit:" line.

There are three parameters used by the command: Right-margin, Left-margin and Paragraph-margin. MARGIN deals with one paragraph and realigns the text to compress it as much as possible without violating the above three margins. See the Environment option under the SET mode for how to set the margin values.

Example: The paragraph in Figure 3.13 has been MARGINED with the parameters on the left while the same paragraph in Figure 3.14 has been MARGINED with the parameters on the right.

Left-margin 0
Right-margin 72
Paragraph-margin 3

Left-margin 10
Right-margin 70
Paragraph-margin 0

Figure 3.13

This quarter, the equipment is different, the course materials are substantially different, and the course organization is different from previous quarters. You will be misled if you depend upon a friend who took the course previously to orient you to the course.

Figure 3.14

This quarter, the equipment is different, the course materials are substantially different, and the course organization is different from previous quarters. You will be misled if you depend upon a friend who took the course previously to orient you to the course.

A paragraph is defined to be something occurring between two blank lines. To MARGIN a paragraph move the cursor to anywhere in that paragraph and type "M". When doing an exceptionally long paragraph it may take several seconds before the routine is ready to redisplay the screen.

Portions of the text can be protected from being MARGINED by the use of the Command character. If the Command character appears as the first non-blank character in a line then that line is protected from the MARGIN command. The MARGIN command treats a line beginning with the command character as though it were a blank line, that is, it will consider that line to terminate (begin) the paragraph. Warning: Do not use the MARGIN command when in a line beginning with the Command character.

SET mode is entered by typing "S" while at the Edit level. The prompt for the SET command does not appear on the ">Edit:" prompt line due to space limitations. On entering the SET mode the following prompt line appears:

>Set: M(arker E(nvironment <esc>

M(arker:

When editing, it is particularly convenient to be able to jump directly to certain places in a long file by using markers set in the desired places. Once set, it is possible to jump to these markers using the M(arker option in the JUMP mode. When in the SET mode, type "M" for M(arker and the following prompt line appears:

Name of marker?

The name may be up to 8 characters followed by a <return>. Marker names are case sensitive so that lower and upper cases of the same letter are considered to be different characters. The marker will be entered at the position of the cursor in the text; therefore, first move the cursor to the desired position before setting the marker. (If the marker already existed, it will be reset.)

Only 10 markers are allowed in a file at any one time. If on typing "SM", the prompt:

Figure 3.15

```
Marker ovflw.
Which one to replace.
0) name1
1) name2
. ....
. ....
9)name10
```

appears, it is necessary to eliminate one in order to replace it. Choose a number 0 thru 9, type that number and that space will now be available for use in setting the desired marker.

If a copy or deletion is made between the beginning of the file and the position of the marker, the marker will not subsequently return to the desired place as the absolute position has changed.

The Editor enables the user to set the environment which the user determines to be most convenient for the editing being done. When in the SET mode type "E" for E(nvironment, the screen display is replaced with the following prompt shown in Figure 3.16.

Figure 3.16

```
>Environment: {options} <etr> or <sp> to leave
A(uto indent   True
F(illing       False
L(ef t margin  0
R(igh t margin 79
P(ara margin   5
C(ommand ch    ^
T(oken def     True

7436 bytes used, 12020 available
```

```
Patterns:
  <target>= 'xyz', <subst>= 'abc'
```

By typing the appropriate letter, any or all of the options may be changed. The options shown are the default options for the Editor on the Terak BS10A. Implementations for other machines may have different defaults.

Auto-indent affects only the INSERT mode of the Editor. Auto-indent is set to True (turned on) by typing "AT" and to False (turned off) by typing "AF".

Filling affects the INSERT mode and allows the MARGIN command to function. Filling is set to True (turned on) by typing "FT" and to False by typing "FF".

L(left margin
R(right margin
P(ara margin:

When Filling is True the margins set in the Environment are the margins which affect the INSERT mode and the MARGIN command. They also affect the Center and justifying commands in the ADJUST mode. To set the Left-margin, type "L" followed by a positive integer and a <space>. The positive integer typed should replace the old value for the L(left margin in the prompt shown in Figure 3.16. All positive integers with less than four digits are valid margin values.

The Command character affects the MARGIN command and the Filling option in the INSERT mode as described in those sections. Change Command characters by typing "C" followed by any character. For example typing "C","*" will change the Command character to "*". This change will be reflected in the prompt.

This option affects FIND and REPLACE. Token is set to True by typing "TT" and to False by typing "TF". If Token is True, Token is the default and if Token is False, Literal is the default.

The VERIFY command is executed by typing "V" while at the Edit level. The status of the Editor is verified by displaying the updated screen. The Editor attempts to adjust the window so that the cursor is at the center of the screen.

QUIT mode is reached by typing "Q" while at the Edit level. On entering QUIT mode the screen display is replaced by the following prompt:

Figure 3.17

>Quit:
U(pdate the workfile and leave
E(xit without updating
R(eturn to the editor without updating
W(rite to a file name and return

One of the four options must be selected by typing U, E, R or W.

This causes the Editor to write the file just modified into the workfile and store it as SYSTEM.WRK.TEXT. It is available for either the Compile or Run options or for the Save option in the Filer. The Filer treats SYSTEM.WRK.TEXT as text file.

This causes the Editor to leave without making any changes in SYSTEM.WRK.TEXT. This means that any modifications made since entering the Editor are not recorded in the permanent workfile.

This option returns to the Editor without updating. The cursor is returned to the exact place in the file it occupied when "Q" was typed. Usually this command is used after unintentionally typing "Q".

This option puts up a further prompt:

Figure 3.18

>Quit:
Name of output file (<cr> to return) -->

The modified file may now be written to any file name. If it is written to the name of an existing file, the modified file will replace the old file. This command can be aborted by typing <return> instead of a file name and return will be to the Editor. After the file has been written to disk, the Editor will prompt with the following:

Figure 3.19

```
-----
>Quit
Writing.....
Your file is 1978 bytes long.
Do you want to E(xit) from or R(eturn to the Editor)?
-----
```

Typing "E" exits from the Editor and returns to the Command level while typing "R" returns the cursor to the exact position in the file as when "Q" was typed.

```
*****
* REFERENCE SECTION * * Section 1.3.4 *
*****
```

```
<down-arrow> moves <repeat-factor> lines down
<up-arrow>    "      "      lines up
<right-arrow> "      "      spaces right
<left-arrow>  "      "      spaces left
<space>      "      "      spaces in direction
<back-space> "      "      spaces left
<tab>        moves <repeat-factor> tab positions in direction
<return>     moves to the beginning of line <repeat-factor> lines in directio
```

```
"<" " ," " "-" change direction to backward
">" " ." " "+" change direction to forward
"="          moves to the beginning of what was just found/replaced/inserted/
              exchanged
```

adjust: Adjusts the indentation of the line that the cursor is on. Use the arrow keys to move. Moving up (down) adjust line above (below) by same amount of adjustment on the line you were on. Repeat-factors are valid.

copy: Copies what was last inserted/deleted/zapped into the file at the position of the cursor.

delete: Treats the starting position of the cursor as the anchor. Use any moving commands to move the cursor. <ctr> deletes everything between the cursor and the anchor.

- **Find:** Operates in L)iteral or T)oken mode. Finds the <targ> string. Repeat-factors are valid, direction is applied. "S" = use same string as before.
 - **I)nsert:** Inserts text. Can use <backspace> and to reject part of your insertion.
 - **K)ump:** Jumps to the beginning, end or previously set marker.
 - **M)argin:** Adjusts anything between two blank lines to the margins which have been set. Command characters protect text from being margined. Invalidates the copy buffer.
 - **P)age:** Moves the cursor one page in direction. Repeat-factors are valid, direction is applied.
 - **Q)uit:** Leaves the editor. You may U)date, E)xit, W)rite, or R)eturn.
 - **R)eplace:** Operates in L)iteral or T)oken mode. Replaces the <targ> string with the <subs> string. V)erify option asks you to verify before it replaces. "S" option uses the Same string as before. Repeat-factors replace the target several times. Direction is valid.
 - **S)et:** Sets M)arkers by assigning a string name to them. Sets E)nvironment for A)uto-indent, F)illing, margins, T)oken, and C)ommand characters.
 - **V)erify:** Redisplays the screen with the cursor centered.
 - **X)change:** Exchanges the current text for the text typed while in this mode. Each line must be done separately. <back-space> causes the original character to re-appear.
 - **Z)ap:** Treats the starting position of the last thing found/replaced/inserted as an anchor and deletes everything between the anchor and the current cursor position.
- <repeat-factor> is any number typed before a command. Typing a / is the infinite number.

* L2 EDITOR * * Section 1.3.5 *

Version I.5 September 1978

The L2 Editor is being released on an experimental basis. Not all options are yet fully implemented so this section may not be complete. The main advantage of this version is that it is able to handle files larger than can fit into the main memory buffer at one time; the upper limit being determined by the space available on disk. It also automatically makes a backup copy of the file being edited. In many respects this Editor works exactly as this release and displays the same prompt lines. Where the versions are the same, the user is directed to read the main Editor section.

Entering the Workfile and Getting a Program

If, on typing E, there is not enough room on the disk:

ERROR: Not enough room for backup!

will be displayed. This disk must then be K(runched in order to provide room if that is possible, a file removed or another disk must be used.

The same prompt line is displayed; see section 1.3.2.

1) With a name. If a file is chosen, a backup copy will be made before the file is available for editing.

Figure 5.1

```
Copying to filename.back.  
>Edit  
Reading....
```

After this series of prompt lines, the first part of the text will appear on the screen.

2) With a return. A new file is created in the same manner as in section 1.3.2.

The paragraphs on moving the cursor, Insert and Delete in section 1.3.2. should be read and are applicable here.

Leaving the Editor and Updating the workfile

When all changes and additions have been made, the Editor is exited by typing "Q" and the following prompt is displayed.

Figure 5.2

```
>Quit:  
  U(pdate the workfile and leave  
  E(xit (but workfile not updated)  
  R(eturn to the Editor without doing anything.
```

Notice that the Write option is no longer available. One of these three options must be chosen. See also Miscellaneous commands in section 1.3.3.

U(update:

This works in the same manner, however additional information is supplied indicating the name of file updated and the length.

When a new file is created, the following appears:

Figure 5.3

Writing.*
The workfile, *SYSTEM.WRK.TEXT, is n blocks long.

When an existing file has been used, this example shows the extra information now given:

Figure 5.4

Writing.*
The workfile, *X:F1.TEXT, is 44 blocks long.
The backup file is X:F1.BACK.

The newly edited file is referred to as .TEXT, while the .BACK file contains the original file with no modifications.

E(exit:

This causes the Editor to return to the command level without making any changes in the workfile. No .BACK file is made and the existing .BACK is removed. For example, if F1.TEXT is the file being used, then a copy F1.BACK will be made on entering the editor and on leaving by using the E option, F1.BACK will be removed and only F1.TEXT will remain. However, since F1.TEXT is a copy of the original, it will be in different place in the directory.

R(return:

This is the same. See section 1.3.3.

MOVING COMMANDS

JUMP

Jump mode displays the same prompt line as before. In this case "B" and "E" refer to the beginning(end) of the buffer not the beginning(end) of the file.

Typing "M" causes the Editor to display:

Jump to what marker?

It is now possible to use 20 markers and these will be set in the same way as in section 1.3.3. To jump to the desired marker, type in the name. If the marker is present, the Editor will jump to that position, otherwise, the Editor will jump to the last position of the cursor in the file. If Find needs to search a section of the file, other than the buffer, Leaping..... will be displayed.
BANISH

This is a new command and is reached by typing "B" at the Edit level. This is the prompt that will appear:

>Banish: To the L(eft or R(ight <esc>

Prior to doing a large insertion or copy, in order to provide more room in the buffer and avoid buffer overflow, it is possible to move characters from the buffer into the stack. There is a left and a right stack; left being ahead of the cursor and right, behind the cursor. The user can make the choice according to the current situation. In general, the screen is the boundary for the operation.
NEXT

In order to move beyond the bounds of the buffer, type "N". The following prompt will then be displayed:

Next: F(orwards, B(ackwards in the file; S(tart, E(nd of the file. <esc>

Choose one of the five options available. When using "F" or "B", an implicit banish occurs using the cursor as the point of reference. For example, when "F" is typed, everything above the top of the screen is banished to the left stack. More characters are added to the bottom of the screen to extend the buffer in the forward direction. When "B" is used the characters below the cursor are banished to the right stack and part of the screen will become blank. More characters are added above the 'window' of the screen.

Figure 5.5 SYMBOLIC FILE

left stack			right stack	
Backwards		BUFFER	Forward	
Start			End	

PAGE

See section 1.3.3.

EQUALS

See section 1.3.3.

TEXT CHANGING COMMANDS

INSERT

See section 1.3.3.

DELETE

See section 1.3.3.

ZAP

See section 1.3.3.

COPY

See section 1.3.3.

EXCHANGE

See section 1.3.3.

FIND

Read section 1.3.3. The Editor will display: Finding.....
and if the pattern is not in the buffer:

End of buffer encountered. Get more from disk? (Y/N)

On typing "Y", the Editor will move another section of the file
into the buffer to continue searching. Find is still directional.

REPLACE

See section 1.3.3.

FORMATTING COMMANDS

ADJUST

See section 1.3.3.

MARGIN

See section 1.3.3.

MISCELLANEOUS COMMANDS

SET

See section 1.3.3. The same prompt line is displayed.

M(marker:

Read section 1.3.3. The names of the markers can be seen by typing "SE" for Set Environment while at the Edit level. To set the marker, type "SM". In the event that 20 markers have already been set, this will be indicated by:

Marker overflow. Which one to replace? (Type in the letter or <sp>

E(nvironment:

To set the environment, type "SE". The following is an example of the prompt displayed:

Figure 5.5

```
>Environment: options <etr> or <sp> to leave
A(uto Indent False
F(illing True
L(eftright margin 4
R(ight margin 70
P(ara margin 1
C(ommand ch ^
S(et tabstops
T(oken def True
```

```
11582 bytes used. 2754 available.
```

```
There are 0 pages in the left stack, and 10 pages in the right stack.
You have 86 pages of room, and at most 13 pages worth in the buffer.
```

Markers:

```
<P1 P2 >P3
```

```
Created August 15, 1978: Last updated August 15, 1978 (Revision 1).
```

By typing the appropriate letter, any or all of the options can be changed. See section 1.3.3. The arrow before the marker name indicates the relative position of the marker in the file to the buffer. No arrow indicates that the marker is in the current buffer.

It is now possible to vary the tabstops. Type "S" while in the environment and the following prompt will appear:

Set tabs: <right, left vectors> C(ol# N(o R(ight L(eft D(ecimal stop <ctr>

At present, these are not yet fully implemented so that the effect of using any of them is to have a variable tabstop instead of being set at eight characters apart.

VERIFY

See section 1.3.3.

* YET ANOTHER LINE ORIENTED EDITOR - YALOE * * Section 1.4 *

Version I.5 September 1978

This text editor is intended for use on systems that do not have powerful screen terminals. It is designed to be very similar to the text-editor which accompanies DEC's RT-11 system.

The editor assumes, but is not dependent on, the existence of the workfile text. Upon reading it YALOE will proclaim 'workfile STUFF read in'. If it does not find such a file, it will proclaim 'No work file read in'. This means that you entered YALOE with an empty workfile. From this point you may create a file in YALOE; and when you exit by typing 'QU', your workfile will no longer be empty.

The editor operates in one of two modes: Command Mode or Text Mode. In command mode all keyboard input is interpreted as commands instructing the editor to perform some operation. When you first enter the editor you will be in the Command Mode. The Text Mode is entered whenever the user types a command which must be followed by a text string. After the command F(ind, G(et, I(nsert, M(acro define, R(ead file, W(rite to file, or eX(change has been typed, all succeeding characters are considered part of the text string until an <esc> is typed. Note: when typed <esc> echoes a '\$'. The <esc> terminates the text string and causes the editor to re-enter the Command Mode, at which point all characters are again considered commands.

NOTE: Follow command strings in YALOE with <esc><esc> to execute them. (This is unlike the rest of the systems 'immediate' commands.)

1.4.1 SPECIAL KEY COMMANDS

Various characters have special meanings, as described below. Some of these apply only in YALOE. Many have similar effects in the rest of the system; for these the ASCII code to which the system responds as indicated can be changed using the program SETUP, described in Section 4.3. (<esc> is the most particular anomaly to YALOE.)

<esc>	Echoes a '\$'. A single <esc> terminates a text string. A double <esc> executes the command string.
RUBOUT <linedel>	Deletes current line. On hard-copy terminals echoes 'ZAP' and a carriage return. On others, it clears the current line on the screen. In both cases the contents of that line are discarded by the editor.

CTRL H
<chardel>

Deletes character from the current line. On hard-copy terminals it echoes a percent sign followed by the character deleted. Each succeeding CTRL H the user deletes and echoes another character. An enclosing percent sign is printed when a key other than CTRL H is typed. This erasure is done right to left up to the beginning of the command string. CTRL H may be used in both Command and Text mode.

CTRL X

Causes the editor to ignore the entire command string currently being entered. The editor responds with a <cr> and an asterisk to indicate that the user may enter another command. For example:

```
*IDALE AND  
KEITH<CTRL X>  
*
```

A <chardel> would cause deletion of only KEITH; CTRL X would erase the entire command.

CTRL O

Will switch you to the optional character set (i. e. bit 7 turned on). This works only on the TERA 8510A. The CTRL O is used as a toggle between the character sets. NOTE: You may find while in the editor that weird characters are showing up on the terminal instead of normal ones. It could be because you accidentally typed CTRL O. To get back just type CTRL O again.

CTRL F
<flush>

All output to the terminal is discarded by the system until the next CTRL F is typed.

CTRL S
<stop>

All output to the terminal is held until another CTRL S is typed.

All other control characters are ignored and discarded by YALOE.

1.4.2 COMMAND ARGUMENTS

A command argument precedes a command letter and is used either to indicate the number of times the command should be performed or to specify the particular portion of text to be affected by the command. With some commands this specification is implicit and no argument is needed; other commands, however, require an argument.

Command arguments are as follows:

- n n stands for any integer. It may be preceded by a + or -. If no sign precedes n, it is assumed to be a positive number. Whenever an argument is acceptable in a command, its absence implies an argument of 1 (or -1 if only the - is present).
- m m is a number 0..9.
- O 'O' refers to the beginning of the current line.
- / '/' means 32700. '-/' means -32700. It is used for a large repeat factor.
- = '=' is used only with the J, D and C commands and represents -n, where n is equal to the length of the last text argument used, for example *GTHIS\$=D\$\$ finds and removes THIS.

1.4.3 COMMAND STRINGS

All EDIT command strings are terminated by two successive <esc>s. Spaces, carriage returns and tabs (CTRL I) within a command string are ignored unless they appear in a text string.

Several commands can be strung together and executed in sequence. For example:

```
*B  GTHE INSERTED$  -3CING$      5K      OSTRING$$
```

As a rule, commands are separated from one another by a single <esc>. This separating <esc> is not needed, however, if the command requires no text. Commands are terminated by a single <esc>; a second <esc> signals the end of a command string, which will then be executed. When the execution of the command string is complete, the editor prompts for the next command with '*'.
 *

If at any point in executing the command, an error is encountered, the command will be terminated, leaving the command executed only up to that point.

1.4.4 THE TEXT BUFFER

The current version of your text is stored in the Text Buffer. This buffer's area is dynamically allocated; its size and the room left for expansion may be ascertained by using the ? command.

The editor can only work on files that fit entirely within the Text Buffer. The Screen Oriented Editor in the next major release will not have this limitation.

1.4.4 THE CURSOR

The "cursor" is the position in your text where the next command will be executed. In other words it is the current "pointer" into the Text Buffer. Most edit commands function with respect to the cursor:

A,B,F,G,J: Moves it.
D,K: Remove text from where it is.
U,I,R: Add text to where it is.
C,X: Remove and then add text at it.
L,V: Print the text on the terminal from it.

1.4.5 INPUT/OUTPUT COMMANDS

L(list, V(erify, W(rite, R(ead, Q(uit, E(rase, and O

The L(list command prints the specified number of lines on the console terminal without moving the cursor.

-2L\$	Prints all characters starting at the second preceding line and ending at the cursor.
4L\$	Prints all characters beginning at the cursor and terminating at the 4th <cr>.
OL\$	Prints from the beginning of the current line up to the cursor.

The V(erify command prints the current text line on the terminal. The position of the cursor within the line has no effect and the cursor is not moved. No arguments are used. The V(erify command is equivalent to a OLL (list) command.

The W(rite command is of the form

WCfile title>

File title is any legal file title as described in Section 1.2 less the file type. The editor will automatically append a '.TEXT' suffix to the file title given unless the file title ends with '.', ']', or '.TEXT'. If the filename ends in a '.', the dot will be stripped from the filename.

The W(rite command will write the entire Text Buffer to a file with the given file title. It will not move the cursor nor alter the contents of the Text Buffer.

If there is no room for the Text Buffer on the volume specified in the file title given, the message:

OUTPUT ERROR. HELP!

will be printed. It is still possible to write the Text Buffer out by writing it to another volume.

The R(ead command is of the form

R<file title>

The editor will attempt to read the file title as given. In the event no file with that title is present, a '.TEXT' is appended and a new search is made.

The R(ead command inserts the specified file into the Text Buffer at the cursor. The cursor remains in the Text Buffer before the text inserted. If the file read in does not fit into core buffer, the entire Text Buffer will be undefined in content, i.e. this is an unrecoverable error.

The Q(uit command has several forms

GU	Quit and update by writing out a new SYSTEM.WRK.TEXT
GE	Quit and escape session; do not alter SYSTEM.WRK.TEXT
GR	Don't quit; return to the editor
G	A prompt will be sent to the terminal giving all the above choices; enter option mnemonic (U, E, or R) only.

Executing the GU command is a special case of the write command, and the attempt to write out SYSTEM.WRK.TEXT may fail. In this case use the W command to write out your file and then GE to exit the editor.

The QR command is used on the occasions when a G is accidentally typed, and you wish to return to the editor rather than leave it.

The E(rase command (intended for CRT terminals) erases the screen.

The O command (also intended for CRT terminals) can be used to have the context around the cursor displayed on the screen each time the cursor is moved. The argument of the O command determines the size (# of lines) in that context. This option is initially disabled when the editor is entered and can be enabled by issuing an O command. A second O command disables the option; succeeding 'O's successively enable, disable etc. The cursor is denoted as a split in the line..

1.4.6 CURSOR RELOCATION COMMANDS

J(ump, A(dvance, B(eginning, Q(ue, F(ind

When using character and line oriented commands, a positive (n or +n) argument specifies the number of characters or lines in a forward direction, and a negative argument the number of characters or lines in a backward direction. The editor recognizes a line of text as a unit when it detects a <CR> in the text.

Carriage return characters are treated the same as any other character. For example assume the cursor is positioned as indicated in the following text (^ represents the current position of the cursor and does not appear in actual use. It is present here only for clarification):

```
THERE WAS A CROOKED MAN^<CR>
AND HUMPTY DUMPTY FELL ON HIM<CR>
```

The J(ump command moves the cursor over the specified number of characters in the Text Buffer. The edit command -4J moves the cursor back 4 characters.

```
THERE WAS A CROOKED^ MAN<CR>
AND HUMPTY DUMPTY FELL ON HIM<CR>
```

The command 10J moves the cursor forward 10 characters and places it between the 'H' and the 'U'.

```
THERE WAS A CROOKED MAN<CR>
AND H^UMPTY DUMPTY FELL ON HIM<CR>
```

The A(dvance command moves the cursor a specified number of lines. The cursor is left positioned at the beginning of the line.

Hence the command 0A moves the cursor to the beginning of the current line.

```
THERE WAS A CROOKED MAN<CR>
^AND HUMPTY DUMPTY FELL ON HIM<CR>
```

The command -1A (or -A) moves the cursor back one line.

```
^THERE WAS A CROOKED MAN<CR>
AND HUMPTY DUMPTY FELL ON HIM<CR>
```

The B(eginning command moves the cursor to the beginning of the Text Buffer.

Search commands are used to locate specific characters or strings of characters within the Text Buffer.

The G(et and F(ind commands are synonymous. Starting at the position of the cursor, the current Text Buffer is searched for the nth occurrence of a specified text string. A successful search leaves the cursor immediately after the nth occurrence of the text string if n is positive and immediately before the text string if n is negative. An unsuccessful search generates an error message and leaves the cursor at the end of the Text Buffer for n positive and at the beginning for n negative.

***BGSTRING=J#** This command string will look for the string STRING starting at the beginning of the Text Buffer; and if found it will leave the cursor immediately before it.

1.4.7 TEXT MODIFICATION COMMANDS

I(nsert, D(elete, K(ill, C(hange, eX(change

The I(nsert command causes the editor to enter the TEXT mode. Characters are inserted immediately following the cursor until an <esc> is typed. The cursor is positioned immediately after the last character of the insert. Occasionally with large insertions the temporary insert buffer becomes full. Before this happens a message will be printed on the console terminal, 'Please finish'. In response type two successive <esc>s. To continue, type I to return to the Text mode.

NOTE: Forgetting to type the I command will cause the text entered to be executed as commands.

The D(elete command removes a specified number of characters from the Text Buffer, starting at the position of the cursor. Upon completion of the command, the cursor's position is at the first character following the deleted text.

-2D* Deletes the two characters immediately preceding the cursor.

B\$HOSE \$=D* Deletes the first string 'HOSE' in the Text Buffer, since =D used in combination with a search command will delete the indicated text string.

The K(ill command deletes n lines from the Text Buffer, starting at the position of the cursor. Upon completion of the command, the cursor's position is the beginning of the line following the deleted text.

2K* Deletes characters starting at the current cursor position and ending at (and including) the second <CR>.

/K* Deletes all lines in the Text Buffer after the cursor.

The C(hange command replaces n characters, starting at the cursor, with the specified text string. Upon completion of the command, the cursor immediately follows the changed text.

OCAPPLES* Replaces the characters from the beginning of the line up to the cursor with 'APPLES', (equivalent to using OX).

BGOHOSE\$=CLIZARD* Searches for the first occurrence of 'HOSE' in the Text Buffer and replace it with 'LIZARD'.

The eX(change command exchanges n lines, starting at the cursor, with the indicated text string. The cursor remains at the end of the changed text.

-5XTEXT* Exchanges all characters beginning with the first character on the 5th line back and ending at the cursor with the string 'TEXT'.

OXTEXT* Exchanges the current line from the beginning to the cursor with the string 'TEXT', (equivalent to using OC).

/XTEXT* Exchanges the lines from the cursor to the end of the Text Buffer with the text 'TEXT', (equivalent to using /C).

1.4.8 OTHER COMMANDS

S(ave, U(nsave, M(acro, N (macro execution) and '?'

The S(ave command copies the specified number of lines into the Save Buffer starting at the cursor. The cursor position does not change, and the contents of the Text Buffer are not altered. Each time a S(ave is executed, the previous contents of the Save Buffer, if any, are destroyed. If executing the S(ave command would have overflowed the Text Buffer, the editor will generate a message to this effect and not perform the save.

The U(nsave command inserts the entire contents of the Save Buffer into the Text Buffer at the cursor. The cursor remains before the inserted text. If there is not enough room in Text Buffer for the Save Buffer, the editor will generate a message to this effect and not execute the unsave.

The Save Buffer may be removed with the command OU.

The M(acro command is used to define macros. A maximum of ten macros, identified by the integer (0..9) preceding the 'M', are allowed. The default number is 1. The M(acro command is of the form:

mM%command string%

This says to store the command string into Macro Buffer number m, where m is the optional integer 0..9. The delimiter, '%' in this example, is always the first character following the M command and may be any character which does not appear in the macro command string itself. The second occurrence of the delimiter terminates the macro.

All characters except the delimiter are legal Macro command string characters, including single <esc>s. All commands are legal in a macro command string. Example of a macro definition:

*5M%GBEGIN%=CEND BEGIN%V%\$%

This defines macro number 5. When macro number 5 is executed, it will look for the string 'BEGIN', change it to 'END BEGIN', and then display the change.

If an error occurs when defining a macro, the message

'Error in macro definition'

will be printed, and the macro will have to be redefined.

The execute macro command, N, executes a specified macro command string. The form of the command is:

nNm\$

Here n is simply any command argument as previously defined; m is the macro number (an integer 0..9) to be executed. If m is omitted, 1 is assumed. Because the digit m is technically a command text string, the N command must be terminated by an <esc>.

Attempts to execute undefined macros cause the error message 'Unhappy macnum'. Errors encountered during macro execution cause the message 'Error in macro'. Errors encountered in macro command syntax cause the message 'Error in macro definition'.

The ? command prints a list of all the commands and the sizes of the Text Buffer, Save Buffer, and available memory left for expansion.

1.4.9 SUMMARY OF ALL COMMANDS

n - an argument m - macro number

nA: Advance the cursor to the beginning of the n th line from the current position.

B: Go to the Beginning of the file.

nC: Change by deleting n characters and inserting the following text. Terminate text with <esc>.

nD: Delete n characters.

E: Erase the screen.

nF: Find the n th occurrence from the current cursor position of the following string. Terminate target string with <esc>.

nG: Get - ditto -

H: - invalid -

I: Insert the following text. Terminate text with <esc>.

nJ: Jump cursor n characters.

nK: Kill n lines of text. If current cursor position is not at the start of the line, the first part of the line remains.

nL: List n lines of text.

mM: Define macro number m.

nNm: Perform macro number m, n times.

nO: On, off toggle. If on, n lines of text will be displayed above and below the cursor each time the cursor is moved. If the cursor is in the middle of a line then the line will be split into two parts. The default is whatever fills the screen. Type O to turn off.

P: - invalid -

Q: Quit this session, followed by:

U:(pdate Write out a new SYSTEM.WRK.TEXT

E:(scape Escape from session

R:(return Return to editor

R: Read this file into buffer (insert at cursor);
'R' must be followed by <file name> <esc>;
WARNING: If the file will not fit into the buffer, the content of the buffer becomes undefined!

nS: Put the next n lines of text from the cursor position into the Save Buffer.

T: - invalid -

U: Insert (Unsave) the contents of the Save Buffer into the text at the cursor; does not destroy the Save Buffer.

V: Verify: display the current line

W: Write this file (from start of buffer);
'W' must be followed by <filename> <esc>.

nX: Delete n lines of text, and insert the following text; terminate with <esc>.

Y: - invalid -

Z: - invalid -

- Notes -

DUE TO THE LARGE NUMBER OF BUGS IN THE DEBUGGER, WE HAVE OMITTED THE
DEBUGGER, AND ITS CORRESPONDING DOCUMENTATION FROM THE SYSTEM RELEASE.

THE DEBUGGER WILL BE AVAILABLE AT SOME TIME IN THE FUTURE, AND YOU WILL
BE NOTIFIED OF THIS FACT. PLEASE DO NOT ASK US ABOUT THE DEBUGGER, AS
THE REPLY YOU GET WILL BE THE SAME AS THE MESSAGE ON THIS PAGE.

Thank you for your patience in this matter. ed.

Pages 74 through 80 have been omitted.

- notes -

Pages 72..80

* PASCAL COMPILER * * Section 1.6 *

Version 1.5 September 1978

The UCSD Pascal compiler, a one-pass recursive descent based on the P2 portable compiler from Zurich, is invoked by using the C(ompile or R(un) command of the outermost level of the UCSD Pascal system. If a workfile exists, it compiles that. Otherwise, it prompts the user for a source file name. It generates codefiles to run directly on the Pascal interpretive machine.

Unless the HAS SLOW TERMINAL boolean inside the system communication area (see section 4.3) is true, the compiler, during the course of compilation, will display on the CONSOLE device output detailing the progress of the compilation. This output can be suppressed with the G+ compiler option (see section on compiler options below). Below is an example of the output which appears on the CONSOLE device:

```
PASCAL compiler [1.5 unit compiler]
< 0>.....
P1 [7050]
< 19>.....
P2 [3040]
< 61>.....
TEST [3003]
< 119>.....
```

The identifiers appearing on the screen are the identifiers of the program and its procedures. The identifier for a procedure is displayed at the moment when compilation of the procedure body is started. The numbers within [] indicate the number of (16 bit) words available for symbol table storage at that point in the compilation. The numbers enclosed within < > are the current line numbers. Each dot on the screen represents 1 source line compiled.

If the compilation is successful, that is, no syntax errors detected, the compiler writes a codefile to the disk called *SYSTEM.WRK.CODE. This is the codefile which is executed if the user had typed the R(un) command. See Section 1.1 INTRODUCTION AND OVERVIEW for further details on the system commands.

Should the compiler detect a syntax error, the text surrounding the error and an error number together with the marker '<<<<' will point to the symbol in the source where the error was detected. In the event that both the G and L options are set, the compilation will continue, with the syntax error going to the listing file, and the console remaining undisturbed. The compiler will then give the user the option of typing a space, an <esc> or 'E'. Typing a space instructs the compiler to attempt to continue the compilation, while escape causes the termination of the compilation, and "E" results in a call to the editor, which automatically places the cursor at the symbol where the error was detected.

Most of the syntax errors detected by the UCSD Pascal compiler are the standard ones listed in Jensen & Wirth. A complete list of all UCSD syntax errors can be found in Table 5. All error numbers will be accompanied by a textual message upon entry to the editor if the file *SYSTEM.SYNTAX is available.

1.6.1 COMPILE TIME OPTIONS

Compile time options in the UCSD Pascal compiler are set according to a convention described on pages 100-102 of Jensen and Wirth, where compile time options are set by means of special "dollar sign" comments inside the Pascal program text. The syntax used in UCSD's compiler control comments is essentially as described in Jensen and Wirth. The actual options and the letters associated with those options bear only little resemblance to the options listed on pages 101 and 102 of Jensen and Wirth. If a '+' or '-' is not specified after an option letter, '+' is assumed. The following sections describe the various options currently available to the user of the UCSD Pascal compiler.

D:

This option causes the compiler to issue breakpoint instructions into the codefile during the course of the compilation in order that the interactive Debugger can be used more effectively. See Section 3.2 "DEBUGGER" for details

Default value: D-

D-: causes the compiler to omit breakpoint instructions during the course of the compilation.

D+: causes the compiler to emit breakpoint instructions.

G:

Affects the boolean variable GOTOOK in the compiler. This boolean is used by the compiler to determine whether it should allow the use of the Pascal GOTO statement within the program.

Default value: G-

G+: allows the use of the GOTO statement.

G-: causes the compiler to generate a syntax error upon encountering a GOTO statement.

C: The (*\$C comment*) places the comment, (80 character maximum*) in the code file generated. This option is used at UCSD to place copyright information in the codefile.

The G-option has been used at U.C.S.D to restrict novice programmers from excessive uses of the GOTO statement in situations where more structured constructs such as FOR, WHILE, or REPEAT statements would be more appropriate.

I:

When an 'I' is followed immediately by a '+' or '-', the control comment will affect the boolean variable IOCHECK within the compiler. An alternative use of 'I' in a compiler control comment causes the compiler to include a different source file into the compilation at that point. See section INCLUDE-FILE MECHANISM for syntax.

IOCHECK OPTION

Default value: I+

I+: instructs the compiler to generate code after each statement which performs any I/O, in order to check to see if the I/O operation was accomplished successfully. In the case of an unsuccessful I/O operation the program will be terminated with a run time error.

I-: instructs the compiler not to generate any I/O checking code. In the case of an unsuccessful I/O operation the program is not terminated with a run time error.

The I-option is useful for system level programs which do many I/O operations and also checks the IORESULT function after each I/O operation. The system program can then detect and report the I/O errors, without being terminated abnormally with a run time error. However this option is set at the expense of the increased possibility that I/O errors, (and possibly severe program bugs), will go undetected.

INCLUDE FILE MECHANISM

The syntax for instructing the compiler to include another source file into the compilation is as follows:

```
(*#IFILENAME*)
```

The characters between 'I' and '*' are taken as the filename of the source file to be included. The comment must be closed at the end of the filename, therefore no other options, such as Q+, or L+, etc. can follow the filename. Note that if a file name starts with '+' or '-' as the first character of the filename, a blank must be inserted between '(*I' and 'FILENAME'. For example, the comment:

```
(*ITURTLE.TEXT*)
```

would cause the file TURTLE.TEXT to be compiled into the program at that point in the compilation.

```
(*I +FARKLE.STUFF*)
```

would cause the source file +FARKLE.STUFF to be included into the compilation.

If the initial attempt to open the include file fails, the compiler concatenates a ".TEXT" to the file-name and tries again. If this second attempt fails, or some I/O error occurs at some point while reading the include file, the compiler responds with a fatal syntax error.

The compiler accepts include files which contain CONST, TYPE, VAR, PROCEDURE, and FUNCTION declarations even though the original program has previously completed its declarations. To do so, the include compiler control comment must appear between the original program's last VAR declaration and the first of the original program's PROCEDURE or FUNCTION declarations. Note that an include file may be inserted into the original program at any point desired, provided the rules governing the normal ordering of Pascal declarations will not be violated. Only when these rules are violated does the above procedure apply.

The compiler cannot keep track of nested include comments, i.e. an include file may not have an include file control comment. This results in a fatal syntax error.

The include file option was added to the compiler at U.C.S.D in order to make it easier to compile large programs without having to have the entire source in one very large file which in many cases would be too large to edit in the existing editors' buffer.

L:

Controls whether the compiler will generate a program listing of the source text to a given file. The default value of this option is L-, which implies that no compiled listing will be made. If the character following "L" is "+", then the compiled listing will be sent to a diskfile with the title '*SYSTEM.LST.TEXT'. The user may override this default destination for the compiled listing by specifying a filename following "L". For example the following control comment will cause the compiled listing to be sent to a diskfile called "DEMO1.TEXT":

(*\$L DEMO1.TEXT*)

To specify a file-name inside a control comment, see the section describing the include file mechanism.

Note that listing files which are sent to the disk may be edited as any other text file provided the filename which is specified contains the suffix ".TEXT". Without the ".TEXT" suffix the file will be treated by the system as a datafile rather than as a text file.

The compiler outputs next to each source line the line number, segment procedure number, procedure number, and the number of bytes or words (bytes for code, words for data) required by that procedure's declarations or code to that point. The compiler also indicates whether the line lies within the actual code to be executed or is a part of the declarations for that procedure by outputting a "D" for declaration and an integer 0..9 to designate the lexical level of statement nesting within the code part. If the D+ option is set then the listing file will include an asterisk on each line where it is appropriate for a user to specify a breakpoint while in the interactive Debugger. This information can be very valuable for debugging a large program since a run time error message will indicate the procedure number, and the offset where the error occurred.

Q:

The Q compiler option is the "quiet compile" option which can be used to suppress the output to the CONSOLE device of procedure names and line numbers detailing the progress of the compilation.

Default value: is set equal to current value of the SLOWTERM attribute of the system communication record SYSCOM. (actually SYSCOM^.MISCINFO.SLOWTERM)

Q+: causes the compiler to suppress output to CONSOLE device.

Q-: causes the compiler to send procedure name and line number output to the CONSOLE device.

R:

This option affects the value of the boolean variable RANGECHECK in the compiler. If RANGECHECK is true, the compiler will output additional code to perform checking on array subscripts and assignments to variables of subrange types.

P:

This option causes the listing to continue from top-of-form. i.e. the compiler does:

PAGE(LISTFILE)

Default value: R+

R+: turns range checking on.

R-: turns range checking off.

Note that programs compiled with the R-option set will run slightly faster; however if an invalid index occurs or a invalid assignment is made, the program will not be terminated with a run time error. Until a program has been completely tested and known to be correct, it is usually best to compile with the R+ option left on.

S:

This option determines whether the compiler operates in "swapping" mode. There are two main parts of the compiler: one processes declarations; the other handles statements. In swapping mode, only one of these parts is in main memory at a time. This makes about 2500 additional words available for symbol table storage at the cost of slower compilation speed due to the overhead of swapping the compiler segment in from disk. On fullsize, single density floppy disks this amounts to a factor of two reduction in compile speed. This option must occur prior the the compiler encountering any Pascal syntax.

Default value: S-

S+: puts compiler in swapping mode.

S-: puts compiler in non-swapping mode.

U:

USER PROGRAM OPTION:

This option sets the boolean variable SYSCOMP in the compiler which is used by the compiler to determine whether this compilation is a user program compilation, or a compilation of a system program.

Default value: U+

U+: informs the compiler that this compilation is to take place on the user program lex level.

U-: informs the compiler to compile the program at the system lex level. This setting of the U compile time option also causes the following options to be set: R-, G+, I-.

NOTE: This option will generate programs that will not behave as expected. Not recommended for non-systems work without knowing its method of operation.

USE LIBRARY OPTION:

In this version of the 'U' option, the U is followed by a file name. The named file becomes the library file in which subsequent USEed UNITS are sought. The default file for the library is *SYSTEM.LIBRARY. (see section 3.3.2 for more details on UNITS)

Following is an example of a valid USES clause using the 'U' option:

```
USES UNIT1,UNIT2, {Found in *SYSTEM.LIBRARY}
  {SU A.CODE}
  UNIT3,
  {SU B.LIBRARY}
  UNIT4,UNIT5;
```

* UCSD BASIC COMPILER * * Section 1.7 *

Version I.5 September 1978

This section has been designed for programmers who are already familiar with Basic. Its intent is to describe to those experienced users the details of UCSD Basic in a manner sufficiently detailed so as to enable the writing or modification of programs in a manner compatible with the UCSD Basic Compiler.

The first section contains a brief description of the features included in UCSD Basic; the second, the descriptions of the features unique to UCSD Basic, and the third a list of those features which we intend UCSD Basic to allow, but which are not yet implemented.

The UCSD Basic Compiler has been written in the Pascal language. Some of the intrinsics of the Pascal language, which are not found in standard Basic, are found within the UCSD version of Basic. Many of these are noted in the first section, all of them are noted or recapped in the second.

The UCSD BASIC Compiler is invoked just like the Pascal compiler, provided the compiler code is named *SYSTEM.COMPILER. Originally it will be named BASIC.COMPILER. If you want a disk to be BASIC oriented, you must change the name of, or remove, the Pascal compiler, and change the name of BASIC.COMPILER to *SYSTEM.COMPILER. That disk, and any copies of it, will now compile BASIC programs as a result of the C(ompile or R(un) command.

The Basic compiler has only real and string variables. When applying a real to indexing or other integer purposes the rounded value of the number is used. In the functions below x and y can be real variables or expressions which evaluate to real values. Similarly s1 and s2 can be string variables or expressions which evaluate to a string.

Real variables: letter(digit).
String variables: letter(digit)\$. The digit is optional.

ATN(x) Returns the angle in radians whose tangent is x.

EXP(x) Returns the base of the natural logarithms raised to the power x.

INT(x) Returns the value of x rounded to the nearest integer.

LOG(x) Returns the log (base 10) of x.

LN(x) Returns the natural log of x.

MOD(x,y) Returns x modulo y.

SIN(x) Returns the sine of the angle x. Where x is in radians.

COS(x) Returns the cosine of an angle x. Where x is in radians.

CAT*(s1,s2,...) Returns a string which is equal to the concatenation of all the strings in the parameter list.

COP*(s1,x,y) Returns a copy of the portion of the string s1, y consecutive characters, starting with the character at position x.

DEL*(s1,x,y) Returns the contents of the string s1 with y consecutive characters deleted. The deletion starts with the character at position x.

INS*(s1,s2,x) Returns the contents of string s2 with string s1 inserted immediately before the character which is at position x.

LEN(s1) Returns the length of the string s1.

POS(s1,s2) Returns an integer which is equal to the position of the first character in the first occurrence of the string s1 in the string s2.

OTHER FUNCTIONS

ORD(s) Returns the ASCII value of the first character of the string s.

STR*(x) Returns the string containing the character associated with the ASCII value x.

GET\$ Reads a single character from the keyboard without prompt or echoing, and returns it as a string. GET\$ requires no arguments.

OLD(c,s)

NEW(c,s) c is a numeric constant without a fraction part, which becomes associated with the disk file whose name is in s. OLD expects that file to already exist, NEW creates a new one with the name s, removing any previous file of that name. These functions must occur before associated print or input statements. The numbers may not be reassigned and must be in the range 1..16. For best results, use only at the top of a program. In order that a file created by NEW be editable with either of the system editors, '.text' must be appended to the file title.

These functions return IORESULT as described in section 2.1.

Arithmetic statements and operations

- , +	subtract, add
/ , *	divide, multiply
^ , **	exponentiation

Relational operators

=	equals
<> , ><	not equals
>	greater than
<	less than
>= , =>	greater than or equal
<= , =<	less than or equal

INPUT list
OR
INPUT #c list

Inputs from the main system device, usually the keyboard. If the optional #c is present, INPUT inputs from the disk file number c. The input list may contain any combination of real variables and string variables. When a program expects input the prompt "?" is printed. Input of real numbers may be terminated with any non-numeric character. Input of strings must be terminated with a return.

PRINT list
OR
PRINT #c list

Writes to the main output device the list following the PRINT command. If the optional #c is present, PRINT outputs to the diskfile number c. The output list may contain any variable, subscripted array variable, any arithmetic or string expression, or any literal text. The list may be separated by commas or semi-colons. If the list ends in a semi-colon the carriage return is suppressed. Literals may be enclosed in either type of quotation marks. Double quotation marks prints a single quotation mark.

```
FOR var = exp1 TO exp2 STEP exp3
:
NEXT var
```

Each execution of the loop increments the loop counter "var" by the amount of expression 3. If the STEP is omitted it is assumed to be 1. Only increasing STEP values are allowed. Evaluation of limits and increments is done at the beginning of the loop. Note that RETURN's into or GOTO's into a FOR loop may cause the loop to be undefined.

```
IF exp1 (relation operator) exp2 THEN (line number)
    GOTO
```

Either the reserved word THEN or GOTO can be used in this statement. If the relation between the exp1 and exp2 is found to be true the branch occurs. A string is considered to be less than another string if it is lexicographically smaller.

```
ON exp GOTO(ln1,ln2..)
```

If the expression, when rounded, evaluates to 1 it goes to the first line number (ln1) if it evaluates to 2 it goes to ln2, etc. This is the only form of the computed GOTO which is available. If the expression is out of range an error occurs.

```
DEF FNname(list)=expression      or      DEF FNname(list)
:
FNEND
```

Single line and multi-line functions are allowable. The function name must be a legal variable name for the type of value returned. Functions may be defined recursively. The parameter list is called by value, that is, changes inside the function don't affect the value of the external parameters.

```
LET var=exp
    or
var=exp
```

This command assigns a new value to the variable. If the variable is a string, the expression must evaluate to a string, and if a real, evaluation must be to a real.

```
DIM var (n1,n2,...)
```

A single or multidimensional array may be declared with this command. The variable name determines the type of the array. The array indices are 0..n1,0..n2,... Both real and string multidimensional arrays can be used. If no dimensions are declared the dimensions are assumed to be 0..10, 0..10, 0..1, 0..1 ... The number of dimensions automatically declared depends on the number of dimensions which are used in the program, but must be consistent over all uses of any given array.

GOSUB linenumber

Executes a subroutine call. The calling address is placed on the subroutine stack. Subroutine calls may be recursive.

RETURN

Returns to the line after the last GOSUB which is still pending. It pops the top address off the stack and uses it as the return address. A return when no GOSUB's are pending is an error.

GOTO linenumber

Program execution jumps to the given line number.

REM text

This line is a remark.

OTHER FEATURES OF UCSD PASCAL

Arithmetic

For loops: Note that var=exp1 is done before exp2 or exp3 are evaluated.

Continuation of statements is allowed. Any line not beginning with a line number is assumed to be the continuation of the line above.

Functions: All parameters of functions are call by value. You are not allowed to use the parameters to return values from a function. Function calls are allowed to be recursive.

Strings: The string functions and procedures are those found in the UCSD Pascal language.

Arrays: Arrays of more than two dimensions are allowed.

Print: Tab stops are not allowed. All list elements are printed without spaces between them. The carriage return can be suppressed by ";" as the last symbol in the line.

Subroutines: Subroutines may be recursive.

Comments: In line comments may be inserted. The portion of any line following the @ symbol is ignored by the compiler.

PASCAL FUNCTIONS: The code of PASCAL FUNCTIONS may be added to the BASIC compiler as new standard BASIC functions. This is accomplished by a straight-forward addition to the BASIC compiler.

Certain features of the UCSD Basic compiler are still in the process of being implemented. The most important of these are listed below.

Data and Read: The standard initialization statements.

Matrix statement for standard matrix operations.

Integer variables.

More standard functions.

Create the BASIC program using one of the system text editors. Once you have ensured that the BASIC compiler has been named SYSTEM.COMPILER, you can use the commands C(ompile and R(un at the COMMAND level, just as if you were using Pascal on a disk which has the Pascal compiler as its SYSTEM.COMPILER. For a more detailed description of COMMAND see Section 1.1.

* THE LINKER * * Section 1.8 *

Version 1.5 September 1978

The UCSD LINKER allows the user to combine pre-compiled files, which may have been written either in PASCAL or in assembly language, into the system workfile. The user may wish to incorporate certain useful routines into programs without having to rewrite or even recompile these routines. For example, one might wish to use a fast assembly language routine for some "real-time" application. This routine could be assembled separately, stored in a library, and eventually accessed via the LINKER.

To link in routines (either procedures or functions), the calling program declares those routines to be EXTERNAL, much as PROCEDURES or FUNCTIONS may be declared FORWARD (see Section 3.3.1). This notifies the compiler that the routines may be called, but are not provided yet. The compiler will then inform the system that linking is required before execution.

The LINKER can also be used to link in UNITS. A UNIT is a group of related routines which will be used together to perform a common task. UCSD TURTLEGRAPHICS is an example of a UNIT containing procedures and functions with which a "turtle" can be moved on the screen. A UNIT can be used by typing the command USES <unitname> directly after the PROGRAM <identifier>. For more information on UNITS, see Section 3.3.2.

Any files which reference UNITS or EXTERNAL routines and have not yet been linked may be compiled and saved, but will need to be linked before they can be executed.

1.8.1 USING THE LINKER

If the program in the workfile contains EXTERNAL declarations, or uses UNITS, typing R(un will automatically invoke the LINKER after the compiler. The LINKER will search the file *SYSTEM.LIBRARY for the routines or UNITS specified, and will attempt to link them into the workfile. If the UNIT or EXTERNALLY declared routine is not present in *SYSTEM.LIBRARY, the LINKER will respond with an appropriate message:

```
Unit,  
Proc,  
Func,  
Global,  
or Public <identifier> undefined
```

The LINKER may also be invoked explicitly, and, in fact, must be invoked explicitly in cases where

1) the file into which UNITS or EXTERNAL routines are to be linked is not the workfile, or

(2) the external routines to be linked reside in library files other than *SYSTEM.LIBRARY.

(Note: In the current implementation UNITS must reside in *SYSTEM.LIBRARY at the time of compilation in order to be USED by a PASCAL program.)

In order to explicitly invoke the LINKER, the user types 'L' at Command level and receives the prompt:

Host file?

The hostfile is the file into which the routines or UNITS are to be linked. The LINKER appends .CODE to all file names typed in except for *
<ret>. Typing a <ret> in response to the prompt causes the LINKER to use the workfile as the hostfile. The LINKER then asks for the name(s) of the library files in which the UNITS or EXTERNAL routines are to be found:

Lib file? <codefile identifier>

Up to eight library files may be referenced. Typing '*' in response to a request for a libfile name will cause the LINKER to reference *SYSTEM.LIBRARY. The user will be notified about each library file that is successfully opened.

Example: Lib file? * <ret>
Opening *SYSTEM.LIBRARY

For information on LIBRARIES and the LIBRARIAN see Section 4.2.

When all relevant libfile names have been entered the user must type <ret> to proceed. The LINKER will now prompt with:

Map file? <file identifier> <ret>

The LINKER writes the map file to the file requested by the user. The map file contains relevant LINKER info regarding the linking process. Responding with <ret> to this prompt will suspend this option. Note that .TEXT is appended unless a '.' is the last letter of the filename.

The LINKER now reads up all segments required to enable the linking process. The user is now prompted to enter the destination file for the linked code output (this will often be the same file name as that of the host file). Linking will commence after the <ret> following the output file name has been typed. An empty line, <ret> only, causes the output file to be placed in the workfile e.g. *SYSTEM.WRK.CODE.

During the linking process the linker will report on all

segments being linked as well as all external routines being copied into the output codefile. The linking process will be aborted if any required segments or routines are missing or undefined. The user will be informed of their absence with messages as described at the beginning of this section.

1.8.2 NOTES ON LINKER CONVENTIONS AND IMPLEMENTATION

Codefiles may contain up to 16 segments. Block 0 of a codefile contains information regarding name, kind, relative address and length of each code segment. This information is called the segtable, and is represented as a record:

```
RECORD
  DISKINFO: ARRAY[0..15] OF
    RECORD
      CODELENG, CODEADDR: INTEGER
    END

  SEGNAME: ARRAY[0..15] OF PACKED ARRAY[0..7] OF CHAR;

  SEGKIND: ARRAY[0..15] OF (LINKED, HOSTSEG, SEGPROC, UNITSEG,
    SEPRTSEG);

  TEXTADDR: ARRAY[0..15] OF INTEGER;
END
```

CODELENG and CODEADDR give, respectively, the length of the code segment in bytes, and the block address of the code segment. A description of SEGKINDs follows:

- LINKED: The codesegment is fully executable. Either all external references (UNITs or EXTERNALs) have been resolved, or none were present.
- HOSTSEG: the segkind assigned to the outer block of a PASCAL program if the program has external references.
- SEGPROC: the segkind assigned to a PASCAL segment procedure.
- UNITSEG: the segkind assigned to a compiled SEGMENT. (see Section 3.3.1)
- SEPRTSEG: This segkind is assigned to a separately compiled procedure or function. Assembly language codefiles are always of this type, as well as Pascal UNITs which are not SEGMENT UNITs.

For an unlinked code segment (that is, a segment containing unresolved external references) the compiler generates linker information. This information is a series of variable-length records, one for each UNIT, routine or variable which is referenced in, but not defined in the source. The first 8 words of each record contain the following information:

LIENTRY=RECORD

```

NAME: ALPHA;
CASE LITYPE:  LITYPES OF
  UNITREF,
  GLOBREF,
  PUBLREF,
  PRIVREF,
  SEPPREF,
  SEPFREF,
  CONSTREF:
  (FORMAT: OPFORMAT; (format of lientry.name can be
                    any of BIG, BYTE or WORD.)
  NREFS: INTEGER;    (# of references to lientry.name in
                    compiled code segment)
  NWORDS: LCRANGE); (size of privates in words)
GLOBDEF:
  (HOMEPROC: PROCRANGE; (which procedure it occurs in)
  ICOFFSET: ICRANGE); (byte offset in p-code)
PUBLDEF:
  (BASEOFFSET: LCRANGE); (compiler assigned word offset)
CONSTDEF:
  (CONSTVAL: INTEGER); (users defined value)
EXTPROC, EXTFUNC,
SEPPROC, SEPFUNC:
  (SRCPROC: PROCRANGE; (procedure number in source segment)
  NPARAMS: INTEGER); (number of parameters expected)
EOFMARK:
  (NEXTBASELC: LCRANGE) (private var allocation info)
END(lientry);

```

If the LITYPE is one of the first case variant, then following this portion of the record is a list of pointers into the code segment. Each of these pointers is the absolute byte address within the code segment of a reference to the variable, UNIT or routine named in the lientry. These are 8 word records, but only the first NREFS of them are valid.

* ADAPTABLE ASSEMBLER * * Section 1.9 *

Version 1.5 September 1978

Users of UCSD Pascal occasionally need to write and execute small assembly routines written in the language of the host machine. These routines would be used within a Pascal program to provide low-level or time critical facilities. The UCSD Adaptable Assembler (in conjunction with the UCSD Linker) has been designed to meet those needs. The UCSD Pascal Project will be maintaining all our Pascal interpreters using this assembler in the near future. By this process the users of the UCSD Pascal system will become essentially independent of any manufacturer's system software.

This assembler was modelled after The Last Assembler (TLA) developed at the University of Waterloo. The basic concept behind both the TLA and the UCSD Adaptable Assemblers is the use of a central machine independent core that is common to all versions of the assembler. This central core is augmented with machine specific code to handle the peculiarities of each individual machine.

For the 1.5 release PDP-11 and Z80 assemblers will be available. Neither of these adaptations took longer than one person-week of effort.

This document is intended for a reader who is already fluent in at least one assembly language.

1.9.1 USAGE

Before attempting to execute the assembler program for a specific machine, an opcodes file (Z80.OPCODES or 11.OPCODES) must be located on the system disk. The errors file (Z80.ERRORS or 11.ERRORS) contains the error messages that are used for error flagging during the assembly. This file is optional; if used, it must also appear on the system disk.

To use the UCSD assembler, type A[ssem from the Command line. This will execute SYSTEM.ASSEMBLER. (The user should arrange that the right version of the assembler (PDP-11 or Z80) have that title.)

The program displays the version of the assembler being executed and assumes that the current workfile is the one to be assembled. If there is no current workfile then the program asks which file is to be assembled.

The next prompt line is:

Output file for the assembled listing (<CR> for none):

As usual for a console or printer output the words CONSOLE or PRINTER must be followed by a colon, i. e. CONSOLE:. If the colon is neglected the output is sent to a file of the name given. At this point, the program reports whether or not the output device (if any) is on line. The assembled code is written out to a file called *SYSTEM.WRK.CODE which cannot be executed by itself but must be changed to link in with a host file.

The program then starts assembling the workfile, flagging errors as they are found. If an error, other than an I/O error, is found, a general message indicates the nature of the error and also gives the option to continue or exit. The error message will be taken from the ERRORS file if possible. If that is not possible, due to space limitations or the absence of the errors file, the error message number is given. The assembly is aborted if the I/O error encountered is not due to data typed in by the user, otherwise the user is prompted to try again. (See the complete list of Assembler syntax errors and machine specific errors in Table 6.)

The console displays, on the left hand side of the screen, one dot for each line of code assembled and a line counter every 50 lines. When an include file is started, the console displays:

```
.INCLUDE <FILE ID>
```

indicating which file has been included.

At the end of the assembly the assembler program indicates that it is finished and tells the user how many errors were found. In addition an alphabetic symbol table is generated.

The reference symbol table consists of three parts. The first column represents the symbol identifier, the second, the symbol type, and the third, the location that it is defined or the value it has. Actual values are given for the symbols representing absolutes and definition locations are given for the symbols representing labels. The location number is given as a hi-byte first number and corresponds to the index numbers on the left hand side of the listing. Only symbols which have definition locations or absolute values have numbers in the third column; other types have dashes.

Below is an example of an assembled listing with symbol table.

```

0000! .PROC PRIMARYZ
Memory after initialization: 6068
0000!
0000! FLOPPY .EQU 0BFDH ;Rom-based floppy driver
0000! SECMEM .EQU 9000H ;First location in memory of boot
0000! SECENT .EQU 9000H ;Entry point of bootstrap
0000! SECDSK .EQU 08H + 1700H ;Sector start of second bootstrap
0000! B1DSK .EQU 10H + 1700H ;Sector start of BIOS part 1
0000! B2DSK .EQU 18H + 1700H ;Sector start of BIOS part 2
0000! .ORG 1000H ;Primary bootstrap for ZILOG DD

1000!
1000! FD 21 **** PRIMARY LD IY,SECREAD ;Get block for second bootstrap
1004! CD F0B CALL FLOPPY
1007! FD 21 **** LD IY,B1READ ;Get block for part 1 of BIOS
100B! CD F0B CALL FLOPPY
100E! FD 21 **** LD IY,B2READ ;Get block for part 2 of BIOS
1012! CD F0B CALL FLOPPY
1015! C3 0090 JP SECENT ;Jump into second bootstrap
1018!
1002* 1B10
1018! SECREAD
1018! 00 .BYTE $-$ ;Unused
1019! 0A .BYTE 0AH ;Read command
101A! 0090 .WORD SECMEN ;Memory location for second boot

101C! 0002 .WORD 200H ;Number of bytes in boot
101E! 0000 .WORD $-$ ;Completion return address
1020! 0010 .WORD PRIMARY ;Error in return address
1022! 00 .BYTE $-$ ;Completion result code
1023! 0B17 .WORD SECDSK ;Disk block of second boot
1025!
1009* 2510
1025! B1READ
1025! 00 .BYTE $-$ ;Unused
1026! 0A .BYTE 0AH ;Read command
1027! 0093 .WORD SECMEN+300H ;Memory location of BIOS part 1
1029! 0002 .WORD 200H ;Number of bytes in BIOS part 1
102B! 0000 .WORD $-$ ;Completion return address
102D! 0010 .WORD PRIMARY ;Error return address
102F! 00 .BYTE $-$ ;Completion result code
1030! 1017 .WORD B1DSK ;Disk block of BIOS part 1
1032!
1010* 3210
1032! B2READ
1032! 00 .BYTE $-$ ;Unused
1033! 0A .BYTE 0AH ;Read command
1034! 0095 .WORD SECMEN+500H ;Memory location of BIOS part

1036! 0002 .WORD 200H ;Number of bytes in BIOS part 2
1038! 0000 .WORD $-$ ;Completion return address
103A! 0010 .WORD PRIMARY ;Error return address

```

```

103C! 00          .BYTE $-$          ;Completion result code
103D! 1817       .WORD B2DSK        ;Disk block of BIOS part 2
103F!
103F!          .END

```

PAGE- 2 PRIMARYZ FILE: #5: PRIMARY.Z SYMBOLTABLE DUMP

```

AB - Absolute  LB - Label    UD - Undefined  MC - Macro
RF - Ref       DF - Def      PR - Proc       FC - Func
PB - Public    PV - Private  CS - Constant

```

```

B1DSK      AB 1710!  B1READ      LB 1025!  B2DSK      AB 1718!  B2READ      LB 1032!
FLOPPY     AB 08FD!  PRIMARY     LB 1000!  PRIMARYZ   PR ----!  SECDSK      AB 1708!
SECENT     AB 9000!  SECMEM     AB 9000!  SECREAD    LB 1018!

```

NOTES:

The location values in the symbol table dump refer to the locations in the listing.

The ****'s in the listing call attention to the use of a label not yet defined.

If a star (*) appears after the location number at the left of the listing, it indicates that a forward reference occurring earlier in the assembly has been resolved. The number to the left of the '*' is the location where the reference occurred while the number to the right is the new contents of that location.

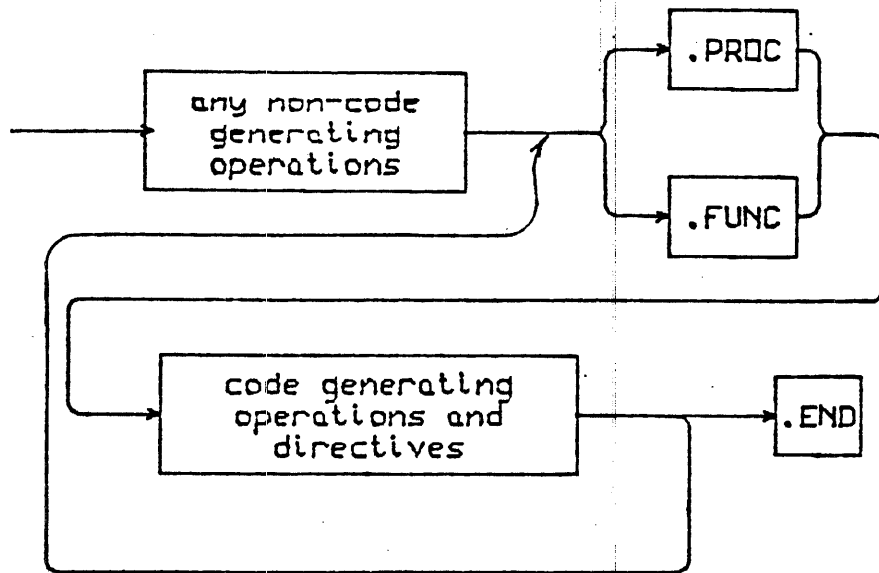
1.9.2 HIGH-LEVEL SYNTAX

All objects declared before the first .PROC or .FUNC are available for use throughout the assembly. No code is allowed to be generated before the first .PROC or .FUNC. The symbol table is reduced at the beginning of each .PROC or .FUNC to the point where it was at the start of the first .PROC or .FUNC.

Only labels may begin in the first column and may optionally be followed by a colon. Local labels must have '\$' in the first column and may be up to 8 digits long. If the statement has no label, the first column must contain a space.

All assemblies must end with a .END. However each .PROC or .FUNC need not because they are ended by the occurrence of the next .PROC or .FUNC. Only the last one needs a .END.

A general railroad diagram for all assembly files looks like:



The non-code generating operations are:

.EQU, .DEF, .REF, .PAGE, .TITLE, .LIST, .MACRO, .IF

The code generating operations are any other pseudo-ops and all assembly code for the program.

1.9.3 EXPRESSIONS (one-pass restrictions)

Since the Adaptable Assembler makes only one pass through the source code, something must be assumed (upon encountering an undefined identifier in an expression) about the nature of the identifier in order for the assembly to continue. It is therefore assumed that the undefined identifier will eventually be defined as a label, which is the most probable case. Any identifier which is not a label must be defined before it is used.

Labels may be equated to an expression containing either labels and/or absolutes. One must define a label before it is used unless it will simply be equated to another label. Local labels may not occur on the left hand side of an equate (.EQU).

Local labels are mainly used to jump around within a small segment of code without having to use up storage area needed by regular labels. The local label stack may hold up to 21 labels. These are cut back every time upon encountering a regular label and are thus rendered invalid. An example of the use of local labels is shown below, the jump to label \$04 being illegal.

```

                $03   STA   4           ; LEGAL USE OF LOCAL LABEL
                .
                JP    NZ,$03
                .
                JP    NZ,$04           ; ILLEGAL USE OF LOCAL LABEL
REALLAB        .EQU $
                $04   .EQU $
```

Identifiers are character strings starting with an alpha character. Other characters must be alphanumeric or the ASCII underline ('_'). Only the first 8 characters are used by the assembler even though more may be entered.

The following operators can be used in expressions processed by this assembler.

For unary operations:

'+' plus
'-' minus
'~' ones complement

For binary operations:

'+' plus
'-' minus
'^' exclusive or
'*' multiplication
'/' truncating division
'%' remainder division
'|' bit wise
'&' bit wise
'=' equal (valid only in .IF)
'<' not equal (valid only in .IF)

All constants must start with an integer 0-9.
All operations are applied to whole words.

The default radix is Hex for the Z80 version and Octal for the PDP-11.

1.9.4 ASSEMBLER DIRECTIVES: OVERVIEW

Assembler directives (also referred to as "pseudo-ops") allow the programmer to instruct the assembler to do various functions other than provide direct executable code. The following directives are common to all UCSD versions but may differ from manufacturer's standard syntax.

In the following pseudo-op descriptions square brackets, [], are used to denote optional elements. If an element type is not listed it cannot be used in that situation. As usual, angle brackets, <>, denote meta symbols.

For example: [label] .ASCII "<character string>"
indicates that a label may be given but is not necessary and that between the double quotes must go the character string to be converted (not necessarily the words "character string").

The following terms represent general concepts in the explanation of each directive:

value = any numerical value, label, constant, expression.

valuelist = is a list of one or more values separated by commas.

idlist = a list of one or more identifiers separated by commas.

expression = any legal expression as defined in Section 1.9.3.

identifier:integer list = a list of one or more identifier-integer pairs separated by commas. The colon-integer is optional in each pair and the default is 1.

Small examples are included after each pseudo-op definition to supply the user with a reference to the specific syntax and form of that directive. The larger example, included in section 3.3.2, is used to show the combined use and detailed examples of directive operations.

1.9.4.1 ROUTINE DELIMITING DIRECTIVES

Every assembly must include at least one .PROC or .FUNC, and one .END, even in the case of stand-alone code which will not be linked into a Pascal host (i. e. an interpreter). The most frequent use of the assembler, however, will be small routines intended to be linked with a Pascal host. In this case, .PROCs and .FUNCS are used to identify and delimit the assembly code to be accessed by a Pascal external procedure or function. The .END appears at the end of the last routine and serves as the final delimiter.

References to a .PROC or .FUNC are made in the Pascal host by use of EXTERNAL declarations. At the time of this declaration the actual parameter names must be given. For example, if the Pascal declaration is:

```
PROCEDURE FARKLE(X,Y:REAL);EXTERNAL;
```

the associated declaration for the .PROC would be

```
.PROC FARKLE,4
```

A .PROC, .FUNC, or any assembly routine should be inserted into the *SYSTEM.LIBRARY (execute LIBRARIAN) so that it can be referenced by the *SYSTEM.LINKER and linked in at run time. An alternate method would be to execute the LINKER and tell it what files to link in. Either method works. However, if the Pascal host is updated and the assembly routines aren't in the *SYSTEM.LIBRARY, the linker will have to be executed after each update. Therefore, we suggest that the routines be inserted into the *SYSTEM.LIBRARY to avoid this repetition. If the

linker is called automatically using the Run command, it will search the *SYSTEM.LIBRARY for the appropriate definition of the assembly routine and link the two together.

.PROC Identifies a procedure that returns no value. A .PROC is ended by the occurrence of a new .PROC, .FUNC, or .END.

FORM: .PROC <identifier>[,<expression>]

[expression] indicates the number of words of parameters expected by this routine. The default is 0.

EXAMPLE: .PROC DLDRIVE,2

.FUNC Identifies a function that returns a value. Two words of space to be used for the function value will be placed on the stack before any parameters. A .FUNC is ended the same way as the .PROC.

FORM: .FUNC <identifier>[,<expression>]

[expression] indicates the number of words of parameters expected by this routine. The default is 0.

EXAMPLE: .FUNC RANDOM,4

.END Used to denote the physical end of an assembly.

1.9.4.2 LABEL DEFINITIONS AND SPACE ALLOCATION DIRECTIVES

.ASCII Converts character values to ASCII equivalent byte constants and places the equivalents into the code stream.

FORM: [label] .ASCII "<character string>"
where <character string> is any string of printable ASCII characters, including a space. The length of the string must be less than 80 characters. The double quotes are used as delimiters for the characters to be converted. If a double quote is desired in the string, it must be specifically inserted using a .BYTE.

EXAMPLE: .ASCII "HELLO"

for the insertion of AB"CD the code must be constructed as:

```
.ASCII "AB"  
.BYTE 34  
.ASCII "CD"
```

Note: The 34 is the ASCII number for a double quote in hex. The representation actually used will depend on the default radix of the particular machine in use.

.BYTE

Allocates a byte of space into the code stream for each value listed. Assigns the associated label, if any, to the address at which the byte was stored. Expression must have a value between -128 and +255. If the value is outside of this range an error will be flagged.

FORM: [label] .BYTE [valuelist]

the default for no stated value is 0.

EXAMPLE: TEMP .BYTE 4

the associated output would be: 04

.BLOCK

Allocates a block of space into code stream for each value listed. Amount allocated is in bytes. Associates the label (if present) with the starting address of the block allocated.

FORM: [label] .BLOCK <length>[.value]

<length> is the the number of bytes to hold the <value> specified. The default for no stated value is 0.

EXAMPLE: TEMP .BLOCK 4,6

the associated output would be:

```
06  
06 ( four bytes with the value 06 )  
06  
06
```

.WORD

Allocates a word of space in the code stream for each value in the valuelist. Associates the declaration label with the word space allocation.

FORM: [label] .WORD <valuelist>

EXAMPLE: TEMP .WORD 0,2,4,...

the associated output would be:

0000
0002
0004 (words with these values in them)

EXAMPLE: L1 .WORD L2

L2 .EGU \$ \$ represents the LC on the Z80
.WORD 5.

if LC was 50 at the .EGU
the associated output would be:

0050 (* assignment due to the L2 value *)
0005 (* assignment due to the .WORD 5 *)

.EGU

Assigns a value to a label. Labels may be equated to an expression containing either labels and/or absolutes. One must define a label before it is used unless it will simply be equated to another label. A local label may not appear on the left hand side of an equate (.EGU).

FORM: <label> .EGU <value>

EXAMPLE: BASE .EGU R6

.ORG

Sets the current location counter (LC) to the value of the .ORG. It would normally be used in a stand-alone program. For example, there is one .ORG in the 8080/Z80 interpreter.

.ORG is currently implemented only for advancing the location counter. It is not currently possible to set the location counter back.

1.9.4.3 MACRO FACILITY DIRECTIVES

A macro is a named section of text that can be defined once and repeated in other places simply by using its name. The text of the macro may be parameterized, so that each invocation results in a different version of the macro contents.

At the invocation point, the macro name is followed by a list of parameters which are delimited by commas (except for the last one, which is terminated by end of line or the comment indication (;)). At invocation time, the text of the macro is inserted (conceptually speaking) by the assembler after being modified by parameter substitution. Whenever %n (where n is a single decimal digit greater than zero) occurs in the macro definition, the text of the nth parameter is substituted. Leading and trailing blanks are stripped from the parameter before the substitution. If a reference occurs in the macro definition to a parameter not provided in a particular invocation, a null string is substituted.

A macro definition may not contain another macro definition. A definition can certainly, however, include macro invocations. This "nesting" of macro invocations is limited to five levels deep.

The expanded macro is always included in the listing file (if listing is enabled at the point of invocation). Macro expansion text is flagged, in the listing, by a '#' just left of each expanded line. Comments occurring in the macro definition are not repeated in the expansion.

.MACRO Indicates the start of a macro and gives it an identifier.

.ENDM Indicates the end point of a MACRO.

```
FORM:      .MACRO <identifier>
           .
           (macro body)
           .
           .ENDM
```

```
EXAMPLE:   .MACRO  HELP
           STA    %1          ; < comment >
           LDA    %2          ; < comment >
           .ENDM
```

The listing where the macro call is made may look like:


```

        HELP  FIRST,SECOND
*       STA  FIRST
*       LDA  SECOND

```

The statement HELP, calls the macro and sends it two parameters, FIRST and SECOND. These parameters are in turn referenced inside the macro using the identifiers X1 for the variable FIRST, and X2 for the variable SECOND.

1.9.4.4 CONDITIONAL ASSEMBLY DIRECTIVES

Conditionals are used to selectively exclude or include sections of code at assembly time. When the assembler encounters an .IF directive, it evaluates the associated expression. In the simplest case, if the expression is false, the assembler simply discards the text until a .ENDC is reached. If there is an .ELSE directive between the .IF and .ENDC directives, the text before the .ELSE is selected if the expression is true, and the text after the .ELSE if the condition is false. The unassembled part of the conditional will not be included in any listing. Conditionals may be nested.

The conditional expression takes one of two forms. The first is the normal arithmetic/logical expression used elsewhere in the assembler. This type of expression is considered false if it evaluates to zero; true otherwise. The second form of conditional expression is comparison for equality or inequality (indicated by '=' and '<>', respectively). One may compare strings, characters, or arithmetic/logical expressions.

```

.IF      Identifies the beginning of the conditional.
.ENDC    Identifies the end of a conditional .IF
.ELSE    Identifies the alternate to the .IF. If the conditional
         expression is equal to 0 then the else is used.

FORM:    [label] .IF <expression>
         .
         .
         .ELSE (* only if there is an else *)
         .
         .
         .ENDC

```

where the expression is the conditional expression to be met.

```
EXAMPLE:      .IF LABEL1-LABEL2      ;arithmetic expression
              ; This text assembled only if subtraction
              ; result is now zero
              .
              .IF "%1" ="STUFF"      ;comparison expression
              ; This text assembled if subtraction above
              ; was true and if text of first parameter
              ; (assume we are in macro ) is equal to "STUFF"
              .ENDC                  ;terminate nested cond.
              .
              .ELSE
              ; This text assembled if subtraction result
              ; was zero
              .
              .
              .ENDC                  ;terminate outer level
                                      ;conditional
```

1.9.4.5 PASCAL HOST COMMUNICATION DIRECTIVES

The directives .CONST, .PUBLIC, and .PRIVATE allow the sharing of information and data space between an assembly routine and a Pascal host. These external references must eventually be resolved by the Linker. Refer to Section 1.8 Linker, for further details.

.CONST Allows access of globally declared constants in the PASCAL host by the assembly routine. .CONST can only be used in a program to replace 16 bit relocatable objects.

FORM: .CONST <idlist>

EXAMPLE: (* see example after .PRIVATE *)

.PUBLIC Allows a variable declared in the global data segment of the PASCAL host to be used by an assembly language routine and the host program.

FORM: .PUBLIC <idlist>

EXAMPLE: (* see example after .PRIVATE *)

.PRIVATE Allows variables of the assembly routine to be stored in the global data segment and yet be inaccessible to the Pascal host. These variables retain their values for the entire execution of the program.

FORM: .PRIVATE <identifier:integer list>

the integer is used to communicate the number of words to be allocated to the identifier.

EXAMPLE: (* for .CONST, .PRIVATE, .PUBLIC *)

Given the following Pascal host program:

```
PROGRAM EXAMPLE;  
CONST SETSIZE=50; LENGTH=80;  
  
VAR I, J, F, HOLD, COUNTER, LDC: INTEGER;  
    LST1: ARRAY[0..9] OF CHAR;  
  
BEGIN  
    ...  
    ...  
    ...  
END.
```

and the following section of an assembly routine:

```
.CONST    LENGTH  
.PRIVATE  PRT, LST2: 9  
.PUBLIC   LDC, I, J
```

This will allow the const LENGTH to be used in the assembly routine almost as if the line LENGTH EQU 80 had been written. (Recall the limitation mentioned above for the use .CONST identifiers.) The variables LDC, I, J to be used by both the Pascal host and the assembly routine, and the variables PRT, LST2 to be used only by the assembly routine. Further, the LST2:9 causes the variable LST2 to correspond with the beginning of a 9 word block of space in the global data segment.

1.9.4.6 EXTERNAL REFERENCE DIRECTIVES

The use of .DEF and .REF is similar to that of .PUBLIC. .DEFs and .REFs associate labels between assembly language routines rather than between an assembly routine and a Pascal host program. Just as with .PRIVATE and .PUBLIC, these external references must eventually be resolved by the Linker. If such resolution cannot be accomplished, the Linker will indicate the offending label. Naturally, the assembler cannot be expected to flag these errors, since it has no knowledge of other assemblies.

.DEF Identifies a label that is defined in the current routine and available to be used in other .PROCs or .FUNCS.

FORM: .DEF <identifierlist>

EXAMPLE: (* see listing in section 3.3.2.3 for example *)

.REF Identifies a label used in this routine which has been declared in an external .PROC or .FUNC with a .DEF. During the linking process, corresponding .DEFs and .REFs are matched.

FORM: .REF <identifierlist>

EXAMPLE: (* see listing in section 3.3.2.3 for example *)

Note: The .PROC and the .FUNC directive also generates a .DEF with the same name. This allows assembly procedures to call .PROC and .FUNCS if they have been defined in a .REF.

1.9.4.7 LISTING CONTROL DIRECTIVES

.LIST Allows selective listing of assembly routines.
& If no output file is declared then the default is CONSOLE:
.NOLIST when a .LIST is encountered. The .NOLIST is used to turn off the .LIST option. Listing may be turned on and off repeatedly within an assembly.

FORM: .LIST or .NOLIST

.PAGE Allows the programmer to explicitly ask for top of form page breaks in the listing.

if no listing output file is specified then all .LIST and .NOLIST directives are simply ignored.

FORM: . PAGE

.TITLE Allows the titling of each page if desired. The title may be up to 80 characters in length. At the start of each procedure the title is set to blanks and must be reset if title is desired. The title,

INTERP SYMBOLTABLE DUMP

shown in Section 1.9.1 was caused by a .TITLE directive. **

FORM .TITLE <title>
where <title> is a string

EXAMPLE .TITLE GRC12 interpreter

1.9.4.8 FILE DIRECTIVES

.INCLUDE Causes the indicated source file to be included at that point.

FORM: .INCLUDE <file identifier.TEXT> where the file identifier is any file to be included. Only spaces are allowed between the end of the file name and the end of the include line.

CORRECT EXAMPLE: .INCLUDE SHORTSTART.TEXT

CORRECT EXAMPLE: .INCLUDE SHORTSTART.TEXT
; calls starter

IN-CORRECT EXAMPLE: .INCLUDE SHORTSTART.TEXT ; calls starter

For a list of general errors and also notes on the Z80 and PDP-11 based machines see Table 6.

** Note: The title is only cleared at the start of the file. In section 1.9.1 the title SYMBOLTABLE DUMP was not set by a .TITLE directive. That heading is always used on pages containing symboltable dumps. Upon assembling a further procedure the heading printed returns to what it was set to before the symboltable dump.

- Notes -

* SYSTEM INTRINSICS * * Section 2.1 *

Version I.5 September 1978

WARNING

Most of the UCSD intrinsics assume that users are fluent in the use of PASCAL and are experienced in the use of the system. Any necessary range or validity checks are the responsibility of the user. Since some of these intrinsics do no checking for range validity, they may easily cause the system to die a horrible death. Those intrinsics which are particularly dangerous are noted as such in their descriptions.

PARAMETERS

Required parameters are listed along with the function/procedure identifier. Optional parameters are in [square brackets]. The default values for these are in {metabackets} on the line below them.

NOTE

Following are some definitions of terms used in these documents. They tend to take the place of formal parameters in the dummy declaration headers that preface each description of a particular routine, or set of routines.

ARRAY : a PACKED ARRAY OF CHARACTERS
BLOCK : one disk block, {512 bytes}

BLOCKS : an INTEGER number of blocks
BLOCKNUMBER : an absolute disk block address

BOOLEAN : any BOOLEAN value
CHARACTER : any expression which evaluates to a character
DESTINATION : a PACKED ARRAY OF CHARACTERS to write into or a STRING, context dependent

EXPRESSION : part or all of an expression, to be specified
FILEID : a file identifier, must be
VAR fileid: FILE OF <type>;
or TEXT;
or INTERACTIVE;
or FILE;

INDEX : an index into a STRING or PACKED ARRAY OF CHARACTERS, context dependent or as specified.

NUMBER : a literal or identifier whose type is either INTEGER or REAL.

RELBLOCK : a relative disk block address, relative to the start of the file in context, the first block being block zero.

SIMPLVARIABLE : any declared PASCAL variable which is of one of the following TYPES:

	DOUBLE	CHAR	REAL	STRING
	or PACKED ARRAY[...] OF CHAR			
SIZE	: an INTEGER number of bytes or characters; any integer value			
SOURCE	: a STRING or PACKED ARRAY OF CHARacters to be used as a read-only array, context dependent or as specified. **			
SCREEN	: an array 2600 bytes long; or as needed.			
STRING	: any STRING, call-by-value unless otherwise specified, i.e. may be a quoted string, or string variable or function which evaluates to a STRING			
TITLE	: a STRING consisting of a file name			
UNITNUMBER	: physical device number used to determine device handler used by the interpreter			
VOLID	: a volume identifier, STRING[7]			

*** i.e. in string intrinsics, SOURCE is going to have to be a string, in intrinsics that deal with packed arrays of characters, it may be either. A word of caution about using STRINGS in intrinsics that expect character arrays, the zeroeth element of the string is the length byte, which may cause the programmer some unexpected problems. (Were he not aware of that fact!) ed.*

* STRING INTRINSICS * * Section 2.1.1 *

Version I.5 September 1978

FUNCTION LENGTH (STRING) : INTEGER

Returns the integer value of the length of the STRING.

Example:

```
GEESTRING := '1234567';  
WRITELN(LENGTH(GEESTRING), ' ', LENGTH(''));
```

Will print:

7 0

FUNCTION POS (STRING , SOURCE) : INTEGER

This function returns the position of the first occurrence of the pattern in SOURCE to be scanned. The INTEGER value of the position of the first character in the matched pattern will be returned; or if the pattern was not found, zero will be returned. Example:

```
STUFF := 'TAKE THE BOTTLE WITH A METAL CAP';  
PATTERN := 'TAL';  
WRITELN(POS(PATTERN, STUFF));
```

Will print:

26

FUNCTION CONCAT (SOURCES) : STRING

There may be any number of source strings separated by commas.

This function returns a string which is the concatenation of all the strings passed to it. Example:

```
SHORTSTRING := 'THIS IS A STRING';  
LONGSTRING := 'THIS IS A VERY LONG STRING.';  
LONGSTRING := CONCAT('START ', SHORTSTRING, '-', LONGSTRING);  
WRITELN(LONGSTRING);
```

Will print:

START THIS IS A STRING-THIS IS A VERY LONG STRING.

FUNCTION COPY (SOURCE , INDEX , SIZE) : STRING

This function returns a string containing SIZE characters copied from SOURCE starting at the INDEXth position in SOURCE.

Example:

```
TL := 'KEEP SOMETHING HERE';   KEPT := COPY(TL, POS('S', TL), 9);
WRITELN(KEPT);
```

Will print:

SOMETHING

PROCEDURE DELETE (DESTINATION , INDEX , SIZE)

This procedure removes SIZE characters from DESTINATION starting at the INDEX specified. Example:

```
OVERSTUFFED := 'THIS STRING HAS FAR TOO MANY CHARACTERS IN IT.';
DELETE(OVERSTUFFED, POS('HAS', OVERSTUFFED)+3, 8);
WRITELN(OVERSTUFFED);
```

Will print:

THIS STRING HAS MANY CHARACTERS IN IT.

PROCEDURE INSERT (SOURCE , DESTINATION , INDEX)

This inserts SOURCE into DESTINATION at the INDEXth position in DESTINATION.

Example:

```
ID := 'INSERTIONS';
MORE := ' DEMONSTRATE';
DELETE(MORE, LENGTH(MORE), 1);
INSERT(MORE, ID, POS('ID', ID));
WRITELN(ID);
```

Will print:

INSERT DEMONSTRATIONS

PROCEDURE STR (LONG , DESTINATION)

This converts the long integer LONG into a string. The resulting string is placed in DESTINATION. See section 3.3.3 for more about the use of long integers.

Example:

```
INTLONG := 102039503;
STR(INTLONG, INTSTRING);
INSERT('.', INTSTRING, PRED(LENGTH(INTSTRING)));
WRITELN('$', INTSTRING);
```

Will print:

```
$1020395.03
```

Note about using strings and string functions:

In order to maintain the integrity of the LENGTH of a string, only string functions or full string assignments should be used to alter strings. Moves and/or single character assignments do not affect the length of a string which means it probably becomes wrong. The individual elements of STRING are of type CHAR and may be indexed 1..LENGTH(STRING). Accessing the string outside this range will have unpredictable results if range-checking is off or cause a run-time error (1) if range checking is on.

- Notes -

* INPUT AND OUTPUT INTRINSICS * * Section 2.1.2 *

Version 1.5 September 1978

PROCEDURE RESET (FILEID, [TITLE]);
PROCEDURE REWRITE (FILEID, TITLE);

These procedures open files for reading and writing and mark the file as open. The FILEID may be any PASCAL structured file, and the TITLE is a string containing any legal file title.

The difference between them is that REWRITE creates a new file on disk for output files; RESET simply marks an already existing file open for I/O. (Note: if the device specified in the title is a non-directory structured device, e.g. PRINTER, then the file is opened for input, output, or both in either case.) If the file was already open, and another RESET or REWRITE is attempted to it, an error will be returned in IDRESULT. The file's state will remain unchanged.

RESET (FILEID) without optional string parameter "rewinds" the file by setting the file pointers back to the beginning (zero th record) of the file. The boolean functions EOF and EOLN will now be set by the implied GET in RESET.

These procedures behave differently with files of type INTERACTIVE. RESET on files of types other than INTERACTIVE will do an initial GET to the file, setting the window variable to the first record in the file (as described in Jensen & Wirth). RESET on a file of type INTERACTIVE will not do an initial GET.

PROCEDURE UNITREAD (UNITNUMBER, ARRAY, LENGTH, [BLOCKNUMBER], [INTEGER]);
PROCEDURE UNITWRITE (UNITNUMBER, ARRAY, LENGTH, [BLOCKNUMBER], [INTEGER]);
{ sequential } { 0 }

THESE ARE DANGEROUS INTRINSICS

These procedures are the low-level procedures which do I/Os to various devices. The UNITNUMBER is the integer name of an I/O device. The ARRAY is any declared packed array, which may be subscripted to indicate a starting position. This is used as the starting address to do the transfers from/to. The LENGTH is an integer value designating the number of bytes to transfer. The BLOCKNUMBER is required only when using a block-structured device (i.e. a disk) and is the absolute blocknumber at which the transfer will start from/to. If the BLOCKNUMBER is left out, 0 is assumed. The INTEGER value is optional (assumed 0) and indicates (if 1) that the transfer is to be done asynchronously. The blocknumber is not necessary. A ',,n' will be sufficient. (See UNITBUSY and UNITWAIT.) (*when using the asynchronous I/O facilities*)

FUNCTION UNITBUSY (UNITNUMBER) : BOOLEAN;

This function returns a BOOLEAN value, indicating if TRUE that the device specified is waiting for an I/O transfer to complete.

Example:

```
UNITREAD(2{non-echoing keyboard},CH[O],
         1{for one character},{no block no.},1{asynchronous});
WHILE UNITBUSY(2){While the READ has not been completed} DO
  WRITELN(OUTPUT,'I am waiting for you to type something');
WRITELN(OUTPUT,'Thank you for typing a ',CH[O]);
```

Execution of this example will continuously type out the line 'I am waiting for you to type something' until a character is struck on the keyboard. Suppose a '!' were typed. The message 'Thank you for typing a !' will then appear, and program execution will proceed normally.

PROCEDURE UNITWAIT (UNITNUMBER);

This waits for the specified device to complete the I/O in progress. It can be simulated by:

```
WHILE UNITBUSY(n) DO {waste a small amount of time};
```

PROCEDURE UNITCLEAR (UNITNUMBER);

UNITCLEAR cancels all I/Os to the specified unit and resets the hardware to its power-up state.

FUNCTION BLOCKREAD (FILEID, ARRAY, BLOCKS, [RELBLOCK]) : INTEGER;

FUNCTION BLOCKWRITE (FILEID, ARRAY, BLOCKS, [RELBLOCK]) : INTEGER;

{ sequential }

These functions return an INTEGER value equal to the number of blocks of data actually transferred. The FILE must be an untyped file (i.e. F: FILE;). The length of ARRAY should be an integer multiple of bytes-per-disk-block. BLOCKS is the number of blocks you want transferred. RELBLOCK is the blocknumber relative to the start of the file, the zeroeth block being the first block in the file. If no RELBLOCK is specified, the reads/writes will be done sequentially. A random access I/O moves the file pointers. CAUTION should be exercised when using these, as the array bounds are not heeded. EOF(FILEID) becomes true when the last block in a file is read.

PROCEDURE CLOSE (FILEID OPTION);

OPTION may be null or ', LOCK', or ', NORMAL', or ', PURGE', or ', CRUNCH'. (Note the commas!)

If OPTION is null then a NORMAL close is done, i.e. CLOSE simply sets the file state to closed. If the file was opened using REWRITE and is a disk file, it is deleted from the directory.

The LOCK option will cause the disk file associated with the FILEID to be made permanent in the directory if the file is on a directory-structured device and the file was opened with a REWRITE; otherwise a NORMAL close is done.

The PURGE option will delete the TITLE associated with the FILEID from the directory. The unit will go off-line if the device is not block structured.

The CRUNCH option is as yet undefined in what it will do.... The intent is to lock a file with the minimum number of blocks of useful information.

All CLOSEs regardless of the option will mark the file closed and will make the implicit variable FILEID^ undefined. CLOSE on a CLOSEd file causes no action.

FUNCTION EOF (FILEID) : BOOLEAN;
FUNCTION EOLN (FILEID) : BOOLEAN;

If (FILEID) is not present, the fileid INPUT is assumed (e.g. IF EOF THEN...). EOLN and EOF return false after the file specified is RESET. They both return true on a closed file. When EOF (FILEID) is true, FILEID^ is undefined. When GET (FILEID) sets FILEID^ to the EOLN character or the EOF character, EOLN (FILEID) will return true, and FILEID^ (in a FILE OF CHAR) will be set to a blank. If, while doing puts or writes at the end of a file, the file cannot be expanded to accommodate the PUT or WRITE, EOF(FILEID) will return true.

FUNCTION IORESULT : INTEGER;

After any I/O operation, IORESULT contains an INTEGER value corresponding to the values given in Table 2.

```
PROCEDURE GET ( FILEID );  
PROCEDURE PUT ( FILEID );
```

These procedures are used for operations on typed files. A typed file is any file for which a type is specified in the variable declaration, ie. 'FILEID : FILE OF <type>'. This is as opposed to untyped files which are simply declared as: 'FILEID: FILE;'. 'F: FILE OF CHAR' is equivalent to 'F: TEXT'. In a typed file each logical record is a memory image fitting the description of a variable of the associated <type>.

GET (FILEID) will leave the contents of the current logical record pointed at by the file pointers in the implicitly declared "window" variable FILEID^ and increment the file pointers.

PUT (FILEID) puts the contents of FILEID^ into the file at the location of the current file pointers and then updates those pointers.

```
PROCEDURE READ<LN> ( FILEID, SOURCE );  
PROCEDURE WRITE<LN> ( FILEID, SOURCE );
```

These procedures may be used only on TEXT (FILE OF CHAR) or INTERACTIVE files for I/O. If 'FILEID,' is omitted, INPUT or OUTPUT (whichever is appropriate) is assumed. A READ(String) will read up to and not including the end-of-line character (<a carriage return>) and leave EDLN(FILEID) true. This means that any subsequent READs of STRING variables will return the null string until a READLN or READ(character) is executed.

There are three files of type INTERACTIVE which are predeclared: INPUT, OUTPUT, and KEYBOARD. INPUT results in echoing of characters typed to the console device. KEYBOARD does no echoing and allows the programmer complete control of the response to user typing. OUTPUT allows the user to halt or flush the output.

```
PROCEDURE PAGE ( FILEID );
```

This procedure, as described in Jensen & Wirth (ibid.), sends a top-of-form (ASCII FF) to the file.

```
PROCEDURE SEEK ( FILEID, INTEGER );
```

This procedure changes the file pointers so that the next GET or PUT from/to the file uses the INTEGERth record of FILEID. Records in files are numbered from 0. A GET or PUT must be executed between SEEK calls since two SEEKS in a row may cause unexpected, unpredictable junk to be held in the window and associated buffers.

* CHARACTER ARRAY MANIPULATIONS INTRINSICS * * Section 2.1.5 *

Version 1.5 September 1978

CAUTION

These intrinsics are all byte oriented. Use them with care. Read the descriptions carefully before trying them out as no range checking of any sort is performed on the parameters passed to these routines. The programmer should know exactly what he is doing before he does it since the system does not protect itself from these operations.

FUNCTION SCAN (LENGTH, PARTIAL EXPRESSION, ARRAY) : INTEGER;

This function returns the number of characters from the starting position to where it terminated. It terminates on either matching the specified LENGTH or satisfying the EXPRESSION. The ARRAY should be a PACKED ARRAY OF CHARACTERS and may be subscripted to denote the starting point. If the expression is satisfied on the character at which ARRAY is pointed, the value returned will be zero. If the length passed was negative, the number returned will also be negative, and the function will have scanned backward. The PARTIAL EXPRESSION must be of the form:

"<" or "=" followed by <character expression>

Examples:

Using the array:

DEM := '.....THE TERAK IS A MEMBER OF THE PTERODACTYL FAMILY.'

SCAN(-26, '=' , DEM[30]); will return -26
SCAN(100, '<' , DEM); will return 5
SCAN(15, '=' , DEM[0]); will return 8

PROCEDURE MOVELEFT (SOURCE, DESTINATION, LENGTH);
PROCEDURE MOVERIGHT (SOURCE, DESTINATION, LENGTH);

These functions do mass moves of bytes for the length specified. MOVELEFT starts from the left end of the specified source and moves bytes to the left end of the destination. MOVERIGHT starts from the right ends of both arrays and also moves byte by byte.

These procedures will optimize to word moves (in the 11 version) if at all possible. MOVERIGHT never attempts this optimization; MOVELEFT will optimize only if the destination is at an address below the I/O page. (The reason for not doing word moves to the I/O page is that some hardware relies on byte addressing in this address space.)

In short: MOVELEFT starts at the left end of both arrays and copies bytes traveling right. MOVERIGHT starts at the right end of both arrays and copies bytes traveling left. The reason for having both of these is if you are working in a single array and the order in which characters are moved is critical. The following chart is an attempt to show what happens if you use the procedure which moves in the wrong direction for your purposes.

```
VAR ARRAY: PACKED ARRAY [1..30] OF CHAR;

(*123456789a123456789b123456789c*)
ARRAY: !THIS IS THE TEXT IN THIS ARRAY!
      MOVERIGHT(ARRAY[10],ARRAY[1],10);
ARRAY: !NE TEXT INE TEXT IN THIS ARRAY!
      MOVELEFT(ARRAY[1],ARRAY[3],10)
ARRAY: !NENENENENETEXT IN THIS ARRAY!
      MOVELEFT(ARRAY[23],ARRAY[2],8);
ARRAY: !NIS ARRAYENETEXT IN THIS ARRAY!
```

```
PROCEDURE FILLCHAR ( DESTINATION, LENGTH, CHARACTER );
```

This procedure takes a (subscripted) PACKED ARRAY OF CHARACTERS and fills it with the number (LENGTH) of CHARACTERs specified. This can be done by:

```
A[0] := <character expression>;
MOVELEFT(A[0],A[1],n-1);
```

but FILLCHAR is twice as fast, as no memory reference is needed for a source.

See the note about word move optimization in the section on MOVELEFT. The notes about MOVELEFT also apply to FILLCHAR.

The intrinsic SIZEOF (Section 2.1.6) is meant for use with these intrinsics; it is convenient not to have to figure out or remember the number of bytes in a particular data structure.

* MISCELLANEOUS ROUTINES * * Section 2.1.6 *

Version I.5 September 1978

FUNCTION SIZEOF (VARIABLE OR TYPE IDENTIFIER) : INTEGER;

This function returns the number of bytes that the "item" passed as a parameter occupies in the stack. SIZEOF is particularly useful for FILLCHAR and MOVExxxx intrinsics.

FUNCTION LOG (NUMBER) : REAL;

This function returns the log base ten of the NUMBER passed as a parameter.

PROCEDURE TIME (VAR HIWORD, LOWORD: INTEGER);

This procedure returns the current value of the system clock. It is in 60ths of a second. (This is somewhat hardware-dependent; we assume a 16-bit integer size and 32-bit clock word. The HIWORD contains the most significant portion. WARNING! The sign of the LOWORD may be negative since the time is represented as a 32-bit unsigned number.) Both HIWORD and LOWORD must be VARIABLES of type INTEGER.

FUNCTION PWOFTEN (EXPONENT: INTEGER) : REAL;

This function returns the value of 10 to the EXPONENT power. EXPONENT must be an integer in the range 0..37.

PROCEDURE MARK (VAR HEAPPTR: ^INTEGER)
PROCEDURE RELEASE (VAR HEAPPTR: ^INTEGER);

These procedures are used for returning dynamic memory allocations to the system. HEAPPTR is of type ^INTEGER. MARK sets HEAPPTR to the current top-of-heap. RELEASE sets top-of-heap pointer to HEAPPTR.

PROCEDURE HALT;

This procedure generates a HALT opcode that, when executed, causes a non-fatal run-time error to occur. At this point in execution, the Debugger is invoked, therefore, if the Debugger is not in core when this occurs, a fatal run-time error, #14, will occur.

PROCEDURE GOTOXY(XCOORD , YCOORD);

This procedure sends the cursor to the coordinates specified by (XCOORD, YCOORD). The upper left corner of the screen is assumed to be (0,0). This procedure is written to default to a Datamedia-type terminal. If your system uses other than a Datamedia or Terak 8510a, you will need to bind in a new GOTOXY using the GOTOXY package described in Section 4.10.

Version I.5 September 1978

This section is a summary and quick reference guide which notes the areas in which U. C. S. D. Pascal differs from the Standard Pascal, and refers the user to the appropriate documents which explain various aspects of U. C. S. D. Pascal. The Standard Pascal referred to by this section is defined in PASCAL USER MANUAL AND REPORT (2nd edition) by Kathleen Jensen and Niklaus Wirth (Springer-Verlag, 1975).

Many of the differences lie in the area of FILES and I/O in general. It is recommended that the reader first concentrate upon the sections which describe the differences associated with the standard procedures EOF, EOLN, READ, WRITE, RESET, and REWRITE.

2.2.1 CASE STATEMENTS

Jensen and Wirth on page 31, state that if there is no label equal to the value of the case statement selector, the result of the case statement is undefined. U. C. S. D. Pascal defines that if there is no label matching the value of the case selector then the next statement executed is the statement following the case statement. For example, the following sample program will only output the line "THAT'S ALL FOLKS" since the case statement will "fall through" to the WRITELN statement following the case statement:

```
PROGRAM FALLTHROUGH;  
VAR CH: CHAR;  
BEGIN  
  CH:='A';  
  CASE CH OF  
    'B': WRITELN(OUTPUT, 'HI THERE');  
    'C': WRITELN(OUTPUT, 'THE CHARACTER IS A 'C'');  
  END;  
  WRITELN(OUTPUT, 'THAT'S ALL FOLKS');  
END.
```

Contrary to the syntax diagrams for <field list> on pages 116-118 of Jensen and Wirth, the U. C. S. D. Pascal compiler will not permit a semicolon before the "END" of a case variant field declaration within a RECORD declaration. See Table 6 for revised syntax diagrams for <field list>.

2.2.2 COMMENTS

The U.C.S.D. Pascal compiler recognizes any text appearing between either the symbols "(*" and "*)" or the symbols "{" and "}" as a comment. Text appearing between these symbols is ignored by the compiler unless the first character of the comment is a dollarsign, in which case the comment is interpreted as a compiler control comment. See section 1.6 "Pascal Compiler" for details on compiler control comments.

Note that if the beginning of the comment is delimited by the "(*" symbol, the end of the comment must be delimited by the matching "*)" symbol, rather than the ")" symbol. When the comment begins with the "{" symbol) the comment continues until the matching "}" symbol appears. This feature allows a user to "comment out" a section of a program which itself contains comments. For example:

```
(      XCP := XCP + 1;  (* ADJUST FOR SPECIAL CASE... *)  )
```

Note that the compiler does not keep track of nested comments. When a comment symbol is encountered, the text is scanned for the matching comment symbol. The following text will result in a syntax error:

```
(* THIS IS A COMMENT  (* NESTED COMMENT *)  END OF FIRST COMMENT *)  
                                ^error here.
```

2.2.3 DYNAMIC MEMORY ALLOCATION

The standard procedure DISPOSE defined on page 158 of Jensen and Wirth is not implemented in U.C.S.D. Pascal. However, the function of DISPOSE can be approximated by a combined use of the U.C.S.D. intrinsics MARK and RELEASE. The process of recovering memory space described below is only an approximation to the function of DISPOSE as one cannot explicitly ask that the storage occupied by one particular variable be released by the system for other uses.

The current U.C.S.D. implementation allocates storage for variables created by use of the standard procedure NEW in a stack-like structure called the "heap". The following program is a simple demonstration of how MARK and RELEASE can be used to change in the size of the heap.

```
PROGRAM SMALLHEAP;  
  
TYPE PERSON=  
    RECORD  
        NAME: PACKED ARRAY[0..15] OF CHAR;  
        ID: INTEGER  
    END;
```

```

VAR P: ^PERSON; (* "^" means "pointer to" as defined in J&W *)
    HEAP: ^INTEGER;

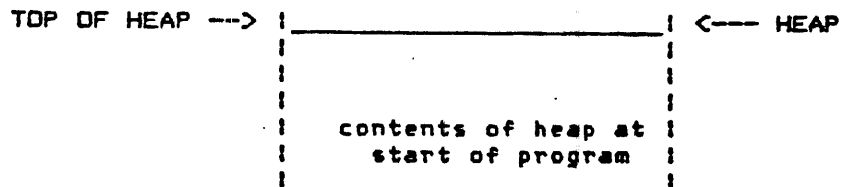
```

```

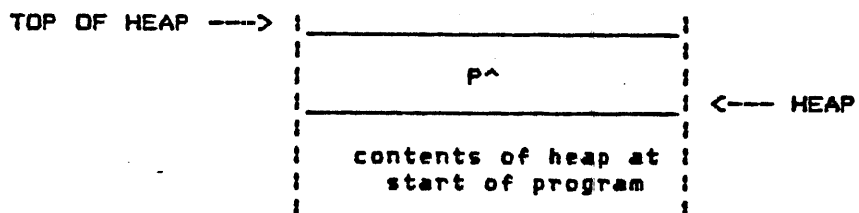
BEGIN
    MARK(HEAP);
    NEW(P);
    P^.NAME := 'FARKLE, HENRY J.';
    P^.ID := 999;
    RELEASE(HEAP);
END.

```

The above program first calls MARK to place the address of the current top of heap into the variable HEAP. HEAP being declared to be a pointer to an INTEGER is not really important, as HEAP could have been declared as pointing to almost anything. The parameter supplied to MARK must be a pointer variable, but need not be a pointer that is declared to be a pointer to an INTEGER. This is a particularly handy construct for deliberately accessing the contents of memory which is otherwise inaccessible. Below is a pictorial description of the heap at this point in the program's execution:



Next the program calls the standard procedure NEW and this results in a new variable P^ which is located in the heap as shown in the diagram below:



Once the program no longer needs the variable P[^] and wishes to "release" this memory space to the system for other uses, it calls RELEASE which resets the top of heap to the address contained in the variable HEAP.

If the above sample program had made a series of calls to the standard procedure NEW between the calls to MARK and RELEASE, the storage occupied by several variables would have been released at once. Note that due to the stack nature of the heap it is not possible to release the memory space used by a single item in the middle of the heap. It is for this reason the use of MARK and RELEASE can only approximate the function of DISPOSE as described in Jensen and Wirth.

Furthermore, it should be noted that careless use of the intrinsics MARK and RELEASE can lead to "dangling pointers", pointing to areas of memory which are no longer part of the defined heap space.

2.2.4 EOF(F)

To set EOF to TRUE for a textfile F being used as an input file from the CONSOLE device, the user must type the EOF character. The system default EOF character is the control-C character. The EOF character can be altered by a suitable reconfiguration of the system variable SYSCOM[^].CRTINFO.EOF using SETUP. For further information concerning system configuration and the SETUP program see Section 4.3.

If F is closed, for any FILE F, EOF(F) will return the value TRUE. If EOF(F) is TRUE, and F is a FILE of type TEXT, EOLN(F) is also TRUE. After a RESET(F), EOF(F) is FALSE. If EOF(F) becomes TRUE during a GET(F) or a READ(F,...) the data obtained thereby is not valid.

When a user program starts execution, the system performs a RESET on the predeclared files INPUT, OUTPUT, and KEYBOARD. See section 2.2.11 READ for further details concerning the predeclared file KEYBOARD.

As defined in Jensen and Wirth, EOF and EOLN by default will refer to the file INPUT if no file identifier is specified.

2.2.5 EOLN(F)

EOLN(F) is defined only if F is a textfile. F is a textfile if the <type> of the window variable, F[^], is of type CHAR. EOLN becomes TRUE only after reading the end of line character. The end of line character is a carriage return. In the example program below, care must be taken as regards when the carriage return is typed while inputting data:


```

PROGRAM ADDLINES;
VAR K, SUM: INTEGER;

BEGIN
  WHILE NOT EOF(INPUT) DO
    BEGIN
      SUM:=0;
      READ(INPUT,K);
      WHILE NOT EOLN(INPUT) DO
        BEGIN
          SUM:=SUM+K;
          READ(INPUT,K);
        END;
      WRITELN(OUTPUT);
      WRITELN(OUTPUT, 'THE SUM FOR THIS LINE IS ', SUM);
    END;
  END.

```

In order for EOLN(F) to be TRUE in the above program, the carriage return must be typed immediately after the last digit of the last integer on that line. If instead a space is typed followed by the carriage return, EOLN will remain FALSE and another READ will take place.

2.2.6 FILES

Changes were made in order to bring U.C.S.D. Pascal closer to the standard definition of the language.

A. INTERACTIVE FILES

Files of <type> INTERACTIVE behave exactly as files of <type> TEXT. The standard predeclared files INPUT and OUTPUT will always be defined to be of <type> INTERACTIVE. All files of any <type> other than INTERACTIVE, are defined to operate exactly as described in Jensen and Wirth. For files which are not of <type> INTERACTIVE, the definitions of EOF(F), EOLN(F), and RESET(F) are exactly as presented in Jensen and Wirth. For more details concerning files of <type> INTERACTIVE see section 2.2.11 "READ AND READLN" and section 2.2.12 "RESET" and section 2.1.2..

B. UNTYPED FILES

U.C.S.D. Pascal has one type of file declaration which is not found in the syntax of Jensen and Wirth. This type and its use is demonstrated in the sample program below:

```

(*$I--*)
PROGRAM FILEDEMO;

VAR G, F: FILE;
    BUFFER: , PACKED ARRAY[0..511] OF CHAR;
    BLOCKNUMBER, BLOCKSTRANSFERRED: INTEGER;
    BADIO: BOOLEAN;

(* This program reads a diskfile called 'SOURCE.DATA' and
copies the file into another diskfile called 'DESTINATION'
using untyped files and the intrinsics BLOCKREAD and
BLOCKWRITE *)

BEGIN
    BADIO:=FALSE;
    RESET(G, 'SOURCE.DATA');
    REWRITE(F, 'DESTINATION');
    BLOCKNUMBER:=0;
    BLOCKSTRANSFERRED:=BLOCKREAD(G, BUFFER, 1, BLOCKNUMBER);
    WHILE (NOT EOF(G)) AND (IORESULT=0) AND (NOT BADIO) AND
        (BLOCKSTRANSFERRED=1) DO
        BEGIN
            BLOCKSTRANSFERRED:=BLOCKWRITE(F, BUFFER, 1, BLOCKNUMBER);
            BADIO:=((BLOCKSTRANSFERRED<1) OR (IORESULT<>0));
            BLOCKNUMBER:=BLOCKNUMBER+1;
            BLOCKSTRANSFERRED:=BLOCKREAD(G, BUFFER, 1, BLOCKNUMBER);
        END;
    CLOSE(F, LOCK);
END.

```

The two files which are declared and used in the above sample program are both untyped files. An untyped file F can be thought of as a file without a window variable F[^] to which all I/O must be accomplished by using the functions BLOCKREAD and BLOCKWRITE. Note that any number of blocks can be transferred using either BLOCKREAD or BLOCKWRITE. The functions return the actual number of blocks read. A somewhat sneaky approach to doing a quick transfer would be:

```
WHILE BLOCKWRITE(F, BUFFER, BLOCKREAD(G, BUFFER, BUFBLOCKS))>0 DO (*IT*);
```

This is, however considered unclean. The program above has been compiled using the I-Compile Time Option, thereby requiring that the function IORESULT and the number of blocks transferred be checked after each BLOCKREAD or BLOCKWRITE in order to detect any I/O errors that might have occurred.

C. RANDOM ACCESS OF FILES

The U.C.S.D. implementation of structured files supports the ability to randomly access individual records within a file by means of the intrinsic SEEK. SEEK expects two parameters, the first being the file identifier, and the second, an integer specifying the record number to which the window should be moved. The first record of a structured file is numbered record 0. The following sample program demonstrates the use of SEEK to randomly access and update records in a file:

```
PROGRAM RANDMACCESS;
VAR DISK: FILE OF
    RECORD
        NAME: STRING[20];
        DAY, MONTH, YEAR: INTEGER;
        ADDRESS: PACKED ARRAY[0..49] OF CHAR;
        ALIVE: BOOLEAN
    END;
    RECNUMBER: INTEGER;
    CH: CHAR;

BEGIN
    RESET(DISK, 'RECORDS.DATA');
    WHILE NOT EOF(INPUT) DO
        BEGIN
            WRITE(OUTPUT, 'Enter record number --->');
            READ(INPUT, RECNUMBER);
            SEEK(DISK, RECNUMBER);
            GET(DISK);
            WITH DISK^ DO
                BEGIN
                    WRITELN(OUTPUT, NAME, DAY, MONTH, YEAR, ADDRESS);
                    WRITE(OUTPUT, 'Enter correct name --->');
                    READLN(INPUT, NAME);
                    ...
                    ...
                    ...
                END;

            SEEK(DISK, RECNUMBER); (* Must point the window
                                   back to the record since
                                   GET(DISK) advances the
                                   window to the next record
                                   after loading DISK^ *)

            PUT(DISK);
        END;
    END.
```

Attempts to PUT records beyond the physical end of file will set EDF to the value TRUE. (The physical end of file is the point where the next record in the file will overwrite another file on the disk.) SEEK always sets EDF and EOLN to FALSE. The subsequent GET or PUT will set these conditions as is appropriate.

D. READ AND WRITE FROM ARBITRARILY TYPED FILES

It is not currently possible to READ or WRITE to files of type other than TEXT or FILE OF CHAR.

2.2.7 GOTO AND EXIT STATEMENTS

U.C.S.D. has a more limited form of GOTO statement than is defined as the standard in Jensen and Wirth. U.C.S.D.'s GOTO statement prohibits a GOTO statement to a label which is not within the same block as the GOTO statement itself. The examples presented on pages 31-32 of Jensen and Wirth are not legal in U.C.S.D. Pascal.

EXIT is a U.C.S.D. extension which accepts as its single parameter the identifier of a procedure to be exited. Note that the use of an EXIT statement to exit a FUNCTION can result in the FUNCTION returning undefined values if no assignment to the FUNCTION identifier is made prior to the execution of the EXIT statement. Below is an example of the use of the EXIT statement:

```
PROGRAM EXITDEMO;
VAR T: STRING;
    CN: INTEGER;

PROCEDURE G; FORWARD;

PROCEDURE P;
BEGIN
  READLN(T);
  WRITELN(T);
  IF T[1]='#' THEN EXIT(G);
  WRITELN('LEAVE P');
END;

PROCEDURE Q;
BEGIN
  P;
  WRITELN('LEAVE Q');
END;
```

```

PROCEDURE R;
BEGIN
  IF CN <= 10 THEN G;
  WRITELN('LEAVE R');
END;

BEGIN
  CN: =0;
  WHILE NOT EOF DO
  BEGIN
    CN: =CN+1;
    R;
    WRITELN;
  END;
END.

```

If the above program were supplied the following input

```

THIS IS THE FIRST STRING
*
LAST STRING

```

the following output will result:

```

THIS IS THE FIRST STRING
LEAVE P
LEAVE G
LEAVE R

*
LEAVE R

LAST STRING
LEAVE P
LEAVE G
LEAVE R

```

The EXIT(G) statement causes the PROCEDURE P to be terminated followed by the PROCEDURE G. Processing continues following the call to G inside PROCEDURE R. Thus the only line of output following "*" is "LEAVE R" at the end of PROCEDURE R. In the two cases where the EXIT(G) statement is not executed, processing proceeds normally through the terminations of procedures P and G.

If the procedure identifier passed to EXIT is a recursive procedure, the most recent invocation of that procedure will be exited. If, in the above example, one or both of the procedures P and G declared and opened some local files, an implicit CLOSE(F) is done when the EXIT(G) statement is executed, as if the procedures P and G terminated normally.

The creation of the EXIT statement at U.C.S.D. was inspired by the occasional need for a straightforward means to abort a complicated and possibly deeply nested series of procedure calls upon encountering an error. An example of such a use of the EXIT statement can be found in the recursive descent U.C.S.D. Pascal compiler. The routine use of the EXIT statement is, nevertheless, discouraged.

2.2.8 PACKED VARIABLES

A. PACKED ARRAYS

The U.C.S.D. compiler will perform packing of arrays and records if the ARRAY or RECORD declaration is preceded by the word PACKED. For example, consider the following declarations:

A: ARRAY[0..9] OF CHAR;

B: PACKED ARRAY[0..9] OF CHAR;

The array A will occupy ten 16 bit words of memory, with each element of the array occupying 1 word. The PACKED ARRAY B on the other hand will occupy a total of only 5 words, since each 16 bit word contains two 8 bit characters. In this manner each element of the PACKED ARRAY B is 8 bits long.

PACKED ARRAYS need not be restricted to arrays of type CHAR, for example:

C: PACKED ARRAY[0..1] OF 0..3;

D: PACKED ARRAY[1..9] OF SET OF 0..15;

D2: PACKED ARRAY[0..239,0..319] OF BOOLEAN;

Each element of the PACKED ARRAY C is only 2 bits long, since only 2 bits are needed to represent the values in the range 0..3. Therefore C occupies only one 16 bit word of memory, and 12 of the bits in that word are unused. The PACKED ARRAY D is a 9 word array, since each element of D is a SET which can be represented in a minimum of 16 bits. Each element of a PACKED ARRAY OF BOOLEAN, as in the case of D2 in the above example, occupies only one bit.

The following 2 declarations are not equivalent due to the recursive nature of the compiler:

E: PACKED ARRAY[0..9] OF ARRAY[0..3] OF CHAR;

F: PACKED ARRAY[0..9,0..3] OF CHAR;

The second occurrence of the reserved word ARRAY in the declaration of E causes the packing option in the compiler to be turned off. E becomes an unpacked array of 40 words. On the otherhand, the PACKED ARRAY F occupies 20 total words because the reserved word ARRAY occurs only once in the declaration. If E had been declared as

```
E: PACKED ARRAY[0..9] OF PACKED ARRAY[0..3] OF CHAR;
```

or as

```
E: ARRAY[0..9] OF PACKED ARRAY[0..3] OF CHAR;
```

F and E would have had identical configurations.

The reserved word PACKED only has true significance before the last appearance of the reserved word ARRAY in a declaration of a PACKED ARRAY. When in doubt a good rule of thumb when declaring a multidimensional PACKED ARRAY is to place the reserved word PACKED before every appearance of the reserved word ARRAY to insure that the resultant array will be PACKED.

The resultant array will only be packed if the final type of the array is scalar, or subrange, or a set which can be represented in 8 bits or less. The final type can also be BOOLEAN or CHAR. The following declaration will result in no packing whatsoever because the final type of the array cannot be represented in a field of 8 bits:

```
G: PACKED ARRAY[0..3] OF 0..1000;
```

G will be an array which occupies 4 16 bit words.

Packing never occurs across word boundaries. This means that if the type of the element to be packed requires a number of bits which does not divide evenly into 16, there will be some unused bits at the high order end of each of the words which comprise the array.

Note that a string constant may be assigned to a PACKED ARRAY OF CHAR but not to an unpacked ARRAY OF CHAR. Likewise, comparisons between an ARRAY OF CHAR and a string constant are illegal. (These are temporary implementation restrictions which will be removed in the next major release.) Because of their different sizes, PACKED ARRAYS cannot be compared to ordinary unpacked ARRAYS. For further information regarding PACKED ARRAYS OF CHARACTERS see section 2.2.16 "STRINGS".

A PACKED ARRAY OF CHAR may be output with a single write statement:

```
PROGRAM VERYSLICK;  
VAR T: PACKED ARRAY[0..10] OF CHAR;  
BEGIN  
  T := 'HELLO THERE';  
  WRITELN(T);  
END.
```

Initialization of a PACKED ARRAY OF CHAR can be accomplished very efficiently by using the U. C. S. D. intrinsics FILLCHAR and SIZEOF:

```
PROGRAM FILLFAST;  
VAR A: PACKED ARRAY[0..10] OF CHAR;  
BEGIN  
  FILLCHAR(A[0], SIZEOF(A), ' ');  
END.
```

The above sample program fills the entire PACKED ARRAY A with blanks. For further documentation on FILLCHAR, SIZEOF, and the other U. C. S. D. intrinsics see section 2.1.5 "CHARACTER ARRAY MANIPULATION INTRINSICS".

B. PACKED RECORDS

The following RECORD declaration declares a RECORD with 4 fields. The entire RECORD occupies one 16 bit word as a result of declaring it to be a PACKED RECORD.

```
VAR R: PACKED RECORD  
  I, J, K: 0..31;  
  B: BOOLEAN  
END;
```

The variables I, J, K each take up 5 bits in the word. The boolean variable B is allocated to the 16'th bit of the same word.

In much the same manner that PACKED ARRAYS can be multidimensional PACKED ARRAYS, PACKED RECORDS may contain fields which themselves are PACKED RECORDS or PACKED ARRAYS. Again, slight differences in the way in which declarations are made will affect the degree of packing achieved. For example, note that the following two declarations are not equivalent:

```
VAR A: PACKED RECORD  
  C: INTEGER;  
  F: PACKED RECORD  
    R: CHAR;  
    K: BOOLEAN  
  END;  
  H: PACKED ARRAY[0..3] OF CHAR  
END;
```

```
VAR B: PACKED RECORD  
  C: INTEGER;  
  F: RECORD  
    R: CHAR;  
    K: BOOLEAN  
  END;  
  H: PACKED ARRAY[0..3] OF CHAR  
END;
```

As with the reserved word ARRAY, the reserved word PACKED must appear with every occurrence of the reserved word RECORD in order for the PACKED RECORD to retain its packed qualities throughout all fields of the RECORD. In the above example, only RECORD A has all of its fields packed into one word. In B, the F field is not packed and therefore occupies two 16 bit words. It is important to note that a packed or unpacked ARRAY or RECORD which is a field of a PACKED RECORD will always start at the beginning of the next word boundary. This means that in the case of A, even though the F field does not completely fill one word, the H field starts at the beginning of the

next word boundary.

A case variant may be used as the last field of a PACKED RECORD, and the amount of space allocated to it will be the size of the largest variant among the various cases. The actual nature of the packing is far beyond the scope of this document.

```
VAR K: PACKED RECORD
      B: BOOLEAN;
      CASE F: BOOLEAN OF
        TRUE: (Z: INTEGER);
        FALSE: (M: PACKED ARRAY[0..3] OF CHAR)
      END
    END;
```

In the above example the B and F fields are stored in two bits of the first 16 bit word of the record. The remaining 14 bits are not used. The size of the case variant field is always the size of the largest variant, so in the above example, the case variant field will occupy two words. Thus the entire PACKED RECORD will occupy 3 words.

C. USING PACKED VARIABLES AS PARAMETERS

No element of a PACKED ARRAY or field of a PACKED RECORD may be passed as a variable (call-by-reference) parameter to a PROCEDURE or FUNCTION. Packed variables may, however, be passed as call by value parameters, as stated in Jensen and Wirth.

D. PACK AND UNPACK STANDARD PROCEDURES

U.C.S.D. Pascal does not support the standard procedures PACK and UNPACK as defined in Jensen and Wirth on page 106.

2.2.9 PARAMETRIC PROCEDURES AND FUNCTIONS

U.C.S.D. Pascal does not support the construct in which PROCEDURES and FUNCTIONS may be declared as formal parameters in the parameter list of a PROCEDURE or FUNCTION.

See Section 6.6 for a revised syntax diagram of <parameter-list>.

2.2.10 PROGRAM HEADINGS

Although the U.C.S.D. Pascal compiler will permit a list of file parameters to be present following the program identifier, these parameters are ignored by the compiler and will have no effect on the program being compiled. As a result the following two program headings are equivalent:

PROGRAM DEMO(INPUT,OUTPUT); and PROGRAM DEMO;

With either of the above program headings, a user program will have three files predeclared and opened by the system. These are: INPUT, OUTPUT, and KEYBOARD and are defined to be of <type> INTERACTIVE. If the program wishes to declare any additional files, these file declarations must be declared together with the program's other VAR declarations.

2.2.11 READ AND READLN

Given the following declarations:

```
VAR CH: CHAR;  
    F: TEXT; (* TYPE TEXT = FILE OF CHAR *)
```

the statement READ(F,CH) is defined by Jensen and Wirth on page 85 to be equivalent to the two-statement sequence:

```
CH:=F^;  
GET(F);
```

In other words, the standard definition of the standard procedure READ requires that the process of opening a file load the "window variable" F^ with the first character of the file. In an interactive programming environment, it is not convenient to require a user to type in the first character of the input file at the time when the file is opened. If this were the case, every program would "hang" until a character was typed, whether or not the program performed any input operations at all. In order to overcome this problem, U.C.S.D. Pascal defines an additional file <type> called INTERACTIVE. Declaring a file F to be of <type> INTERACTIVE is equivalent to declaring F to be of type TEXT, the difference being that the definition of the statement READ(F,CH) is the reverse of the sequence specified by the standard definition for files of <type> TEXT: i. e.

```
GET(F);  
CH:=F^;
```

This difference affects the way in which EOLN must be used within a program when reading from a textfile of type INTERACTIVE. As in section 5, EOLN becomes true only after reading the end of line character, a carriage return. When this is read, EOLN is set to true and the character returned as a result of the READ will be a blank. In the following example, the left fragment is taken from Jensen and Wirth; only the RESET and REWRITE statements have been altered. The program on the left will correctly copy the textfile represented by the file X to the file Y. The program fragment on the right performs a similar task, except that the source file being copied is declared to be a file of <type> INTERACTIVE, thereby forcing a slight change in the program in order to produce the desired result.

```

PROGRAM JANDW;
VAR X, Y: TEXT;
    CH: CHAR;
BEGIN
  RESET(X, 'SOURCE.TEXT');
  REWRITE(Y, 'SOMETHING.TEXT');
  WHILE NOT EOF(X) DO
    BEGIN
      WHILE NOT EOLN(X) DO
        BEGIN
          READ(X, CH);
          WRITE(Y, CH);
        END;
      READLN(X);
      WRITELN(Y);
    END;
  CLOSE(Y, LOCK);
END.

```

```

PROGRAM UCSDVERSION;
VAR X, Y: INTERACTIVE;
    CH: CHAR;
BEGIN
  RESET(X, 'CONSOLE:');
  REWRITE(Y, 'SOMETHING.TEXT');
  WHILE NOT EOF(X) DO
    BEGIN
      WHILE NOT EOLN(X) DO
        BEGIN
          READ(X, CH);
          IF NOT EOLN(X) THEN
            WRITE(Y, CH);
        END;
      READLN(X);
      WRITELN(Y);
    END;
  CLOSE(Y, LOCK);
END.

```

Note that the textfiles X and Y in the above two programs had to be opened by using the U.C.S.D. extended form of the standard procedures RESET and REWRITE.

The IF statement in the interactive version of the program fragment on the left is needed in order for the file Y to become an exact copy of the textfile X. Without the IF statement, an extra blank character is appended to the end of each line of the file Y. This extra blank corresponds to the end of line character according to the standard definition in Jensen and Wirth. Note that the CLOSE intrinsic was applied to the file Y in both versions of the program in order to make it a permanent file in the disk directory called "SOMETHING.TEXT". Likewise, the textfile X could have been a diskfile instead of coming from the CONSOLE device in the right hand version of the program.

There are three predeclared textfiles which are automatically opened by the system for a user program. These files are INPUT, OUTPUT, and KEYBOARD. The file INPUT defaults to the CONSOLE device and is always defined to be of <type> INTERACTIVE. The statement READ(INPUT, CH) where CH is a character variable, will echo the character typed from the CONSOLE back to the CONSOLE device. WRITE statements to the file OUTPUT will, by default, cause the output to appear on the CONSOLE device. The file KEYBOARD is the non-echoing equivalent to INPUT. For example, the two statements

```

  READ(KEYBOARD, CH);
  WRITE(OUTPUT, CH);

```

are equivalent to the single statement READ(INPUT, CH).

For more documentation regarding the use of files see sections 2.2.6 "FILES", 2.2.4 "EOF", 2.2.5 "EOLN", 2.2.17 "WRITE AND WRITELN", and 2.2.12 "RESET". See section 2.1.2 "INPUT/OUTPUT INTRINSICS" for more details on the U.C.S.D. intrinsics.

2.2.12 RESET(F)

The standard procedure RESET, as defined on page 9 of Jensen and Wirth, resets the file window to the beginning of the file F. The next GET(F) or PUT(F) will affect record number 0 of the file. In addition, the standard definition of RESET(F) states that the window variable F^ be loaded with the first record in the file. The U.C.S.D. implementation of RESET(F) operates exactly as the standard definition, unless the file F is declared to be of <type> INTERACTIVE in which case the statement RESET(F) points the file window to the start of the file, but does not load the window variable F^. Thus, for files of <type> INTERACTIVE, the U.C.S.D. equivalent of the standard definition of RESET(F) is the two statement sequence:

```
RESET(F);  
GET(F);
```

U.C.S.D. Pascal defines an alternative form of the standard procedure RESET which is used to open a pre-existing file. In it, RESET has two parameters, the first being the file identifier; the second, either a STRING constant or variable which corresponds to the directory filename of the file being opened. See section 2.1.2 "INPUT/OUTPUT INTRINSICS" for more information on this use of RESET.

2.2.13 REWRITE(F)

The standard procedure REWRITE is used to open and create a new file. REWRITE has two parameters, the first, being the file identifier, the second corresponds to the directory filename of the file being opened, and must be either a STRING constant or variable. For example, the statement REWRITE(F, 'SOMEINFO.TEXT') causes the file F to be opened for output, and, if the file is locked onto the disk, the filename of the file in the directory will be "SOMEINFO.TEXT". REWRITE performs exactly as the U.C.S.D. OPENNEW intrinsic and will eventually replace OPENNEW. See section 2.1.2 "INPUT/OUTPUT INTRINSICS" for further documentation regarding the use of REWRITE to open a file.

2.2.14 SEGMENT PROCEDURES

The concept of the SEGMENT PROCEDURE is a U.C.S.D. extension to Pascal, the primary purpose of which is to allow a programmer the ability to explicitly partition a large program into segments, of which only a few need be resident in memory at any one time. The U.C.S.D. Pascal system is necessarily partitioned in this manner because it is too large to fit into the memory of most small interactive computers at one time.

The following is an example of the use of SEGMENT PROCEDURES:

```
PROGRAM SEGMENTDEMO;

(* GLOBAL DECLARATIONS GO HERE *)

PROCEDURE PRINT(T:STRING); FORWARD;

SEGMENT PROCEDURE ONE;
  BEGIN
    PRINT('SEGMENT NUMBER ONE');
  END;

SEGMENT PROCEDURE TWO;
SEGMENT PROCEDURE THREE;
  BEGIN
    ONE;
    PRINT('SEGMENT NUMBER THREE');
  END;
  BEGIN (* SEGMENT NUMBER TWO *)
    THREE;
    PRINT('SEGMENT NUMBER TWO');
  END;

PROCEDURE PRINT;
  BEGIN
    WRITELN(OUTPUT,T);
  END;

  BEGIN
    TWO;
    WRITELN('I'M DONE');
  END.
```

The above program will give the following output:

```
SEGMENT NUMBER ONE
SEGMENT NUMBER THREE
SEGMENT NUMBER TWO
I'M DONE
```

For further documentation on SEGMENT PROCEDURES, their use and the syntax governing their declaration see Section 3.3 "SEGMENT PROCEDURES".

2.2.15 SETS

U.C.S.D. Pascal supports all of the constructs defined for sets on pages 50-51 of Jensen and Wirth. Sets (of enumeration values) are limited to positive integers only. Space is assigned, rounding up to word boundaries, in a bitwise fashion, starting at zero, up to 4079, inclusive. Therefore a set can be at most 255 words in size, and have at most 4080 elements.

Comparisons and operations on sets are allowed only between sets which are either of the same base type or subranges of the same underlying type. For example, in the sample program below, the base type of the set S is the subrange type 0..49, while the base type of the set R is the subrange type 1..100. The underlying type of both sets is the type INTEGER, which by the above definition of compatibility, implies that the comparisons and operations on the sets S and R in the following program are legal:

```

PROGRAM SETCOMPARE;
VAR S: SET OF 0..49;
    R: SET OF 1..100;

BEGIN
  S := [0, 5, 10, 15, 20, 25, 30, 35, 40, 45];
  R := [10, 20, 30, 40, 50, 60, 70, 80, 90];
  IF S = R THEN
    WRITELN('... oops ...')
  ELSE
    WRITELN('sets work');
  S := S + R;
END.

```

In the following example, the construct $I = J$ is not legal since the two sets are of two distinct underlying types.

```

PROGRAM ILLEGALSETS;
TYPE STUFF=(ZERO, ONE, TWO);
VAR I: SET OF STUFF;
    J: SET OF 0..2;

BEGIN
  I := [ZERO];
  J := [1, 2];
  IF I = J THEN ... <<<< error here
END.

```

2.2.16 STRINGS

U.C.S.D. Pascal has an additional predeclared type STRING. Variables of type STRING are essentially PACKED ARRAYS OF CHAR that have a dynamic LENGTH attribute, the value of which is returned by the STRING intrinsic LENGTH. The default maximum LENGTH of a STRING variable is 80 characters but can be overridden in the declaration of a STRING variable by appending the desired LENGTH of the STRING variable within [] after the reserved type identifier STRING. Examples of declarations of STRING variables are:

```
TITLE: STRING; (* defaults to a maximum length of 80 characters *)
NAME: STRING[20]; (* allows the STRING to be a maximum of 20
characters*)
```

Note that a STRING variable has an absolute maximum length of 255 characters. Assignments to string variables can be performed using the assignment statement, the U. C. S. D. STRING intrinsics, or by means of a READ statement:

```
TITLE:= ' THIS IS A TITLE ';
OR
READLN(TITLE);
OR
NAME:= COPY(TITLE, 1, 20);
```

The individual characters within a STRING are indexed from 1 to the LENGTH of the STRING, for example:

```
TITLE[1]:= 'A';
TITLE[ LENGTH(TITLE) ]:= 'Z';
```

A variable of type STRING may not be indexed beyond its current dynamic LENGTH. The following sequence will result in an invalid index run time error:

```
TITLE:= '1234';
TITLE[5]:= '5';
```

A variable of type STRING may be compared to any other variable of type STRING or a string constant no matter what its current dynamic LENGTH. Unlike comparisons involving variables of other types, STRING variables may be compared to items of a different LENGTH. The resulting comparison is lexicographical. The following program is a demonstration of legal comparisons involving variables of type STRING:

```
PROGRAM COMPARESTRINGS;
VAR S: STRING;
    T: STRING[40];

BEGIN
  S:= 'SOMETHING';
  T:= 'SOMETHING BIGGER';
  IF S = T THEN
    WRITELN('Strings do not work very well')
  ELSE
    IF S > T THEN
      WRITELN(S, ' is greater than ', T)
    ELSE
```

```

        IF S < T THEN
            WRITELN(S, ' is less than ', T);
        IF S = 'SOMETHING' THEN
            WRITELN(S, ' equals ', S);
        IF S > 'SOMETHING' THEN
            WRITELN(S, ' is greater than SOMETHING');
        IF S = 'SOMETHING' THEN
            WRITELN('BLANKS DON'T COUNT');
        ELSE
            WRITELN('BLANKS APPEAR TO MAKE A DIFFERENCE');
        S := 'XXX';
        T := 'ABCDEF';
        IF S > T THEN
            WRITELN(S, ' is greater than ', T)
        ELSE
            WRITELN(S, ' is less than ', T);
    END.

```

The above program should produce the following output:

```

SOMETHING is less than SOMETHING BIGGER
SOMETHING equals SOMETHING
SOMETHING is greater than SOMETHING
BLANKS APPEAR TO MAKE A DIFFERENCE
XXX is greater than ABCDEF

```

One of the most common uses of STRING variables in the U.C.S.D. Pascal system is reading file names from the CONSOLE device:

```

PROGRAM LISTER;
VAR BUFFER: PACKED ARRAY[0..511] OF CHAR;
    FILENAME: STRING;
    F: FILE;

BEGIN
    WRITE('Enter filename of the file to be listed --->');
    READLN(FILENAME);
    RESET(F, FILENAME);
    WHILE NOT EOF(F) DO
        BEGIN
            ...
            ...
            ...
        END;
    END.

```

When a variable of type STRING is a parameter to the standard procedure READ and READLN, all characters up to the end of line character (a carriage return) in the source file will be assigned to the STRING variable. Note that care must be taken when reading STRING variables, for example, the single statement READLN(S1, S2) is equivalent to the two statement sequence READ(S1); READLN(S2). In both cases the STRING variable S2 will be assigned the empty string.

For further information concerning the predeclared type STRING see Section 2.1.1 "STRING INTRINSICS".

2.2.17 WRITE AND WRITELN

The standard procedures WRITE and WRITELN are compatible with Standard Pascal, except with respect to a WRITE or a WRITELN of a variable of type BOOLEAN. U.C.S.D. Pascal does not support the output of the words TRUE or FALSE when writing out the value of a BOOLEAN variable.

For a description of WRITE statements of variables of type STRING see Section 2.1.1 "STRING INTRINSICS".

U.C.S.D.'s WRITE and WRITELN do support the writing of entire PACKED ARRAYS OF CHAR in a single WRITE statement:

```
VAR BUFFER: PACKED ARRAY[0..10] OF CHAR;
BEGIN
  BUFFER := 'HELLO THERE'; (* contains exactly 11 characters *)
  WRITELN(OUTPUT, BUFFER);
END.
```

The above construct will work only if the ARRAY is a PACKED ARRAY OF CHAR. See section 2.2.8 PACKED VARIABLES for further information.

The following program demonstrates the effects of a field width specification within a WRITE statement for a variable of type STRING:

```
PROGRAM WRITESTRINGS;
VAR S: STRING;

BEGIN
  S := 'THE BIG BROWN FOX JUMPED...';
  WRITELN(S);
  WRITELN(S:30);
  WRITELN(S:10);
END.
```

The above program will produce the following output:

```
THE BIG BROWN FOX JUMPED...
  THE BIG BROWN FOX JUMPED..
THE BIG BR
```

Note that when a string variable is written without specifying a field width, the actual number of characters written is equal to the dynamic length of the string. If the field width specified is longer than the dynamic length of the string, leading blanks are inserted and written. If the field width is smaller than the dynamic length of the string, the excess characters will be truncated on the right.

2.2.18 IMPLEMENTATION SIZE LIMITS

The following is a list of maximum size limitations imposed upon the user by the current implementation of U.C.S.D. Pascal:

1. Maximum number of bytes of object code in a PROCEDURE or FUNCTION is 1200. Local variables in a PROCEDURE or FUNCTION can occupy a maximum of 1023 words of memory.
2. Maximum number of characters in a STRING variable is 255.
3. Maximum number of elements in a SET is $255 * 16 = 4080$.
4. Maximum number of SEGMENT PROCEDURES and SEGMENT FUNCTIONS is 16. (9 are reserved for the Pascal system, 7 are available for use by the user program)
5. Maximum number of PROCEDURES or FUNCTIONS within a segment is 127.

2.2.19 EXTENDED COMPARISONS.

U.C.S.D. Pascal allows = and <> comparisons of any array or record structure.

2.2.20 LONG INTEGERS.

UCSD Pascal allows integers of up to 36 digits. See section 3.3.3 for details regarding long integers.

2.2.21 UNITS.

UCSD Pascal now supports the modularity concept of UNITS. See section 3.3.2 for details regarding UNITS.

2.2.22 SUMMARY OF U.C.S.D. INTRINSICS

INTRINSIC	SECTION #	DESCRIPTION
BLOCKREAD	2.1.2	Function which reads a variable number of blocks from an untyped file.
BLOCKWRITE	2.1.2	Function which writes a variable number of blocks from an untyped file.

CLOSE	2.1.2	Procedure to close files.
CONCAT	2.1.1	STRING intrinsic used to concatenate strings together.
DELETE	2.1.1	STRING intrinsic used to delete characters from STRING variables.
DRAWLINE	2.1.4	Graphics intrinsic for use on the Terak 8510a.
DRAWBLOCK	2.1.4	Graphics intrinsic for use on the Terak 8510a.
EXIT	2.1.7	Intrinsic used to exit PROCEDURES cleanly.
GOTOXY	2.1.6	Procedure used for cursor addressing whose two parameters X and Y are the column and line numbers on the screen where the cursor is to be placed.
FILLCHAR	2.1.5	Fast procedure for initializing PACKED ARRAYs OF CHAR.
HALT	2.1.6	Halts a user program which may result in a call to the interactive Debugger.
IDSEARCH	---	Routine used by the Pascal compiler, and the PDP-11 assembler.
INSERT	2.1.1	STRING intrinsic used to insert characters in STRING variables.
IDRESULT	2.1.2	Function returning the result of the previous I/O operation. (See Table 2 for a list of values)
LENGTH	2.1.1	STRING intrinsic which returns the dynamic length of a STRING variable.
MARK	2.1.3	Used to mark the current top of the heap in dynamic memory allocation.
MOVELEFT	2.1.5	Low level intrinsic for moving mass amounts of bytes.
MOVERIGHT	2.1.5	Low level intrinsic for moving mass amounts of bytes.
REWRITE	2.1.2	Procedure for opening a new file.
RESET	2.1.2	Procedure for opening an existing file.
POS	2.1.1	STRING intrinsic returning the position of a pattern in a STRING variable.
PWROFTEN	2.1.6	Function which returns as a REAL result the number 10 raised to the power of the integer parameter supplied.

RELEASE	2.1.3	Intrinsic used to release memory occupied by variables dynamically allocated in the heap.
SEEK	2.1.2	Used for random accessing of records within a file.
SIZEOF	2.1.6	Function returning the number of bytes allocated to a variable.
STR	2.1.1	Procedure to convert long integer into string.
TIME	2.1.6	Function returning the time since last bootstrap of system. (returns zero if microcomputer has no real time clock)
TREESEARCH	---	Routine used solely by the Pascal compiler.
UNITBUSY	2.1.2	Low level intrinsic for determining the status of a peripheral device.
UNITCLEAR	2.1.2	Low level intrinsic to cancel I/O from a peripheral device.
UNITREAD	2.1.2	Low level intrinsic for reading from a peripheral device.
UNITWAIT	2.1.2	Low level intrinsic for waiting until a peripheral device has completed an I/O operation.
UNITWRITE	2.1.2	Low level intrinsic used for writing to a peripheral device.

* DRAWLINE AN IMPLEMENTOR'S GUIDE * * Section 3.1 *

Version I.5 September 1978

The DRAWLINE intrinsic uses an incremental technique to plot line segments on a point-addressable matrix. The algorithm guarantees a best (least squares) approximation to the desired line. In general this approximation is not unique. DRAWLINE may pick different representations for a line depending on the starting point. (This could be corrected by always starting at the same end of the line.) No range checking is performed on parameters passed to this intrinsic.

The algorithm is essentially the one described in Newman and Sproul, Principles of Interactive Computer Graphics as the Digital Differential Analyzer. It has been modified to perform only integer arithmetic. Pascal source code is included below. The procedure first determined whether the line will be more horizontal or vertical. In the discussion below, we assume the horizontal case; vertical is similar.

There will be DELTAX points plotted with horizontal increment of 1 each. The vertical increment will be $ABS(DELTA Y / DELTA X) \leq 1$. The Y coordinate arithmetic is scaled by DELTAX to eliminate fractions. An additional savings in execution time has been gained by maintaining the address of the previous point, and doing only addition and subtraction to reach the next point to be plotted.

The RADAR function is complicated as two intersecting lines may have no plotted points in common. The detection condition is either (1) the computed point is TRUE, or (2) both the next horizontal and the next vertical points are TRUE. Condition (2) could be weakened: when the line is more horizontal, only the next vertical point need be checked.

Refer to Section 2.1.4 for a description of the parameter calling sequence.

A PASCAL implementation follows:

```
PROCEDURE DRAWLINE (VAR RANGE: INTEGER; VAR SCREEN: SCREENTYPE;
ROWWIDTH, XSTART, YSTART, DELTAX, DELTAY, INK: INTEGER);
```

```
VAR X, Y, XINC, YINC, COUNT: INTEGER;
```

```
PROCEDURE DRAWDOT;
```

```
PROCEDURE RADAR;
```

```
VAR GOTIT: BOOLEAN;
```

```
BEGIN
```

```
  GOTIT := FALSE;
```

```
  COUNT := COUNT + 1;
```

```
  IF SCREEN [Y, X] THEN GOTIT := TRUE (*LANDED ON THE POINT*)
```

```
  ELSE (*WE MIGHT GO THROUGH A LINE*)
```

```
    IF SCREEN [Y+1, X] THEN
```

```
      GOTIT := SCREEN [Y, X+1];
```

```
  IF GOTIT THEN
```

```
    BEGIN
```

```
      RANGE := COUNT;
```

```
      EXIT(DRAWLINE)
```

```
    END;
```

```
  END (*RADAR*);
```

```
BEGIN (*DRAWDOT*)
```

```
  CASE INK OF
```

```
    0 (*NONE*):      EXIT (DRAWLINE); (*THEY HAD NO BUSINESS HERE*)
```

```
    1 (*WHITE*):    SCREEN [Y, X] := TRUE;
```

```
    2 (*BLACK*):    SCREEN [Y, X] := FALSE;
```

```
    3 (*REVERSE*):  SCREEN [Y, X] := NOT SCREEN [Y, X];
```

```
    4 (*RADAR*):    RADAR
```

```
  END (*CASE*)
```

```
END (*DRAWDOT*);
```

```
PROCEDURE DOFORX; (*MORE HORIZONTAL*)
```

```
VAR ERROR, I: INTEGER;
```

```
BEGIN
```

```
  IF DELTAX = 0 THEN EXIT (DRAWLINE); (*THEY'RE GOING NOWHERE*)
```

```
  ERROR := DELTAX DIV 2;
```

```
  I := DELTAX;
```

```
  REPEAT
```

```
    ERROR := ERROR + DELTAY;
```

```
    IF ERROR >= DELTAX
```

```
      THEN BEGIN ERROR := ERROR - DELTAX; Y := Y + YINC  END;
```

```
    X := X + XINC;
```

```
    DRAWDOT;
```

```
    I := I - 1;
```

```
  UNTIL I = 0;
```

```
END (*DOFORX*);
```

```

PROCEDURE DOFORY;          (*MORE VERTICAL*)
VAR ERROR, I: INTEGER;
BEGIN
  ERROR := DELTAY DIV 2;
  I := DELTAY;
  REPEAT
    ERROR := ERROR + DELTAX;
    IF ERROR >= DELTAY
      THEN BEGIN ERROR := ERROR - DELTAY; X := X + XINC  END;
    Y := Y + YINC;
    DRAWDOT;
    I := I - 1;
  UNTIL I = 0;
END (*DOFORY*);

BEGIN (*DRAWLINE*)
  X := XSTART;
  IF DELTAX < 0
    THEN BEGIN XINC := -1; DELTAX := -DELTAX  END
    ELSE XINC := 1;
  Y := YSTART;
  IF DELTAY < 0
    THEN BEGIN YINC := -1; DELTAY := -DELTAY  END
    ELSE YINC := 1;
  COUNT := 0;
  IF DELTAX >= DELTAY THEN DOFORX ELSE DOFORY;
  IF INK = 4 (*RADAR*) THEN RANGE := COUNT; (*HIT THE LIMIT GIVEN*)
END (*DRAWLINE*);

```

* FILE FORMATS * * Section 3.2 *

Version 1.5 September 1978

Text files are of the format:

<1024 bytes> header page, information for editors. This space is reserved for use by the text editors, and is respected by all portions of the system. When a user program opens a TEXT file, and REWRITES or RESETS it with a title ending in '.TEXT', the I/O subsystem will create and skip over the initial page. This is done to facilitate users editing their input and/or output data. The file-handler will transfer the header page only on a disk-disk transfer, and will omit it on a transfer to a serial device. (i.e. transfers to PRINTER:, and CONSOLE: will omit the header page)

<1024 byte pages> where a page is defined:

```
<IDLE>[indent][text][CR][IDLE][indent][text][CR]...[nulls]>
```

Data Link Escapes are followed by an indent-code, which is a byte containing the value 32+(# to indent). The nulls at the end of the page follow a [CR] in all cases, they are a pad to the end of a page. The reason for the nulls is that the compiler wants integral numbers of lines on a page. The Data Link Escape and corresponding indentation code are optional. In a given text file some lines will have the codes, and some won't.

Foto files are declared in PASCAL as follows:

```
TYPE SCREEN = PACKED ARRAY[0..239,0..319] OF BOOLEAN;  
VAR FOTOFIL: PACKED FILE OF SCREEN;
```

or something similar, which takes up the same dimensional space.

Data files are up to the user.

Code files have one block of information which describes the code kept in the file. First is an array of 16 word pairs, the first word in the pair describes the block which starts the code of the segment which is numbered as the position in the array. The second word is the number of bytes in that segment. For example if the third word in the first block of a code file is an 8, and the fourth word is 1084, you now know that segment 1 of this code file starts on block 8 of the file, and has 1084 bytes of code.

Following this array is an array of arrays of characters. The array is an array of 8 character arrays which describe the segments by name. These 8 characters are those which identify the segment at compile time. Here again, the position in this array corresponds to the segment number.

Following the array of names is an array, again 16 words long, of state descriptors. The values in this array indicate what kind of segment is at the described location. The values for this array, at present, are: LINKED, HOSTSEG, SEGPROC, UNITSEG, SEPRTSEG.

The remainder of the block, 144 words, is reserved for future use by later versions of the system. The format of the first block will most probably change completely for version II. O.

* SEGMENT PROCEDURE NOTES * * Section 3.3.1 *

Version I.5 September 1978

Declarations of SEGMENT procedures and functions are identical to standard Pascal procedures and functions except they are preceded by the reserved word 'SEGMENT', for example:

```
SEGMENT PROCEDURE INITIALIZE;  
BEGIN  
  (* PASCAL code *)  
END;
```

Program behavior differs, however, as code and data for a SEGMENT procedure (function) are in memory only while there is an active invocation of that procedure.

Advantages and benefits:

The user may now put large pieces of one-time code, eg. initialization code, into a SEGMENT procedure. After performing the initialization, for example, the now-useless code is taken out of memory thus increasing the available memory space.

Furthermore the user may now compile his/her program in chunks, specifically in SEGMENTS. The LINKER program (described in Section 1.8) can be used to link together the separate segments to produce one large code file.

Requirements and limitations:

The disk which holds the codefile for the program must be on-line (and in the same drive as when the program was started) whenever one of SEGMENT procedures is to be called. Otherwise the system will attempt to retrieve and execute whatever information now occupies that particular location on the disk, usually with very displeasing and certainly unexpected results.

A maximum of six (6) SEGMENT procedures are ordinarily available to the user.

SEGMENT procedures must be the first procedure declarations containing code-generating statements.

For further details and examples see Section 3.5, INTRODUCTION TO THE PASCAL PSEUDO MACHINE.

* LINKAGE TO EXTERNALLY COMPILED * * Section 3.3.2 *
* AND ASSEMBLED ROUTINES * * * * *

Version 1.5 September 1978

EXTERNAL COMPILATION UNITS

The UCSD Pascal 1.5 system supports a facility for integrating externally compiled and assembled routines and data structures. Use of separately compiled structures allows the user to create files of frequently used routines. After a structure is compiled, the user adds it to a library, using the library maintainer. Files that reference that structure need not compile it directly into their code file, rather, the linker copies the existing code into the host code file. Separate compilation or assembly is supported in these areas: between portions of programs written in Pascal; between assembly language routines and Pascal hosts; and finally, between assembly language routines. Each of these areas is discussed in turn by the following sections.

3.3.2.1 PASCAL TO PASCAL LINKAGES -- UNITS

A UNIT is a group of interdependent procedures, functions, and associated data structures which perform a specialized task. Whenever this task is needed within a program, the program indicates that it USES the UNIT. A UNIT consists of two parts, the INTERFACE part, which declares constants, types, variables, procedures and functions that are public and can be used by the host program, and the IMPLEMENTATION part, which declares constants, types, variables, procedures and functions that are private. These are not available to the host program and are used by the UNIT. The INTERFACE part declares how the program will communicate with the UNIT while the IMPLEMENTATION part defines how the UNIT will accomplish its task.

TURTLEGRAPHICS (example B) is a UNIT which enables the user to draw pictures using a graphics turtle. The INTERFACE consists of procedures like MOVE, TURN, and PENCOLOR, which allow the user to move the turtle and change colors. TURTLEGRAPHICS also employs DRAWLINE, an externally assembled procedure, to draw the lines and the turtle.

A program that uses TURTLEGRAPHICS has no need for DRAWLINE, and, consequently, DRAWLINE is private to that UNIT.

```

PROGRAM DRAWPOLYGON;
USES TURTLEGRAPHICS;
VAR I: INTEGER;
    SIZE, NUMSIDES: INTEGER;

BEGIN
  INITTURTLE; (* Initialize the UNIT's variables *)
  WRITE('What size polygon?');
  READLN(SIZE);
  WRITE('How many sides?');
  READLN(NUMSIDES);
  FOR I:=1 TO NUMSIDES DO
    BEGIN
      MOVE(SIZE);
      TURN(360 DIV NUMSIDES);
    END;
  END.

```

EXAMPLE A

A program must indicate the UNITS that it USES before the LABEL declaration part of the program. At the occurrence of a USES statement, the compiler references the INTERFACE part of the UNIT as though it were part of the host text itself. Therefore all public constants, types, variables, functions, and procedures are global. Name conflicts may arise if the user defines an identifier that has already been defined by the UNIT. Procedures and functions may not USE UNITS locally.

```

UNIT TURTLEGRAPHICS;
INTERFACE
  TYPE
    TGCOLOR= ( NONE, WHITE, BLACK, REVERSE );

  PROCEDURE INITTURTLE;
  PROCEDURE TURN( RELANGLE: Integer );
  PROCEDURE MOVE( RELDISTANCE: Integer );
  PROCEDURE MOVETO( X, Y: Integer );
  PROCEDURE TURNTO( ANGLE: Integer );
  PROCEDURE PENCOLOR( PCOLOR: TGCOLOR );
IMPLEMENTATION

  CONST

    TERXSIZE = 319;
    TERYSIZE = 239;
    RADCONST = 57.29578;

```

TYPE

```
SCREEN = Packed
        Array [0..TERXSIZE, 0..TERYSIZE] of Boolean;
```

VAR

```
(* Private variables *)
TGXPOS: Integer;
TGYPOS: Integer;
TGHEADING: Integer;
TGPEN: TCCOLOR;
```

```
I, J: Integer;
S: SCREEN;
```

```
(* Externally assembled procedure *)
PROCEDURE DRAWLINE( Var RADAR: Integer; Var S: SCREEN;
                   ROW, XO, YO, DX, DY, PEN: Integer );
```

```
EXTERNAL; (* External declaration *)
```

PROCEDURE INITTURTLE;

```
BEGIN
  Fillchar( SCREEN, Sizeof(SCREEN), 0 );
  Unitwrite( 3, SCREEN, 63 );
  HEADING := 0;
  TGXPOS := 0;
  TGYPOS := 0;
END;
```

PROCEDURE MOVE; (* Public procedure, parameters declared above *)

```
BEGIN
  MOVETO( Round(TURTX + DIST*Cos(TURTLEANGLE/RADCONST),
              Round(TURTY + DIST*Sin(TURTLEANGLE/RADCONST) ));
END;
```

PROCEDURE MOVETO;

```
VAR R: Integer;
BEGIN
  DRAWLINE( R, S, 20, 160+TURTX, 120-TURTY,
            X-TURTX, TURTY-Y, ORD(TURTLEPEN) );
END;
```

PROCEDURE TURN; (* Public procedure, parameters declared above *)

```
BEGIN
  HEADING := ( HEADING+RELANGLE ) mod 360;
END;
```

```

PROCEDURE TURNT0;
BEGIN
  HEADING := ANGLE;
END;

PROCEDURE PENCOLOR;
BEGIN
  TGPEN := PCOLOR;
END;

END. (* End of unit *)

```

EXAMPLE B

Example B is a skeleton for a TURTLEGRAPHICS UNIT. Note that the procedures MOVE, TURN, and INITTURTLE, and the TYPE TGCOLOR, are declared in the INTERFACE part and are available for use by the host program. Since the procedure DRAWLINE is not part of the INTERFACE, it is private, and may not be used by the host. The syntax for a UNIT definition is shown below. The declarations of routine headings in the INTERFACE part are similar to forward declarations; therefore, when the corresponding bodies are defined in the IMPLEMENTATION part, formal parameter specifications are not repeated.

A UNIT may also USE another UNIT, in which case the USES declaration must appear at the beginning of the INTERFACE part. In example C, PICTUREGRAPHICS indicates in the INTERFACE part that it USES TURTLEGRAPHICS. Note that the program USEGRAPHICS, which USES PICTUREGRAPHICS, indicates that it USES TURTLEGRAPHICS before using PICTUREGRAPHICS. It is important that the INTERFACE part of TURTLEGRAPHICS be defined before PICTUREGRAPHICS makes references to it, therefore this ordering is required.

NOTE: Variables of type FILE must be declared in the INTERFACE part of a UNIT. A FILE declared in the IMPLEMENTATION part will cause a syntax error upon compilation.

```

UNIT PICTUREGRAPHICS;
INTERFACE
  USES TURTLEGRAPHICS;      (* TURTLEGRAPHICS is defined in the *)
  TYPE                      (* *system.library see section III below *)
    PVECTOR=^VECTOR;
    VECTOR=RECORD
      DELHEADING: INTEGER;
      DELDISTANCE: INTEGER;
      PENDOWN: BOOLEAN;
      NEXTVEC: PVECTOR
    END; (* record *)

```

```

VAR
  START:PVECTOR; (* Head of list of lines *)
  HEAP:^INTEGER;

PROCEDURE MAKESUBPICTURE;

PROCEDURE DRAWSUBPICTURE;

IMPLEMENTATION

PROCEDURE MAKESUBPICTURE;
BEGIN
  (* Calculates next subpicture and stores on heap *)
END;

PROCEDURE DRAWSUBPICTURE;
BEGIN
  LPVEC:=START;          (* Start at beginning of list *)
  WHILE LPVEC<>NIL DO (* and draw each that's there *)
    WITH LPVEC^ DO
      BEGIN
        TURN(DELHEADING);
        MOVE(DELDISTANCE);
        IF PENDOWN THEN TOPEN:=WHITE
        ELSE TOPEN:=NONE;
        LPVEC:=NEXTVEC;
      END;
    END; (* drawsubpicture *)
  END;
END;

```

```

PROGRAM USEGRAPHICS;
USES TURTLEGRAPHICS, PICTUREGRAPHICS;
BEGIN
  INITTURTLE;
  REPEAT
    MARK(HEAP);
    MAKESUBPICTURE;
    DRAWSUBPICTURE;
    RELEASE(HEAP);
  UNTIL START=NIL;
END.

```

```

(* picturegraphics uses *)
(* turtlegraphics      *)

```

EXAMPLE C

```

< Compilation unit > ::= < Program heading > ; < Unit definition > ;
                       < Uses part > < Block > ;
                       < Unit definition > ; < Unit definition > .

< Unit definition > ::= < unit heading > ;
                       < Interface part >
                       < Implementation part >
                       End

< Unit heading > ::= Unit < Unit identifier > !
                  Separate unit < Unit identifier >

< Unit identifier > ::= < Identifier >

< Interface part > ::= Interface
                   < Uses part >
                   < Constant definition part >
                   < Type definition part >
                   < Variable declaration part >
                   < Procedure heading > ! < Function heading >

< Implementation part > ::= Implementation
                        < Label declaration part >
                        < Constant definition part >
                        < Type definition part >
                        < Variable declaration part >
                        < Procedure and Function declaration part >

< Uses part > ::= Uses < Unit identifier >
               , < Unit identifier > ; ! < Empty >

```

DIAGRAM D

The user may define a UNIT in-line, after the heading of the host program. In this case the user compiles both the UNIT, and the host program together. Any subsequent changes in the UNIT or host program require the user to recompile both. The user may also define and compile a UNIT (or a group of UNITS) separately, and use the library manager to store it (or them) in a library. After compiling a host program that uses such a UNIT, the user must link that UNIT into the code file by executing the LINKER. Trying to R(un an unlinked code file will cause the LINKER to run automatically, trying to X(ecute an unlinked file causes the system to remind you to link the file . Changes in a host program require only that the user recompile the program and link in the UNIT. Changes in the IMPLEMENTATION part of a UNIT only require the user to compile the UNIT, and then to relink all compilation units that use that UNIT. Changes in the INTERFACE part of a UNIT require that the user recompile both the UNIT and all compilation units that use that UNIT. In this case all these compilation units must again be linked. For more information see section 1.B LINKER or section 4.2 LIBRARIAN.

The compiler generates LINKER information in the contiguous blocks that follow a program that uses UNITS. This information contains locations of references to externally defined identifiers. The LINKER document explains the format of this information.

3.3.3.2 PASCAL TO ASSEMBLY LANGUAGE LINKAGES -- EXTERNAL PROCEDURES

External procedures are primarily separately assembled assembly language procedures, stored in a LIBRARY on disk. Most programs that require external procedures must have them linked into the compiled code file. Typically the user writes external procedures in assembly language, to handle low-level operations that Pascal is not designed to provide. External assembly language procedures are also used for their comparative speed in 'real time' applications.

A host program declares that a procedure is external in much the same way as a procedure is declared FORWARD. A standard heading is provided, followed by the keyword EXTERNAL. Calls to the external procedure use standard Pascal syntax, and the compiler checks that calls to the external agree in type and number of parameters with the external declaration. It is the user's responsibility to assure that the assembly language procedure respects the Pascal external declaration. The linker checks only that the number of words of parameters agree between the Pascal and assembly language declarations. For more information see section 1.8 Linker and 1.9 Assembler(s).

The conventions of the surrounding system concerning register use and calling sequences must be restricted by writers of assembly language routines. These conventions for the PDP-11 and Z80/8080 implementations are given here.

First, for the PDP-11, registers R0 and R1 are available for use; any others affected by a routine must be saved on entry and restored on exit. The following call and return sequence is recommended for procedures. It has the advantage that calls can be made directly from assembly language as well as from Pascal.

```
.PROC ENTRY, 2

PARAM1 .EGU    6      ;Offset for first parameter
PARAM2 .EGU    4      ;Offset for second paramter
RETADDR .EGU    2      ;Offset for return address
OLDR5   .EGU    0      ;Offset for original value of R5
LOCAL1  .EGU   -2      ;Offset for first local
LOCAL2  .EGU   -4      ;Offset for second local

MOV     R5, -(SP)      ;Save contents of R5
MOV     SP, R5         ;Use R5 to get at locals and parameters
CLR     -(SP)         ;Reserve and Initialize
CLR     -(SP)         ;Two local variables
.
.
.
;Inside routine
MOV     PARAM(R5), LOCAL1(R5) ;Sample statement
```

```

EXIT:  MOV    R5, SP           ; Cut back to entry SP
      MOV    (SP)+, R5       ; Restore previous R5
      MOV    (SP)+, R0       ; Get return address
      ADD    #NPARAMS, SP    ; Discard parameters
      JMP    @R0             ; Return to caller

```

In Z80 assembly language routines, all registers are available for use, and the recommended interface sequence follows: (This code would work for both 8080's and Z80's. Optimizations are possible if the Z80 instructions are available.)

```

      .PROC  ENTRY, 2
      .PRIVATE  RETADDR, LOCAL1, LOCAL2, PARAM1, PARAM2
      ; Reserve static storage for this routine. Much easier to
      ; reference objects like this rather than relative to
      ; register as on PDP-11
      POP    HL               ; Get return address
      LD     (RETADDR), HL    ; and save it
      POP    HL               ; Get and save PARAM2
      LD     (PARAM2), HL
      POP    HL               ; Get and save PARAM1
      LD     (PARAM1), HL
      .
      .
      LD     HL, (PARAM2)     ; Move PARAM2
      LD     (LOCAL1), HL    ; to LOCAL1
      .
      .
EXIT:  LD     HL, (RETADDR)   ; Get return address
      JP     (HL)
      .END

```

For assembly language functions (.FUNC's) the sequence is essentially the same, except that:

1) Two words of zeros are pushed by the compiler before any parameters are put on the stack.

2) After the stack has been completely cleaned up at the routine exit time, the .FUNC must push the function result on the stack.

Here is an example of an external assembly language procedure, and a program that uses it. This example takes a very primitive approach to interrupt handling (which might still be useful in some applications). There is no provision for handling interrupts from the device where a collected buffer is being written to disk. Support for continuous interrupts would be more complex, involving multiple buffers and exclusion mechanisms to assure that buffer switching would occur reliably. The Project intends eventually to provide synchronization

capabilities at the Pascal level, so that interrupt handling can be accomplished with greater convenience and safety.

```

.PROC          DRCOLLECT,0      ; Name of routine for use by linker.
               .CONST         DRBUFLENG      ; Public constant.
               .PUBLIC        DRBUFFER      ; Public variable.

DRADDR        .EGU            167770
DRVECT        .EGU            140
MOV           #HANDLR,@#DRVECT ; Load address of interrupt
MOV           #340,@#DRVECT+2  ; handler and set priority.
MOV           #DRBUFLENG,RO    ; Load RO with size of buffer.
MOV           #DRBUFFER,R1     ; Load R1 with address of buffer.
BIS          #100,@#DRADDR    ; Enable interrupts on DR interface.

LOOP:         TST             RO            ; Exit loop when buffer full.
               BNE            LOOP
               BIC            #100,@#DRADDR ; Disable interrupts.
               RTS            PC          ; Return to PASCAL host program.

HANDLR:       MOV            @#DRADDR+2,(R1)+ ; Load buffer with next word,
               DEC            RO            ; increment R1, decrement RO.
               RTI            ; Return from interrupt.

```

```

PROGRAM COLLECTDATA;
CONST
  DRBUFLENG = 256;

TYPE
  DATABUFFER = Array [1..DRBUFLENG] of integer;

VAR
  I: Integer;
  DRBUFFER: DATABUFFER;
  DATAFILE: File of DATABUFFER;

PROCEDURE DRCOLLECT;
  External;

BEGIN (*of Collect Data*)
  Rewrite( DATAFILE, 'SAMPLE.DATA' );
  For I:=1 to 10 do
    BEGIN
      DRCOLLECT;
      DATAFILE^:=DRBUFFER;
      Put( DATAFILE );
    END;
  Close( DATAFILE, Lock );
END.

```

3.3.2.3 ASSEMBLY LANGUAGE TO ASSEMBLY LANGUAGE LINKAGES

The third way in which separate routines may share data structures and subroutines is by linkage from assembly language to assembly language. This is made possible through the use of the .DEF and .REF pseudo-ops provided in the UCSD assemblers. These generate link information that allows two separately assembled procedures to be Linked together. One possible use for this will be the linking of separate routines and drivers in constructing new UCSD interpreters.

The following are very abbreviated versions of two assembly language routines which make separate references. They are used externally by the UNIT PSGRAPHICS:

The first routine declares three public variables and declares a .DEF for a label to be referenced by the second routine (Note that this is only a skeleton of the actual MOVETO routine):

```
.PROC  MOVETO,6  ; THE 3 REAL PARAMETERS OCCUPY 6 WORDS

;  PROCEDURE MOVETO(X, Y, Z: REAL);
;
;  COMPUTES A NEW PSXPOS & PSYPOS FROM PSMATP AND
;  AN ASSUMED 1.0 AS THE INPUT VECTOR HOMOGENOUS
;  COORDINATE...
;
;  (X Y Z 1) dot PSMATP^ = (X' Y' Z' W')
;  PSXPOS := X'/W';
;  PSYPOS := Y'/W';

; THESE ARE GLOBALS IN THE PASCAL HOST
.PUBLIC PSXPOS
.PUBLIC PSYPOS
.PUBLIC PSMATP

; MOVETO ENTRY POINT

      MOV     R5, -(SP)      ; R5 USED AS FRAME POINTER
      MOV     SP, R5
      MOV     @#PSMATP, R0  ; R0 IS TOS MATRIX POINTER

; PARAMETER DISPLACEMENTS FROM R5 FRAME POINTER
X     .EQU    14
Y     .EQU    10
Z     .EQU     4
W     .EQU   -4
;
;  COMPUTE W', HOMOGENEOUS COORD
;  AND LEAVE IT ON STACK
;
;
```

```

; COMPUTE PSXPOS
;
; NOW COMPUTE PSYPOS
;
;
; CLEAN UP STACK AND RETURN
;
ROUND: ; ROUND REAL ON STACK TO INTEGER
; IF < 0 THEN SUBTRACT 0.5 ELSE
; ADD 0.5, THEN TRUCATE.

.END

```

The second routine references the first routine as well as the separately assembled DRAWLINE routine. MOVETO must be linked into LINETO before the routine can be linked in as an external procedure to a PASCAL UNIT or PROGRAM.

```

.PROC LINETO,6 ; PARAMETERS OCCUPY 6 WORDS
;
; PROCEDURE LINETO(X, Y, Z: REAL);
;
; DRAWS A LINE FROM THE LAST POINT CONTAINED IN
; PSXPOS & PSYPOS TO THE NEW TRANSFORMED POINT
; GIVEN BY X, Y, & Z...
;
; SAVEX := PSXPOS; SAVEY := PSYPOS;
; MOVETO(X, Y, Z);
; DRAWLINE(JUNK, PSBUFF^, 20, 160+SAVEX, 120-SAVEY,
; PSXPOS-SAVEX, SAVEY-PSYPOS, 1);
;
.PUBLIC PSXPOS
.PUBLIC PSYPOS
.PUBLIC PSBUFF
.PRIVATE RANGE

.REF MOVETO
.REF DRAWLINE

; LINETO ENTRY POINT
;
; MOV R5, -(SP)
; MOV SP, R5 ; USE R5 AS STACK FRAME POINTER
SAVEX .EQU -2
SAVEY .EQU -4
X .EQU 14
Y .EQU 10
Z .EQU 4
;
; SAVEX := PSXPOS; SAVEY := PSYPOS;
;
; MOVETO(X, Y, Z);

```

```
;  
JSR    PC,@#MOVETO  
;  
; DRAWLINE(...);  
;  
JSR    PC,@#DRAWLINE  
;  
; ALL DONE... RETURN  
;  
JMP    @RO  
.END
```

For examples and more information see section 1.9 ASSEM

* LONG INTEGERS * * SECTION 3.3.3 *

Version I.5 September 1978

A new addition to U. C. S. D. Pascal predeclared type INTEGER is the optional use of a length attribute (available only on LSI 11/PDP 11 based micros). This essentially constitutes a new type and will, in the remainder of this document, be referred to as LONG INTEGER. The LONG INTEGER is suitable for business, scientific or other applications in which the need for extended number length with complete accuracy is felt. This extension supports the four basic standard INTEGER arithmetic operations (addition, subtraction, division and multiplication) as well as routines facilitating conversion to strings and standard INTEGERS. Strong type checking is enforced throughout to reduce potential errors. Input/Output, in line declaration of constants and inclusion in structured types are all fully supported and are analogous to the usage of standard INTEGERS.

LONG INTEGERS are declared using the standard identifier INTEGER followed by a length attribute in square brackets. This length is an unsigned number, not larger than 36, denoting the minimum number of decimal digits representable by the LONG INTEGER. For example, a variable called 'X' capable of storing at least an eight decimal digit signed number would be created by:

```
VAR X: INTEGER[8];
```

Constants are defined in the normal manner:

```
CONST RYDBERG = 10973731;
```

In the above example RYDBERG would be by default a LONG INTEGER and could be used anywhere a LONG INTEGER could be used.

In general LONG INTEGERS may be used anywhere it is syntactically correct to use REALS (not fully implemented until II.D; for now LONG INTEGERS are limited to arithmetic operations, assignment statements (but not assignment to a REAL), TRUNC, and STR); however care must be taken to ensure that sufficient words have been allocated by the declared length attribute for storage of the result of assignment or arithmetic expression statements (see note in next subsection for complete details). INTEGER expressions are implicitly converted as required upon assignment to, or arithmetic operations with, a LONG INTEGER. The reverse is not true. Unary plus/minus is correctly handled. Examples:

VAR I: INTEGER;
L: INTEGER[N], where N is an acceptable length
S: REAL;

I = L: compile time error, see TRUNC(L) below
L = -L: correct, with the usual exception
L = I: always correct
L = S: never accepted
S = L: will be implemented with II.0

Arithmetic operations which may be used in conjunction with LONG INTEGERS are any or all from the set +, -, *, DIV, unary plus/minus. On assignment the length of the LONG INTEGER is adjusted (during execution) to the declared length attribute of the variable, therefore an interrupt (overflow) may result. An interrupt (overflow) occurs only when the intermediate result exceeds the number of words required to store (as a minimum) thirty-seven decimal digits, or when the final result is assigned to a variable with insufficient length attribute. (On the matter of the length attribute and what it defines: a length attribute of 5 thru 9 may store up to and including 2147483647, length attributes of 10 thru 14 may store thru 140737488355327, 15 thru 18 .. 9223372036854775807. It is left to the interested reader to compute any larger length attribute storage capacities. Thus it would be unwise to attempt to use a LONG INTEGER as a subrange. This range of length attributes all having the same upper bound is a result of the allocation of a full word as the least amount of additional storage, i. e. 5 thru 9 represent a two word INTEGER.) All of the standard relational operators may be used with mixed LONG INTEGER and INTEGER.

The function TRUNC(L), where 'L' is a LONG INTEGER, will convert 'L' to an INTEGER (i. e. TRUNC will accept a LONG INTEGER as well as a REAL as an argument). Interrupt (overflow) will result if L is greater than MAXINT.

The procedure STR(L,S) converts the INTEGER or LONG INTEGER 'L', into a string (complete with minus sign if needed) and places it in the STRING 'S'. The following program segment will provide a suitable dollar and cent routine:

```
STR(L,S); INSERT('.',S,LENGTH(S)-1); WRITELN(S);
```

Where 'L' and 'S' are appropriately declared. TRUNC and STR are the only two routines which currently will accept LONG INTEGERS as parameters. An attempt to declare a LONG INTEGER in a parameter list will result in a compile time error, which may be circumvented by creating a type which is a LONG INTEGER. For example:


```
TYPE LONG = INTEGER[18];  
PROCEDURE BIGNUMBER(BANKACCT: LONG);
```

The LONG INTEGER is stored as a multi-word, twos complement binary number. System and interpreter routines do the I/O conversions as required. Maximum storage efficiency is achieved by dynamic expansion and contraction of word allocation as required. During LONG INTEGER operations the length is placed on the stack above the number itself, the declared length attribute need not be the same and can be less than this length.

PSEUDO

* PSEUDO-MACHINE ARCHITECTURE * * Section 3.4 *

Version I.5 September 1978

The UCSD Pascal P-machine, designed specifically for the execution of Pascal programs on small machines, is an extensively modified descendant of the P-2 pseudo-machine from Zurich. It supports variable addressing, including strings, byte arrays, packed fields, and dynamic variables; logical, integer, real, and set top-of-stack arithmetic and comparisons; multi-element structure comparisons; several types of branches; procedure/function calls and returns, including overlayable procedures; miscellaneous procedures used by systems programs; and an I/O system.

This Section, to be used in conjunction with Section 3.5, describes the P-machine "hardware," communication with the operating system, exceptional condition handling, the instruction set, the I/O system, and the bootloading process.

NOTE: not all of the above will be included in the I.5 release and will only be available sometime later.

3.4.1 HARDWARE

There exists no physical P-machine (yet!). The P-machine exists only as interpreters written in assembly languages of actual computers. However, this can and will be ignored in the following description.

The P-machine uses 16-bit words, with two 8-bit bytes per word. It has several registers and a user memory, in which are kept a stack and a heap. All registers are pointers to word-aligned structures, except IPC, which is a pointer to byte-aligned instructions. The registers are:

SP: Stack Pointer is a pointer to the top of the execution stack. The stack starts in high memory and grows toward low memory. It contains code segments and activation records, and is used to pass parameters, return function values, and as an operand source for many instructions. The stack is extended by loads and procedure calls, and is cut back by stores, procedure returns, and arithmetic operations.

NP: New Pointer is a pointer to the top of the dynamic heap. The heap starts in low memory and grows upward toward the stack. It contains all dynamic variables (see Jensen and Wirth, Chapter 10). It is extended by the standard procedure 'new', and is cut back by the standard procedure 'release'.

- JTAB:** Jump TABLE pointer is a pointer to the procedure attribute table of the currently executing procedure. (See Section 3.5, figure 5.)
- SEQ:** Segment Pointer points to the procedure dictionary of the segment to which the currently executing procedure belongs. (See Section 3.5, figure 6.)
- MP:** Most recent Procedure is a pointer to the activation record of the currently executing procedure. (See Section 3.5, figure 7.) Variables local to the current procedure are accessed by indexing off MP.
- BASE:** BASE Procedure is a pointer to the activation record of the most recently invoked base procedure (lex level 0). Global (lex level 0) variables are accessed by indexing off BASE.

3.4.2 OPERATING SYSTEM/P-MACHINE COMMUNICATION - SYSCOM

It is sometimes necessary for the operating system and the P-machine to exchange information. Hence there exists a variable SYSCOM in the outer block of the operating system, and a corresponding area in memory known to the hardware. The fields in SYSCOM actually relevant to this communication are:

- IORSLT:** contains the error code returned by the last activated or terminated I/O operations. (See I/O section below, and operating system read and write procedures.)
- XEQERR:** contains the error code of the last run-time error. (See exception handling below.)
- SYSUNIT:** contains the unit number of the device the operating system was booted from (usually 4 or 5).
- BUGSTATE:** contains the current bugstate. (See BPT instruction below.)
- QDIRP:** contains a pointer to the most recent disk directory read in, unless dynamic allocation or deallocation has taken place since then. (See MRK, RLS, and NEW instructions below.)
- STKBASE, LASTMP, SEQ, JTAB:** copies of the BASE, MP, SEQ and JTAB registers.
- BOMBP:** contains a pointer to the activation record of the operating system routine EXECERROR when a runtime error occurs. (See exception handling.)

BOMIPC: contains the value of IPC when a run-time error occurs.

HLTLIN: contains the line number of the last conditional halt executed. (See BPT instruction.)

BRKPTS: contains up to four line numbers of breakpointed statements. (See BPT instruction.)

CRTINFO.EOF: contains the end-of-file character (see console input driver).

CRTINFO.FLUSH: contains the flush-output character (see console input, output drivers).

CRTINFO.STOP: contains the stop-output character (see console output and input drivers).

CRTINFO.BREAK: contains the break-execution character (see console input driver).

SEGTABLE: contains the segment dictionary for the pascal system.

3.4.3 EXCEPTION HANDLING - XEGERR

Whenever a run-time error occurs, the P-machine stops executing the current instruction (ideally leaving the evaluation stack in as nice a condition as possible) and transfers control to the XEGERR routine. This routine

- 1) enters the error code into SYSCOM^.XEGERR.
- 2) calculates what MP will be after step 4, and sets SYSCOM^.BOMBP to that. (The size of EXECERROR's activation record must be known by the P-machine.)
- 3) stores the current value of IPC into SYSCOM^.BOMIPC.
- 4) points IPC to a CXP 0,2 (call operating system procedure EXECERROR) instruction.
- 5) resumes execution of interpreter code, starting with the CXP.

3.4.4 OPERAND FORMATS

Although an element of a structure may occupy as little as one bit, as in a PACKED ARRAY OF boolean, variables in the P-machine are always aligned on word boundaries. All top-of-stack operations expect their operands to occupy at least one word, even if not all the information in a word is valid. The least significant bit of a word is bit 0, the most significant is bit 15.

BOOLEAN: One word. Bit 0 indicates the value (false=0, true=1), and this is the only information used by boolean comparisons. However, the boolean operators LAND, LOR, and LNOT operate on all 16 bits.

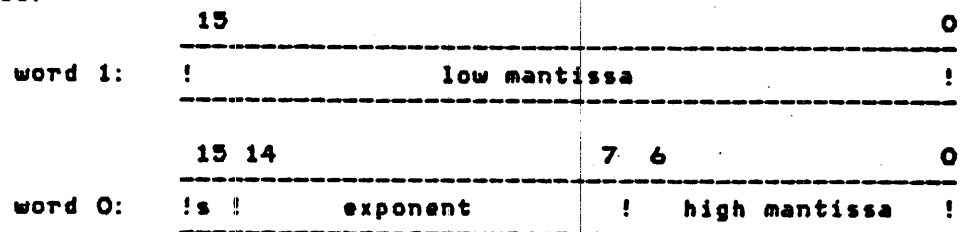
INTEGER: One word, two's complement, capable of representing values in the range -32768..32767.

SCALAR (user-defined): One word, in range 0..32767.

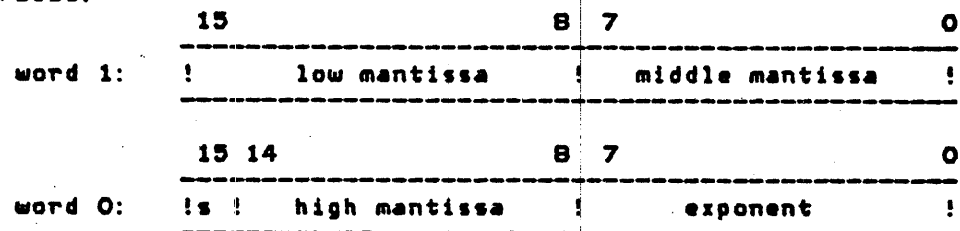
CHAR: One word, with low byte containing character. The internal character set is "extended" ASCII, with 0..127 representing the standard ASCII set, and 128..255 as a user-defined character set.

REAL: Two words, with format implementation dependent. The system is arranged so that only the interpreter needs to know the detailed internal format of REALs (beyond the fact that they occupy two words) Following are the two detailed formats for the CPUs we now (as of I.4) support.

PDP11:



Z80/8080:



Both representations have an excess-128 exponent, a fractional mantissa that is always normalized, exponent base 2, an implicit 24th mantissa bit, and zero represented by a zero exponent. (See PDP11 processor manual or Z80/8080 interpreter listing for greater detail.)

POINTER: One or three words, depending on type of pointer.
 Pascal pointers, internal word pointers: one word, containing a word address.

Internal byte pointers: one word, containing a byte address.

Internal packed field pointers: three words.

word 2: word pointer to word field is in.

word 1: field_width (in bits).

word 0: right_bit_number of field.

SET: 0..255 words in data segment, 1..256 words on stack. Sets are

implemented as bit vectors, always with a lower index of zero. A set variable declared as set of m..n is allocated $(n+1) \div 16$ words. When a set is in the data segment, all words allocated contain valid information.

When a set is on the stack, it is represented by a word containing the length, and then that number of words, all of which contain valid information. All elements past the last word of a set are assumed not to be elements of the set. Before being stored back in the data segment, a set must be forced back to the size allocated to it, and so an ADJ instruction must be issued.

RECORDS and ARRAYS: any number of words (up to 16384 words in one dimension). Arrays are stored in row-major order, and always have a lower index of zero. Only fields or elements are loaded onto the stack - never the structure itself. Packed arrays must have an integral number of elements in each word, as there is no packing across word boundaries (it is acceptable to have unused bits in each word). The first element in each word has bit 0 as its low-order bit.

STRINGS: 1..128 words. Strings are a flexible version of packed arrays of char. A string[n] occupies $(n \div 2)+1$ words. Byte 0 of a string is the current length of the string, and bytes 1..length(string) contain valid characters.

CONSTANTS: constant scalars, sets, and strings may be imbedded in the instruction stream, in which case they have special formats. All scalars (excluding reals) not in the range 0..127: two bytes, low byte first.

Strings: all string literals take length(literal)+1 bytes, and are byte aligned. The first byte is the length, the rest are the actual characters. This format applies even if the literal should be interpreted as a packed array of char (see S1P and S2P below).

Reals and sets: word aligned, and in reverse word order.

3.4.5 INSTRUCTION SET FORMAT

Instructions on the P-machine are one or two bytes long, followed by zero to four parameters. Most parameters specify one word of information, and are one of five basic types.

- UB** unsigned byte: high order byte of parameter is implicitly zero.
- SB** signed byte: high order byte is sign extension of bit 7.
- DB** don't care byte: can be treated as SB or UB, as value is always in the range 0..127.
- B** big: this parameter is one byte long when used to represent values in the range 0..127, and is two bytes long when representing values in the range 128..32767. If the first byte is in 0..127, the high byte of the parameter is implicitly zero. Otherwise, bit 7 of the first byte is cleared and it is used as the high order byte of the parameter. The second byte is used

as the low order byte.
W word: the next two bytes, low byte first, is the parameter value.

Any exceptions to these formats are noted in the instructions where they occur.

3.4.6 ENGLISH INSTRUCTION SET DESCRIPTION

In the following section, references to an element on the stack are context-dependent, and can mean anywhere from one word to 256 words. Also, unless specifically noted to the contrary, operands are popped off the stack - they are not left around.

Abbreviations are used widely, but use fairly simple conventions. Parameters are written as X or X_n, where X is UB, SB, DB, B, or W, and n is an integer indicating the parameter position in the instruction. Tos means the operand on the top of stack, tos-1 the next operand, etc. Mark Stack Control Word is abbreviated to MSCW.

Many instructions refer to the activation record of a procedure, and this document assumes the reader has a general knowledge of procedure calling in stack machines, and the concept of stack frames. An activation record as defined in this document specifically consists of:

- 1) the local data segment of the procedure, and
- 2) the MSCW, containing addressing information (static links), and information on the calling procedures environment when the procedure was called.

(See Section 3.5, figure 7.)

The dynamic chain refers to the calling chain, traversed using the MSCW.MSDYN links. The static chain refers to the lexical or ancestor chain, traversed using the MSCW.MSSTAT links.

MNEMONIC	OP-CODE	PARAMETERS	FULL NAME AND OPERATION
----------	---------	------------	-------------------------

5.A VARIABLE FETCHING, INDEXING, STORING, AND TRANSFERING

5.A.1 ONE WORD LOADS AND STORES

5.A.1.a CONSTANT ONE WORD LOADS

SLDC	0..127		Short load word constant. Pushes the opcode, with high byte zero, onto stack.
------	--------	--	---

LDCN	159		Load constant <u>nil</u> . Pushes the implementation-dependent value of <u>nil</u> .
LDCI	199	W	Load constant word. Pushes W.
5. A. 1. b LOCAL ONE WORD LOADS AND STORE			
SLDL1	216		Short load local word. SLDLx fetches the word with offset x in MP activation record and pushes it.
..	..		
SLDL16	231		
LDL	202	B	Load local word. Fetches the word with offset B in MP activation record and pushes it.
LLA	198	B	Load local address. Fetches address of the word with offset B in MP activation record and pushes it.
STL	204	B	Store local word. Stores tos into word with offset B in MP activation record.
5. A. 1. c GLOBAL ONE WORD LOADS AND STORE			
SLDO1	232		Short load global word. SLDOx fetches the word with offset x in BASE activation record and pushes it.
..	..		
SLDO16	247		
LDO	167	B	Load global word. Fetches the word with offset B in BASE activation record and pushes it.
LAO	165	B	Load global address. Pushes the word address of the word with offset B in BASE activation record.
SRO	171	B	Store global word. Stores tos into the word with offset B in BASE activation record.
5. A. 1. d INTERMEDIATE ONE-WORD LOADS AND STORE			
LOD	182	DB, D	Load intermediate word. DB indicates the number of static links to traverse to find the activation record to use. B is the offset within the activation record.
LDA	178	DB, B	Load intermediate address.
STR	184	DB, B	Store intermediate word.

5. A. 1. e INDIRECT ONE-WORD LOADS AND STORE

STO 154 Store indirect. Tos is stored into the word pointed to by tos-1.

SINDO 248 Load indirect.

5. A. 2 MULTIPLE WORD LOADS AND STORES (SETS AND REALS)

LDC 179 UB, <block> Load multiple word constant. UB is the number of words to load, and <block> is a word aligned block of UB words, in reverse word order. Load the block onto the stack.

LDM 188 UB Load multiple words. Tos is a pointer to the beginning of a block of UB words. Push the block onto the stack.

STM 189 UB Store multiple words. Tos is a block of UB words, tos-1 is a word pointer to a similar block. Transfer the block from the stack to the destination block.

5. A. 3 BYTE ARRAYS

BYT 210 Byte conversion. Convert word pointer tos to a byte pointer. (NOP on the PDP11 and Z80/BOEO implementations.)

LDB 190 Load byte. Push the byte (after zeroing high byte) pointed to by byte pointer tos.

STB 191 Store byte. Store byte tos into the location specified by byte pointer tos-1.

MVB 169 B Move bytes. Tos is a byte source pointer to a block of B bytes, tos-1 is a byte destination pointer to a similar block. Transfer the source block to the destination block. (This instruction is redundant due to word alignment, and will be replaced by MOV in the future.)

IXB 209 Index byte array. Push a byte pointer formed from the integer index tos and the byte pointer tos-1.

5. A. 4 STRINGS

LCA	166	UB, <chars>	Load constant string address. Push a byte pointer to the location UB is contained in, and skip IPC past <chars>.
SAS	170	UB	String assign. Tos is either a source byte pointer or a character. (Characters always have a high byte of zero, while pointer never do.) Tos-1 is a destination byte pointer. UB is the declared size of the destination string. If the declared size is less than the current size of the source string, a run-time error occurs; otherwise all bytes of source containing valid information are transferred to the destination string.
S1P	208		String to packed conversion on tos. Tos is a byte pointer to a string, and is incremented by one byte in order to point to the first character of the string.
S2P	157		String to packed conversion on tos-1. Tos and tos-1 are byte pointers, and tos-1 is incremented by one byte.
IXS	155		Index string array. Performs the same operation as IXB, except before indexing the index is checked to see if it is in the range 1..current length. If not, a run-time error occurs.

5. A. 5 RECORD AND ARRAY INDEXING AND ASSIGNMENT

MOV	168	B	Move words. Tos is a source pointer to a block of B words, tos-1 is a destination pointer to a similar block. Transfer the block from the source to the destination.
SIND0	248		Short index and load word. SINDx indexes the word pointer tos by x words, and pushes the word pointed to by the result.
SIND7	255		
IND	163	B	Static index and load word. Indexes the word pointer tos by B words, and pushes the word pointed to.
INC	162	B	Increment field pointer. The word pointer tos is indexed by B words and the resultant pointer is pushed.
IXA	164	B	Index array. Tos is an integer index, tos-1 is the array base word pointer, and B is the size (in words) of an array element. A word pointer to the indexed element is pushed.

IXP	192	UB_1, UB_2	Index packed array. Tos is an integer index, tos-1 is the array base word pointer. DB_1 is the number of element_per_word, and DB_2 is the field_width (in bits). Compute and push a packed field pointer.
LDP	186		Load a packed field. Push the field described by the packed field pointer tos.
STP	187		Store into a packed field. Tos is the data, tos-1 is a packed field pointer. Store tos into the field described by tos-1.

5. A. 6 DYNAMIC VARIABLE ALLOCATION AND DE-ALLOCATION

NEW	158	1	New variable allocation. Tos is the size (in words) to allocate the variable, and tos-2 is a word pointer to a dynamic variable. If GDIRP is non- <u>nil</u> , cut NP back to GDIRP and set GDIRP to <u>nil</u> . Store NP into word pointed to by tos-1, and increment NP by tos words.
MRK	158	31	Mark heap. Release GDIRP and set to <u>nil</u> if necessary, then store NP into word pointed to by tos.
RLS	158	32	Release heap. Set GDIRP to <u>nil</u> , then store word pointed to by tos into NP.

5. B TOP OF STACK ARITHMETIC AND COMPARISONS

5. B. 1 LOGICAL

LAND	132		Logical and. <u>And</u> tos into tos-1.
LOR	141		Logical or. <u>Or</u> tos into tos-1.
LNOT	147		Logical not. Take one's complement of tos.
EGUBOOL	175	6	Boolean =, <=, <, >=, and > comparisons.
NEGBOOL	183	6	
LEGBOOL	180	6	
LESBOOL	181	6	
GEGBOOL	176	6	
GTRBOOL	177	6	

Compare bit 0 of tos-1 to bit_0 of tos and push true or false.

5. B. 2 INTEGER

ABI	128	Absolute value of integer. Take absolute value of integer tos. Result is undefined if tos is initially -32768.
ADI	130	Add integers. Add tos and tos-1.
NGI	145	Negate integer. Take the two's complement of tos.
SBI	149	Subtract integers. Subtract tos from tos-1.
MPI	143	Multiply integers. Multiply tos and tos-1. This instruction may cause overflow if result is larger than 16 bits.
SGI	152	Square integer. Square tos. May cause overflow.
DVI	134	Divide integers. Divide tos-1 by tos and push quotient. (PDP11 quotient defined as in Jensen and Wirth; ZBO/BOBO quotient defined by floor(tos-1/tos).)
MODI	142	Modulo integers. Divide tos-1 by tos and push the remainder (as defined in Jensen and Wirth).
CHK	136	Check against subrange bounds. Insure that tos-1 \leq tos-2 \leq tos, leaving tos-2 on the stack. If conditions are not satisfied a run-time error occurs.
EQUI	195	Integer =,
NEGI	203	0,
LEGI	200	\leq ,
LESI	201	$<$,
GEQI	196	\geq ,
GTRI	197	and $>$

comparisons. Compare tos-1 to tos and push true or false.

5. B. 3 REALS

All over/underflows cause a run-time error.

FLT	138	Float top-of-stack. The integer tos is converted to a floating point number.
-----	-----	--

FLO	137	Float next to top-of-stack. Tos is a real, tos-1 is an integer. Convert tos-1 to a real number.
TNC	158 22	Truncate real. The real tos is truncated (as defined in Jensen and Wirth) and converted to an integer.
RND	158 23	Round real. The real tos is rounded (as defined in Jensen and Wirth), then truncated and converted to an integer.
ABR	129	Add reals. Take the absolute value of the real tos.
ADR	131	Add reals. Add tos and tos-1.
NGR	146	Negate real. Negate the real tos.
SBR	150	Subtract reals. Subtract tos from tos-1.
MPR	144	Multiply reals. Multiply tos and tos-1.
SGR	153	Square real.
DVR	135	Divide reals. Divide tos-1 by tos.
POT	158 35	Power of ten. The integer tos is checked for $0 \leq \text{tos} \leq 38$, a run-time error occurring if the conditions aren't satisfied. The implementation dependent value 10^{tos} is pushed. This facility allows the rest of the system to be independent of floating point format.
SIN	158 24	Sine. Take the sine of the real tos.
COS	158 25	Cosine.
ATAN	158 27	Arctangent.
EXP	158 29	Exponential. e^{tos} .
LN	158 28	Natural logarithm.
LOG	158 26	Log base 10.
SQT	158 30	Square root.
EGUREAL	175 2	Real =,
NEGREAL	183 2	\neq ,
LEGREAL	180 2	\leq ,
LESREAL	181 2	$<$,
GEGREAL	176 2	\geq ,
GTRREAL	177 2	$>$ and $>$ comparisons.
		Push TRUE or FALSE.

5. B. 4 SETS

ADJ 160 UB

Adjust set. The set tos is forced to occupy UB words, either by expansion (putting zeroes "between" tos and tos-1) or compression (chopping of high words of set), and its length word is discarded.

SGS 151

Build a singleton set. The integer tos is checked to insure that $0 \leq \text{tos} \leq 4079$, a run-time error occurring if not. The set [tos] is pushed.

SRS 148

Build a subrange set. The integers tos and tos-1 are checked as in SGS, and the set [tos-1..tos] is pushed. (The set [] is pushed if $\text{tos-1} > \text{tos}$.)

INN 139

Set membership. See if integer tos_1 is in set tos, pushing TRUE or FALSE.

UNI 156

Set union. The union of sets tos and tos-1 is pushed. (Tos or tos-1.)

INT 140

Set intersection. The intersection of sets tos and tos-1 is pushed. (Tos and tos-1.)

DIF 133

Set difference. The difference of sets tos-1 and tos is pushed. (tos-1 and not tos.)

EGUPWR 175 8
 NEGUPWR 183 8
 LEGUPWR 180 8
 GEGUPWR 176 8

Set =, \subseteq , \subset , \supseteq , and \supset
 (subset of), and \supseteq
 (superset of) comparisons.

5. B. 5 STRINGS

EQUSTR 175 4
 NEGSTR 183 4
 LEGSTR 180 4
 LESSTR 181 4
 GEGSTR 176 4
 GTRSTR 177 4

String =, \subseteq , \subset , \supseteq , and \supset
 comparisons. The string pointed to by word pointer tos-1 is lexicographically compared to the string pointed at by tos.

5. B. 6 BYTE ARRAYS

EGUBYT	175	10
NEGBYT	183	10
LEGBYT	180	10
LESBYT	181	10
GEGBYT	176	10
GTRBYT	177	10

Byte array = $\langle \rangle$, $\langle =$, \langle , $\rangle =$, and \rangle
 comparisons. $\langle =$, \langle , $\rangle =$, and \rangle are only
 emitted for packed arrays of char.

5. B. 7 ARRAY AND RECORD COMPARISONS

EQUWORD	175	12
NEGWORD	183	12

Word or multiword structure = $\langle \rangle$ and $\langle \rangle$
 comparisons.

5. C JUMPS

Simple (non-case statement) jumps are all two bytes long. The first byte is the op-code, the second is a SB jump offset. If this offset is non-negative, it is simply added to IPC. (A value of zero for the jump offset will make any jump a two-byte nop.) If SB is negative, then SB div 2 is used as a word offset into JTAB, and IPC is set to the byte address(JTAB \wedge [SB div 2]) - JTAB[SB div 2].

UJP	185	SB	Unconditional jump. Jump as described above.
FJP	161	SB	False jump. Jump if tos is false.
EFJ	211	SB	Equal false jump. Jump if integer tos $\langle \rangle$ tos-1. Not implemented in I.4.
NFJ	212	SB	Not equal false jump. Jump if integer tos = tos-1. Not implemented in I.4.
XJP	172	W ₁ .W ₂ .W ₃ .	\langle case table \rangle

Case jump. W₁ is word-aligned, and is the minimum index of the table. W₂ is the maximum index. W₃ is an unconditional jump instruction past the table. The case table is W₂-W₁+1 words long, and contains self-relative locations.

If tos, the actual index, is not in the range W₁..W₂, then IPC is pointed at W₃. Otherwise, tos-W₁ is used as an index into the table, and IPC is set to byte_address(casetable[index-min_index])-casetable[index-min_index].

5. D PROCEDURE AND FUNCTION CALLS AND RETURNS

The general scheme used in procedure/function invocation is

- 1) Calculate the `data_size` and `parameter_size` of the called procedure by using the information in the current procedure dictionary (pointed to by `SEG`).
- 2) Extend stack by `data_size` bytes.
- 3) Copy `parameter_size` bytes from the old top-of-stack to the beginning of the space just allocated.
- 4) Build a MSCW, saving `SP`, `IPC`, `SEG`, `JTAB`, `MP`, and a pointer to the most recent activation record of the called procedure's immediate parent.
- 5) Calculate new values for `SP`, `IPC`, `JTAB`, `MP`, and if necessary, `SEG`. Check for stack overflow.
- 6) If the called procedure has a lex level of `-1` or `0` save `BASE` and calculate a new `BASE`.

CLP	206	UB	Call local procedure. Call procedure UB, which is an immediate child of the currently executing procedure and in the same segment. Static link of MSCW is set to old MP.
CGP	207	UB	Call global procedure. Call procedure UB, which is at lex level 1 and in same segment. The static link of the MSCW is set to BASE.
CIP	174	UB	Call intermediate procedure. Call procedure UB in same segment as the currently executing procedure. The static link of the MSCW is set by looking up the call chain until an activation record is found whose caller had a lex level one 1 less than the procedure being called. Use that activation record's static link as the static link of the new MSCW.
CBP	194	UB	Call base procedure. Call procedure UB, which is at lex level <code>-1</code> or <code>0</code> . The static link of the MSCW is set to the static link in BASE's activation record. The BASE is saved, after which it is pointed at the activation record just created.
CXP	205	DB_1,UB_2	Call external procedure. Used to call <u>any</u> procedure not in the same segment as the calling procedure, including procedures at lex level <code>-1</code> or <code>0</code> . It works as follows: 1) Is desired segment in memory? This is determined by traversing up the call chain until an activation record of a procedure in the desired segment is found, or the operating system's resident

CSP -- eds note: *it was pointed out that op-code 158 is CSP, and is scattered throughout this document. This will be cleared up in the next major documentation effort.*

activation record is encountered.

2a) no: read in segment from disk using the information in the segment dictionary, then build an activation record. However, extend stack by `data_size+paramsize` in step 2.

2b) yes: build activation record normally.

3) calculate the dynamic link for the MSCW: If the called procedure has a lex level of -1 or 0, set as in CBP, otherwise set as in CIP.

RNP 173 DB

Return from non-base procedure. DB is the number of words that should be returned as a function value (0 for procedures, 1 for non-real functions, and 2 for real functions). DB words are copied from the bottom of the data segment and "pushed" onto the caller's top-of-stack. The information in the MSCW is then used to restore the caller's correct environment.

RBP 193 DB

Return from base procedure. The saved base is moved into BASE, after which things proceed as in the RNP instruction.

EXIT 158 4

Exit from procedure. Tos is the procedure number, `tos-1` is the segment number. This operator sets IPC to point to the exit code of the currently executing procedure, then sees if the current procedure is the one to exit from. If it is, control returns to the instruction fetch loop.

Otherwise, each MSCW has its saved IPC changed to point to the exit code of the procedure that invoked it, until the desired procedure is found.

If at any time the saved IPC of main body of the operating system is about to be changed, a run-time error occurs.

5. E SYSTEMS PROGRAMS SUPPORT PROCEDURES

See Section 2.1 for description of these procedures.

BYTE ARRAY PROCEDURES

FLC 158 10 Fillchar(dst, len, char).

SCN	158 11	Scan(maxdisp, start, forpast, char, mask).
MVL	158 02	Moveleft(src, dst, numbytes).
MVR	158 03	Moveright(src, dst, numbytes).

COMPILER PROCEDURES (still undocumented)

TRS	158 08	Treearch.
IDS	158 07	Idsearch.

DEBUGGER

BPT	213	Breakpoint (conditional HALT)
-----	-----	-------------------------------

MISCELLANEOUS

TIM	158 09	Time.
XIT	214	

* INTRODUCTION TO THE PASCAL PSEUDO-MACHINE * * Section 3.5 *

Version 1.5

September 1978

UCSD uses an interpreter based implementation of Pascal. This implementation is interpreter based. This means that the compiler emits code for a pseudo-machine which is emulated at run time by a program written in the machine language of the host. The compiler, program editor, small stand-alone operating system, and various utilities are themselves written in Pascal and run on the same interpreter. Thus the entire system can be moved to a new host machine by rewriting the interpreter for the new host.

Figure 3.5.10 (the last page of this document) is a skeleton version of a large Pascal program, here-in-after referred to as "The Program". This document is a top-down description of the realization of that program on the UCSD Pascal system. We will make occasional use of a helpful coincidence: The Program is the framework of the portion of the UCSD Pascal environment that's written in Pascal.

If The Program were expanded to a complete Pascal system, it would consist of at least 6000 lines of Pascal and compile to more than 30,000 bytes of code--too big to fit all at once into the memory of a small machine (by our current definition of small). We have therefore extended Pascal so that a programmer can explicitly partition a program into segments; only some of which need be resident in main memory at a time. The syntax of this extension is shown in figure 3.5.1. (Any syntactic objects not defined explicitly there retain their standard interpretation as defined by Jensen & Wirth: Pascal User Manual and Report.)

```
<program> ::= <program heading> <segment block> .  
  
<segment block> ::= <label declaration part>  
    <constant declaration part> <type definition part>  
    <variable declaration part> <segment declaration part>  
    <segment body>  
  
<segment declaration part> ::= SEGMENT <procedure heading>  
    <segment block>; \ SEGMENT <function heading>  
    <segment block>;  
  
<segment body> ::= <procedure and function declaration part>  
    <statement part>
```

FIGURE 3.5.1. SEGMENT DECLARATION SYNTAX.

Segment declaration syntax (figure 3.5.1) requires that all nested segments be declared before the ordinary procedures or functions of the segment body. Thus, a code segment can be completely generated before processing of code for the next segment starts. This is not a functional limitation, since forward declarations can be used to allow nested segments (COMPILER in The Program) to reference procedures in an outer segment body (CLEARSCREEN). Similarly, segment procedures and functions can themselves be declared forward.

Segmenting a program does not change its meaning in any fundamental sense. When a segment is called (e.g. the COMPILER segment in line A), the interpreter checks to see if it is present in memory due to a previous invocation. If it is, control is transferred and execution proceeds; if not, the appropriate code segment must be loaded from disk before the transfer of control takes place. When no more active invocations of the segment exist, its code is removed from memory. For instance, in The Program, the code for the COMPINIT segment is not present in memory either before or after the execution of line A. Clearly, a program should be segmented in such a way that (non-recursive) segment calls are infrequent; otherwise, much time could be lost in unproductive thrashing (particularly on a system with low performance disk).

			high address
	—>	DEBUGGER	10
not			
	—>	FILER	17
shown			
		EDITOR	12
in			
		COMPINIT	7
the			
		COMPILER	41
program			
	—>	INITIALIZE	3
		USER PROGRAM	1
		PASCALSYSTEM	17
		SEGMENT DICITONARY	1
			low address

FIGURE 3.5.2. PASCAL SYSTEM CODE FILE.

The code file resulting from compilation of The Program is diagrammed in figure 3.5.2*. The file is a sequence of code segments preceded by a segment dictionary. The size of each segment is noted in blocks, the 512-byte disk allocation quantum used on most PDP-11 operating systems. The sizes indicated are representative of a full Pascal system. Each code segment begins on a block boundary. The ordering (from low address to high address) is determined by the order that one encounters segment procedure bodies in passing through The Program.

* An overview of the relationship between figures 3.5.2 through 3.5.8 (to be discussed in the following pages) is given in figure 3.5.9 at the end of this section. It is helpful to study figure 3.5.9 at this point for a better understanding of the section.

The segment dictionary in the first block of a code file contains an entry for each code segment in the file. The entry includes the disk location and size (in bytes) for the segment. The disk location is given as relative to the beginning of the segment dictionary (which is also the beginning of the code file) and is given in number of blocks. This information is kept in the system communications area (also called SYSCOM) during the execution of the code file, and is used in the loading of non-present segments when they are needed. Figure 3.5.3 details the layout of the table and shows representative contents for the Pascal system code file.

location	1	PASCALSYSTEM
size	8500	
	18	USERPROGRAM
	variable	
	22	COMPILER
	20932	
	63	COMPINIT
	3480	
	70	DEBUGGER
	5880	

FIGURE 3.5.3. THE SEGMENT DICTIONARY

A code segment contains the code for the body of each of its procedures, including the segment procedure, itself. Figure 3.5.4 is a detailed diagram of the code segment of The Program (Pascalsystem). Each of a code segment's procedures are assigned a procedure number, starting at 1 for the segment procedure, and ranging as high as 255 (current temporary limit of 127). All references to a procedure are made via its number. Translation from procedure number to location in the code segment is accomplished with the procedure dictionary at the end of the segment. This dictionary is an array indexed by the procedure number. Each array element is a self-relative pointer to the code for the corresponding procedure. Since zero is not a valid procedure number, the zero'th entry of the dictionary is used to store the segment number (even byte) and number of procedures (odd byte). Observe that CLEARSCREEN is the first procedure for which code is generated and that it appears at the beginning of the segment. The outer block code is generated and appears last.

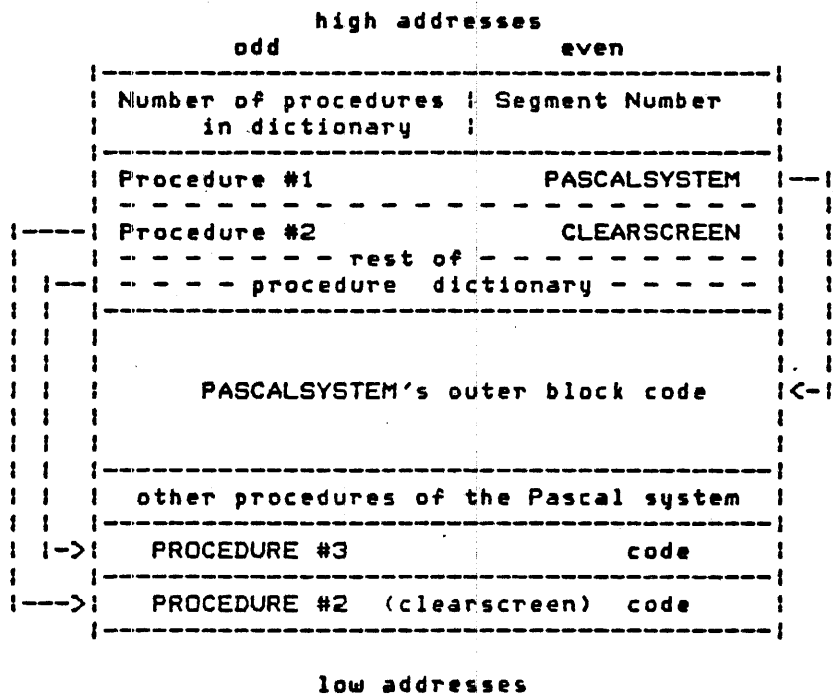


FIGURE 3.5.4. A CODE SEGMENT

A more detailed diagram of a single procedure code section is seen in figure 3.5.5. It consists of two parts: the procedure code itself in the lower portion of the section) and a table of attributes of the procedure. These attributes are:

LEX LEVEL: This odd byte is the depth of absolute lexical nesting for the procedure. (i.e. Lex Level (LL) Pascalsystem=-1, LL COMPILER or CLEARSCREEN=0, LL COMPINIT=1, etc.).

PROCEDURE NUMBER: This even byte refers to the number given in the procedure dictionary of the parent segment procedure. For example, the Procnum of CLEARSCREEN is 2. (see figure 3.5.4).

ENTER IC: This is a self-relative pointer to the first instruction to be executed for this procedure.

EXIT IC: This is a self-relative pointer to the beginning of the block of procedure instructions which must be executed to terminate procedure properly.

PARAMETER SIZE: The param size is the number of bytes of parameters passed to a procedure from its caller.

and **DATA SEGMENT SIZE:** The data size is the size of the data segment (See below) in bytes, excluding the markstack and PARAM SIZE.

Between these attributes and the procedure code there may be an optional section of memory called the "jump table". Its entries are addresses within the procedure code. JTAB is a term commonly applied to the six attributes just discussed and the jump table itself.

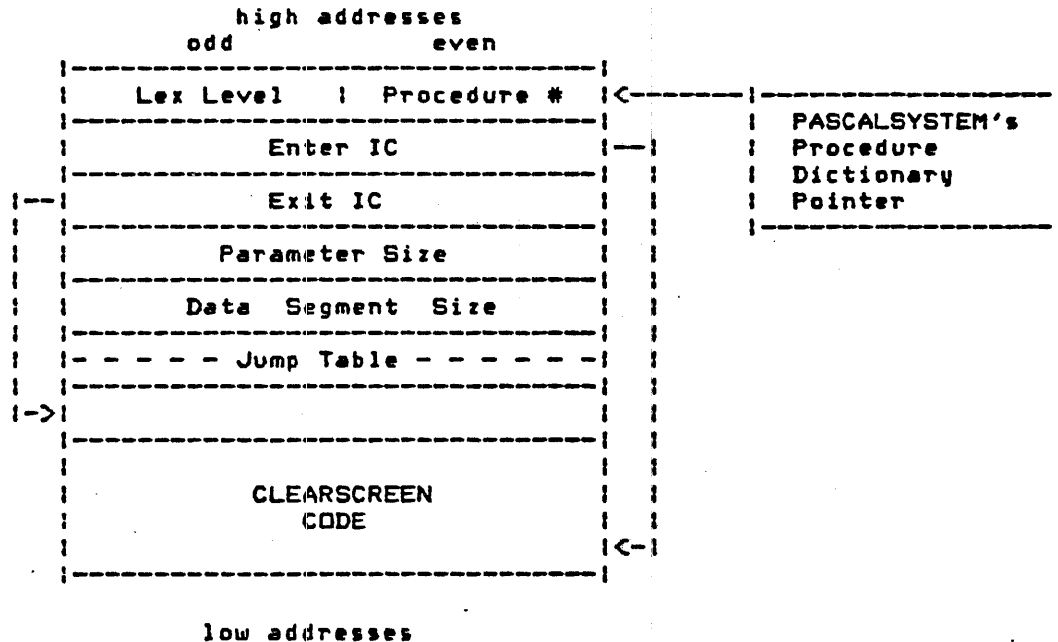


FIGURE 3.5.5. PROCEDURE CODE SECTION (OF CLEARSCREEN)

interpreter occupies the lowest area in memory. In it is the system communications area(also called SYSCOM),which is accessible both to assembly language routines in the interpreter and (as if it were part of the heap) to system routines coded in Pascal. It serves as an important communication link between these two levels of the system. The Pascal heap is next in the memory layout; it grows toward high memory. The single stack growing down from high memory is used for 3 types of items: 1) temporary storage needed during expression evaluation; 2) a data segment containing local variables and parameters for each procedure activation; and 3) a code segment for each active segment procedure. (See figure 3.5.6)

Consider the status of operations just before COMPINIT is called in line B. Conceptually, there are six pseudo-variables which point to locations in memory:

a STACK POINTER(SP):which points to the current top of the stack,

a MARK STACK POINTER(MP):which points to the "topmost" markstack in the stack,(remember that the the stack grows down!),

a SEGMENT(SEG) variable:which points to the base of the procedure dictionary for the currently active segment procedure. For example, just before COMPINIT is called, SEG points to the COMPILER segment's procedure dictionary,

an INTERPRETER PROGRAM COUNTER(IPC):which contains the address of the next instruction to be executed in the code segment of the current procedure,

a JTAB pointer:which points to the collection of procedure attributes and jump table entries in the body of the current procedure code section,

and a NEW POINTER(NP):which points to the current top of the heap.

When segment procedure COMPINIT is called in line B, its code segment (including all compiler initialization procedures) is loaded on the stack. The COMPINIT data segment is built on top of the stack. Figure 3.5.7 is a diagram of the data segment for COMPINIT.

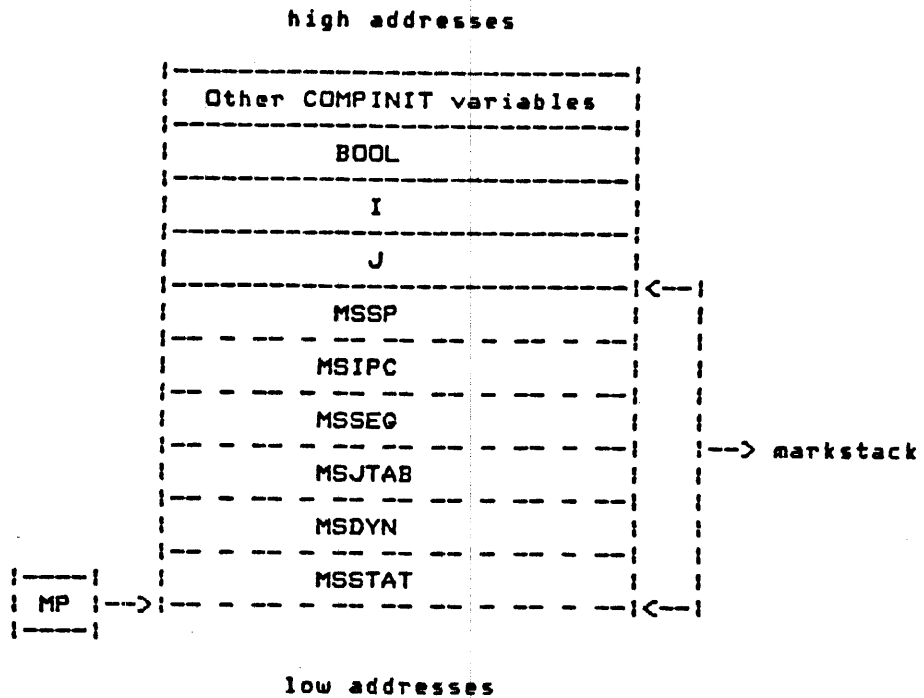


FIGURE 3.5.7. A DATA SEGMENT

In the upper portion of the data segment, space is allocated for variables local to the new procedure. For example, COMPINIT's data segment allocates space for integer variables I and J, as well as boolean BOOL.

In the lower portion of the data segment is a "markstack". When a call to any procedure is made, the current values of the pseudo-variables, which characterize the operating environment of the calling procedure, are stored in the markstack of the called procedure. This is so that the pseudo-variables may be restored to pre-call conditions when control is returned to the calling procedure.

For example, the call to COMPINIT causes conditions in COMPILER just before the call to be stored in COMPINIT's markstack in the following manner:

```

MarkStack DYNamic link (MSDYN) <-- MP
"      "      IPC(MSIPC) <-- IC
"      "      SEGment Pointer (MSSEG) <-- SEG
"      "      Jump TABLE (MSJTAB) <-- JTAB
"      "      Stack Pointer (SP) <-- SP

```

In addition a Static Link field becomes a pointer to the data segment of the lexical parent of the called procedure. In particular, it points to the Static Link field of parent's markstack. After the building of the data segment new values for IC, SEG, SP, MP, JTAB, and NP are established for the new procedure.

When the call to CLEARSCREEN is made on line C, another data segment is added to the stack and again the pseudo-variables are stored in the new markstack, as well as the appropriate Static Link, and updated. Note that now the SEG no longer points to the COMPINIT procedure dictionary, but to the Pascalsystem dictionary.

No code segment for CLEARSCREEN is added to the stack before the data segment since the code for CLEARSCREEN is already present in segment Pascalsystem. Its invocation causes only a data segment to be added to the stack. When CLEARSCREEN and INIT are completed, the COMPILER data segment will again be the top element on the stack.

Figure 3.5.8 is a detailed diagram of the stack during execution of an instruction in CLEARSCREEN, including appropriate pointers for static, dynamic, etc. links of CLEARSCREEN's markstack. Note where the pseudo-variables point in the stack. In particular, JTAB points inside CLEARSCREEN code section which is in the Pascalsystem code segment, IC points inside that CLEARSCREEN code, and SEG points to the base of the Pascalsystem code segment.

Figure 3.5.9 illustrates a top-down process by showing the relationships among diagrams 2 through 7.

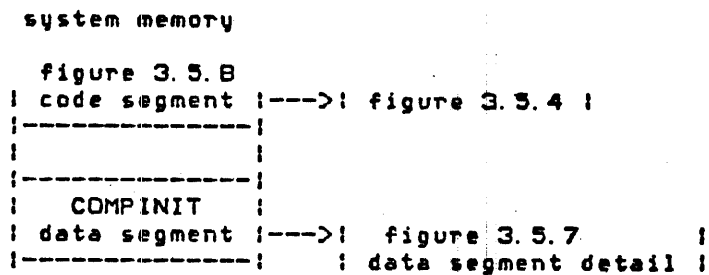
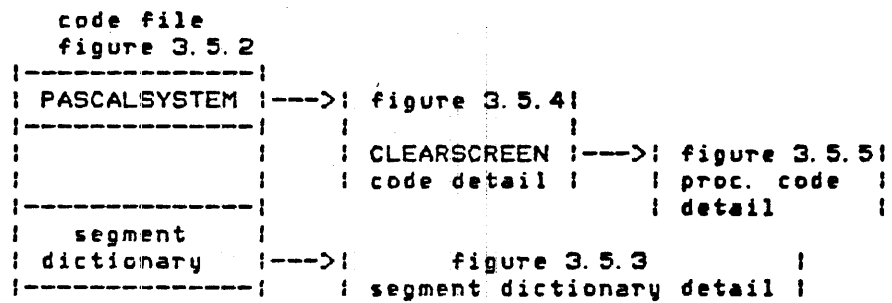


FIGURE 3.5.9. RELATIONSHIP OF DOCUMENT FIGURES

```

PROGRAM PASCALSYSTEM;
VAR
  SYSCOM: SYSCOMREC;
  CH: CHAR;

PROCEDURE CLEARSCREEN: FORWARD;

SEGMENT PROCEDURE USERPROGRAM;
  BEGIN
    ...
  END;
SEGMENT PROCEDURE COMPILER;
VAR
  SY, OP: INTEGER;
  SYMCURSOR: INTEGER;

  PROCEDURE INSYMBOL; FORWARD;

  SEGMENT PROCEDURE COMPINIT;
  VAR
    I, J: INTEGER;
    BOOL: BOOLEAN;
  BEGIN
    ...
    I := 1;
    CLEARSCREEN; -----LINE C
    INSYMBOL;
    ...
  END;

  PROCEDURE INSYMBOL;
  BEGIN ... END;

  PROCEDURE BLOCK;
  BEGIN ... END;
  BEGIN (*COMPILER*)
    ...
    COMPINIT; -----LINE B
    INSYMBOL;
    ...
  END; (*COMPILER*)

SEGMENT PROCEDURE EDITOR;
  BEGIN ... END;

PROCEDURE CLEARSCREEN
  BEGIN
    ...
    WRITE(-----);
    ...
  END;

BEGIN (*PASCALSYSTEM*)
  REPEAT
    READ(CH);
    CASE CH OF
      C: COMPILER; -----LINE A
      E: EDITOR;
      U: USERPROGRAM
    ...
  END(*CASE*)
  UNTIL CH = 'H'
END.

```

FIGURE 3.5.10. THE PROGRAM

 * BYTE-SWAPPING * * Section 3.6 *

Version I.5 September 1978

Byte-swapping problems occur when code generated on one machine is transferred to another or programs which directly interface with memory (e.g. the Patch utility) are written on or for one machine and transferred to another which has a different ordering for its memory.

There are two different ways to order bytes in a given memory:

- A) Byte Zero is the byte containing the least significant half of the word. Byte One contains the most significant half.
- B) Byte Zero is the byte containing the most significant half of the word. Byte One contains the least significant half.

The difference between these is the way Byte quantities are read and stored in memory. Word quantities, such as integers, will be read and looked at in the same way on both types of machines. However, byte quantities such as P-code or characters will be reversed.

An example:

DEFINITION	(A)			(B)		
	ls*		ms*	ms*		ls*
VALUE(Hex)	! 04	! 07	!	! 07	! 04	!
BYTE	0	1		0	1	

(least/most significant bit, thereby least/most significant byte)

If both of the bytes shown above were read as an integer , a word quantity, they would give the value 3,588. However, if the value of byte Zero was wanted (as in: C: PACKED ARRAY[0..1] OF CHAR;) then Definition A would show a value of 04H and Definition B would show a value of 07H. Both definitions would show the value 07H if the most significant byte were specified.

Byte-swapping is not a hard problem to solve, it just requires a little thought. The Patch utility has type declarations for both types of machines and a study of it should suffice to show how to satisfy your programming needs.

* THE CALCULATOR * * Section 4.1 *

Version I.5 September 1978

The prompt, '->', expects a one line expression in algebraic form. Up to 25 different variables are available, each with different values assigned using the syntax of the given grammar. Only the first 8 letters are used to distinguish between variables. Variables having a value may be used as constants. There are two built-in variables: PI (3.141593) and E (2.718282). These values may be changed by the user.

No distinction is made between upper and lower case letters.

The MOD function is the backslash '\': the PASCAL MOD function is used and the operands are rounded to be integers. WARNING: Since this uses the PASCAL defn. of MOD (see Jensen & Wirths' Pascal User Manual and Report Second Edition page 108) the results obtained may not be as expected.

The operand of the factorial function 'FAC' is also rounded to be an integer which must be between zero and thirty-three inclusive or the expression will be rejected.

The uparrow '^' is used for exponentiation. The operand must be positive or the expression will be rejected as $e^{Y \ln(X)}$ is used to calculate the answer.

'LASTX' is a constant which is assigned the value of the previous correct expression by the calculator and may be used in the following expression instead of inserting the same expression again.

Angles for the TRIG functions must be in RADIANS. Degree to Radian conversion is accomplished by $RADANGLE = (PI / 180) * DEGANGLE$.

This program will bomb on an execution error if an over or underflow occurs. If this happens all user assigned variables and their values will be lost.

To leave the calculator mode simply type <RET> immediately following the prompt.

EXAMPLE OF CALCULATOR SESSION:

```
-> PI
      3.141592
-> LASTX
      3.141592
-> HALFPI = PI / 2
      1.570796
-> SIN ( HALFPI )
      1.0
-> A = B = C = D = F = ( FAC (3) / 2 )
```


-> A 3.0
-> C 3.0
-> 1 + 2 3.0
-> 3 + 7 / 4 4.75
-> SQRT(2*2+3*3) 3.605551

 * LIBRARIAN UTILITY * * Section 4.2 *

Version I.5 September 1978

LIBRARY.CODE is a utility program that allows the user to link separately compiled PASCAL units and separately assembled subroutines into a LIBRARY file. It is based upon the original pre-I.5 utility LINKER.CODE and operates in basically the same way.

To add a segment to *SYSTEM.LIBRARY it is necessary to create a new file into which each segment that is wanted from the original *SYSTEM.LIBRARY is first linked. It is then possible to add segments by linking from another code file into the new file being created.

EXAMPLE

Consider the case of adding a segment called TURTLE to the already existing file *SYSTEM.LIBRARY which is assumed to contain the segments PSGRAPHICS and MOVETO.

On executing LIBRARY.CODE, the user is prompted for the name of the output codefile. For this example, respond with the name NEW.LIBRARY. The program now asks for a 'Link Code File'. The response here is *SYSTEM.LIBRARY. The names of all segments currently linked into the input library, i.e. *SYSTEM.LIBRARY, as well as their length in bytes is now displayed. Currently there are a maximum of 16 segments in any PASCAL program or LIBRARY.

0-	MOVETO	2398	4-	0	8-	0	10-	0
1-	PSGRAPHI	864	5-	0	9-	0	11-	0
2-		0	6-	0	10-	0	14-	0
3-		0	7-	0	11-	0	15-	0

The following promptline appears:

Segment # to link and <space>, N(ew file, Q(uit, A(bort

The user now enters the number of a segment within the link code file that is to be linked into the new library file, followed by <space>. Next, the number of the segment in the output file to be linked into (i.e. NEW.LIBRARY) is typed followed by <space>. For each segment linked the librarian reads that segment from the input file and writes it to the output file at the segment requested. It then displays the segment table for the current state of the output library file. In this example, respond with the following:

```

0<space>
Seg to link into? 0<space>
1<space>
Seg to link into? 1<space>

```

When all needed segments have been linked a new input file is requested by typing 'N' for N(ew file). In this example, a separately compiled PASCAL UNIT called TURTLE is assumed to exist in a codefile called TGRAPHICS.CODE. See section 3.2, UNITS. On entering the name of this file the following display appears:

```

0-          0  4-          0  8-          0 10-          0
1-          0  5-          0  9-          0 11-          0
2-          0  6-          0 10- TURTLE 230 14-          0
3-          0  7-          0 11-          0 15-          0

```

The Unit TURTLE occurs in segment 10 and is to be linked into segment 2 within NEW.LIBRARY. The user responds:

```

10<space>
Seg to link into? 2<space>

```

The final display of the output library segment table is thus:

```

0- MOVETO      2398  4-          0  8-          0 10-          0
1- PSGRAPHI    864  5-          0  9-          0 11-          0
2- TURTLE      230  6-          0 10-          0 14-          0
3-              0  7-          0 11-          0 15-          0

```

The output library codefile length is displayed and in this example is 16 (blocks long).

Once the needed segments from all input files have been linked in the user locks the output file by typing 'G' followed by a return, (unless a copyright notice is desired within the codefile). Type 'A' to abort the linking process. The old *SYSTEM.LIBRARY should either be removed or its name changed if it resides upon the same disk and the name NEW.LIBRARY must be changed to *SYSTEM.LIBRARY in order to be used.

NOTE

In response to the initial prompt "Output Code File ->" we could have just as easily said *SYSTEM.LIBRARY followed by another *SYSTEM.LIBRARY in response to the prompt "Link Code File ->". However, in this case the original *SYSTEM.LIBRARY will be removed automatically upon completion of the linking process.

* SETUP - SYSTEM RECONFIGURATION * * Section 4.3 *

Version 1.5 September 1978

The UCSD Pascal Operating System keeps certain information about the user in a file called SYSTEM.MISCINFO. During each system initialization this file is read into memory, and from there it is accessed by many parts of the system, particularly (if the user has a terminal suitable for it) by the screen oriented editor.

Much of this information needs to be initially set up by the user to conform to his particular hardware configuration or his taste or convenience. Most of this information concerns the nature of his terminal and keyboard, although there are a few miscellaneous fields.

SETUP is run like any other compiled Pascal program, by entering the Command level of the system, typing X for eXecute and typing the filename SETUP followed by a carriage return.

SETUP: C(HANGE) T(EACH) H(ELP) Q(UIT)

If this does not happen it may be because the setup program is not on the disk. If so, the system will display the message

no file setup.CODE

If neither of the above happens, something is drastically wrong. Contact UCSD. Assuming all is well, continue.

All commands to the SETUP program are invoked by typing a single letter chosen from the promptline.

SETUP: C(HANGE) T(EACH) H(ELP) Q(UIT)

Type 'H' to find out what the commands at this level do. The program is self teaching, so the rest of this document explains the information SETUP was designed to change.

SETUP does not tell the system how to do random access cursor addressing on the user's terminal (for those terminals which have this capability). To allow the system to use that feature, please refer to Section 4.7 of this document package.

4.3.1 MISCELLANEOUS INFORMATION

HAS CLOCK

Values: TRUE, FALSE

A real time clock is available. A real time clock module, such as the DEC KW11, may be found on many processors. It is assumed to be a line frequency (60 cycle) clock. If available it is used by the PASCAL system to optimize disk directory updates. See section 2.1.6 TIME intrinsic.

HAS B510A

Values: TRUE, FALSE

The system is running on a Terak B510a hardware configuration.

4.3.2 GENERAL TERMINAL INFORMATION

HAS SLOW TERMINAL

Values: TRUE, FALSE.

When this field is true, the system issues abbreviated promptlines and messages.

Suggested setting: 600 baud and under -- True, otherwise False.

HAS RANDOM CURSOR ADDRESSING

Values: TRUE, FALSE

Only applies to video terminals. See Section 4.7 in order to allow the system to make use of this feature.

HAS LOWER CASE

Values: TRUE, FALSE

SCREEN WIDTH

The number of characters per line of a terminal.

SCREEN HEIGHT

The number of lines per display screen of a video terminal.

Set to 0 for a hard copy terminal or other terminal in which paging is not appropriate.

NON-PRINTING CHARACTER

Values: Any printing character.

What should be displayed by the terminal to indicate the presence of a non-printing character.

Recommended setting: ASCII "?".

VERTICAL MOVE DELAY

The number of nulls to send after a vertical cursor move. Many types of terminals require a delay after certain cursor movements which enables the terminal to complete the movement before the next character is sent. This number of nulls will be sent after carriage returns, ERASE TO END OF LINE, ERASE TO END OF SCREEN and MOVE CURSOR UP.

4.3.3 CONTROL KEY INFORMATION

The user may choose which control keys suit his particular keyboard arrangement and his taste.

Some keyboards generate two codes when some single key is pressed. If that is the case for any of the keys mentioned here, it must be noted in the field PREFIXED [<fieldname>] which has either the value TRUE or the value FALSE. The prefix for all such keys must be the same and must be noted in the field LEAD-IN FROM KEYBOARD. This feature may also be used to access control functions with two-character sequences if a user's keyboard is unable to generate many control characters. As an example, suppose the user's keyboard had a vector pad which generated the value pairs ESC "U", ESC "D", ESC "L" and ESC "R" for the keys for Uparrow, Downarrow, Leftarrow and

Rightarrow, respectively. Assume also that all other keys on the keyboard generate only single codes. Then the user would give the following fields the following values:

KEY FOR MOVING CURSOR UP	ASCII "U"
KEY FOR MOVING CURSOR DOWN	ASCII "D"
KEY FOR MOVING CURSOR LEFT	ASCII "L"
KEY FOR MOVING CURSOR RIGHT	ASCII "R"
LEAD-IN KEY FOR KEYBOARD	ESC
PREFIXED[KEY FOR MOVING CURSOR UP]	TRUE
PREFIXED[KEY FOR MOVING CURSOR DOWN]	TRUE
PREFIXED[KEY FOR MOVING CURSOR LEFT]	TRUE
PREFIXED[KEY FOR MOVING CURSOR RIGHT]	TRUE

KEY FOR STOP

Console output stop character. The STOP character is a toggle; when pressed, the key will cause output to the file 'OUTPUT' to cease. When the key is depressed again, the write to file 'OUTPUT' will resume where it left off. This function is very useful for reading data which is being displayed faster than one can read.

Suggested setting: ASCII DC3

KEY FOR FLUSH

Console output cancel character. Similar in concept and usage to the STOP key, the FLUSH key will cause output to the file 'OUTPUT' to go undisplayed until FLUSH is pressed again or the system writes to file 'KEYBOARD'. Note that, unlike the STOP key, processing continues uninterrupted while output goes undisplayed.

Suggested setting: ASCII ACK

KEY FOR BREAK

Typing the character BREAK will cause the program currently executing to be terminated with a run-time error immediately.

Suggested setting: Something difficult to hit accidentally.

KEY TO END FILE

Console end of file character. When reading from the files KEYBOARD or INPUT or the unit 'CONSOLE:', this key sets the Boolean function EOF to TRUE. See section 2.2.4 EOF intrinsic.

Suggested setting: ASCII ETX

KEY TO DELETE CHARACTER

Each time you press this key one character is removed from the current line, until nothing is left on that line.

Suggested setting: ASCII BS

KEY TO DELETE LINE

Depressing LINE DELETE will cause the current line of input to be erased.

Suggested setting: ASCII DEL

The rest of this section contains information only of interest to users who are using video display terminals with a selective erase capability and may be safely ignored by users having any other kind of terminal, such as hardcopy terminals or storage tube terminals.

KEY TO MOVE CURSOR UP
KEY TO MOVE CURSOR DOWN
KEY TO MOVE CURSOR LEFT
KEY TO MOVE CURSOR RIGHT

These keys are used by the screen oriented editor to control the basic motions of the cursor. If the keyboard has a vector pad, set these fields to the values it generates, otherwise, we suggest choosing 4 keys in the pattern of a vector pad and use the control codes which correspond to them, for example the keys 'O', '.', 'K' and ']' on most keyboards encircle an imaginary vector pad. You may wish to use a prefix character before such keys as described above.

EDITOR "ESCAPE" KEY

The key which, in the system screen oriented editor, is to be used to escape from commands, reversing any action taken.
Suggested setting: ASCII ESC

EDITOR "ACCEPT" KEY

The key which, in the system screen oriented editor, is to be used to accept commands, making permanent any action taken.
Suggested setting: ASCII ETX

4.3.4 VIDEOT SCREEN CONTROL CHARACTERS

This section describes the characters which, when sent to the terminal by the computer, controls the terminal's actions. You should consult the manual for your terminal to find the appropriate values. If a terminal does not have one of these characters, the field should be set to 0 unless otherwise directed.

Some screens require a two character sequence to exercise some of their functions. If the first character in all of these sequences is the same, it can be set as the value of the field LEAD-IN TO SCREEN and for each <fieldname> which requires that prefix, the user must set the field PREFIX[<fieldname>] to TRUE. For example, suppose ERASE TO END OF LINE and ERASE TO END OF SCREEN were respectively performed by the sequences ESC "L" and ESC "S" but all the other screen controls were single characters. The user would then set the following fields to the following values:

LEAD-IN TO SCREEN	ASCII ESC
ERASE TO END OF LINE	ASCII "L"
ERASE TO END OF SCREEN	ASCII "S"
PREFIXED[ERASE TO END OF SCREEN]	TRUE
PREFIXED[ERASE TO END OF LINE]	TRUE.

ERASE TO END OF SCREEN

The character which erases the screen from the current cursor position to the end of the screen.

ERASE TO END OF LINE

The character which, when sent to the screen, erases all characters from the current cursor position to the end of the line the cursor is on.

ERASE LINE

The character which, when sent to the screen, erases all the characters on the line the cursor is currently on.

ERASE SCREEN

The character which, when sent to the screen, erases the entire screen.

BACKSPACE

The character which, when sent to the screen, causes the cursor to move space to the left.

MOVE CURSOR HOME

The character which moves your cursor to the upper left of the current page. **IMPORTANT:** If your terminal does not have such a character, set this field to CARRIAGE RETURN, ASCII mnemonic CR.

MOVE CURSOR UP

MOVE CURSOR LEFT

The characters which move your cursor non-destructively one space in those directions.

* BOOTSTRAP COPIER * * Section 4.4 *

Version 1.5 September 1978

The bootstrap copier `BOOTER.CODE` asks for the unitnumber of the volume on which to write the bootstrap. Refer to Table 5 for a list of volume numbers. It will then ask for a file name to write as the bootstrap. It writes the first two blocks of that file, so in order to copy the bootstrap from an existing disk, give it the diskname, and it will copy the bootstrap from the disk named to the unit numbered.

To execute the `BOOTER` program, type `X BOOTER` to Command level (assuming that there a copy of `BOOTER.CODE` on the disk).

* PATCH * * Section 4.5 *

Version I.5 September 1978

On X(ecuting PATCH, the promptline is

C(onsole, P(atchwrite, W(holewrite, G(uit

The options available are:

Working with, and altering the file in the C(onsole mode.
Dumping the file in a Hex, Decimal, Octal, or ASCII format, in the
P(atchwrite mode.
Dumping/concatenating and/or moving blocks in files with the
W(holewrite mode.
Leaving PATCH with the G(uit command.

In the C(onsole mode, the promptline changes with each command.
The promptline always reflects the commands available at any given
time, and no more. The full promptline is:

Patch: R(ead, S(ave, H(ex, M(ixed, G(et, G(uit [nn]

The number in square brackets at the end of the prompt is the current
block being patched. The first command to use is G(et. G(et will
prompt

Filename: <cr for unit i/o>

Respond to this prompt with the name of the file to be
patched. If the disk/device has no directory, or has some problem with
the directory, reference it by its Pascal unitnumber. Type a carriage
return to this prompt, and the prompt is:

Unitnum to patch [4,5,9..12] (0 will Quit)

Having typed a successful entry to one of the two above prompts, the
prompt will now be extended by the R(ead command. R(ead will read up a
block from the file/unit. The prompt on entering R(ead command is

BLOCK:

Respond with a block number in the file/unit specified. There
is no range checking provided on this read, so exercise care in the
number typed. The promptline is now extended with H(ex, M(ixed and
the block number in square brackets. H(ex and M(ixed display the
block read. Using the H(ex command displays the block entirely in
hexadecimal characters, using the M(ixed command will display printing
ASCII characters where possible, and hexadecimal values elsewhere. The
promptline is:

Alter: pad vector 1,5,3,0-0..F hex characters, S(tuff, Q(uit

The vector keys on the terminal causes the cursor to move around in the data, notice that there the cursor will remain only on the data, and will not move off the data. Typing a hexadecimal character changes the character the cursor is over provided that only one or more of the data positions is changed, when Q(uitting from Alter mode, the Patch promptline will be extended with the S(ave command. Typing S(ave writes the changed data back to from where it was read. In the Alter mode, there is one optional command: S(tuff. Typing the S(tuff command displays the promptline:

Stuff for how many bytes:

Key a number from 0 to 512. Type carriage return to cause patch to accept the number, the promptline changes to:

Fill with what hex pair:

Key a byte value in hexadecimal. The data reappears on the screen, with the number of bytes specified, from the position of the cursor filled with the data value specified, to the hex pair prompt.

Using the Patchwrite command causes a full screen prompt to appear:

This procedure writes out sequential blocks to any file as a patch dump. Type the prefix character of the option to be changed. Type 'P' to PRINT, 'G' to GUIT.

A(Input File
B(Begin Block #
C(Num. of Blocks

E(Output File

G(Hexadecimal
H(ASCII
I(Decimal
J(Octal
K(Decimal Bytes
L(Octal Bytes
M(Krunch
N(Double Space

Following each of the fields is the current value of that field. Typing the character in front of the field places the cursor after the field, and removes the current value. Typing 'Y' or 'T' sets a boolean value to True, any other character sets the field to False. The Input File and Output File fields require a filename to be typed followed by carriage return. The integer fields (Begin Block, and Num. of Blocks) require a number to be typed followed by carriage return or space. Any other character sets the value of the field to some unspecified value.

The other options at the Patchwrite level are Print and Quit. Both cause Patch to return to the outer level. Quit does it straight away, Print dumps out the file in the requested format on the way. The options available for the dump need to be selected, the default is none. The options Krunch and Double Space affect the formatting of the output. Krunch, when true, removes blank lines between logical output lines. Double Space when true, double spaces all output.

Using the W(holewrite command causes the full page prompt:

This procedure writes any number of blocks from an existing file to a new file, unchanged. Simply specify the necessary parameters Type 'P' to PUT, 'Q' to QUIT

I(nput File
S tart Block
N(umber of Blcks

O(utput File

The protocol for changing the fields at this level is the same as that for the Patchwrite level. The Wholewrite level is that which allows one to mix/match and mingle files. Put and Quit both cause Patch to return to the outer level, Put writes to the file on its way, Quit does not.

Notice that the Patchwrite and Wholewrite levels remember their vital parameters across sessions (while remaining in Patch). The Console level will clear all memory of the session. The Patchwrite level paginates its output, after each block written, a form-feed is generated. (Specifically PAGE(OUTPUTFILE)).

* RT11 to PASCAL CONVERSION KIT * * Section 4.6 *

Version I.5 September 1978

The utility file labeled RT11TOEDIT is intended for use with RT-11 disks. It assumes the presence of an RT-11 directory spanning blocks 6-7. When the file is executed it asks the user to specify the Pascal system unitnumber of the volume of which the user wants to view the directory. Once a legal on-line unit has been specified, RT11TOEDIT reads each entry on blocks 6-7. The program uses the UNITREAD intrinsic to read the directory and does not open the file in the usual manner. It lists on the screen the entire contents of the directory. For each entry it specifies the file title, file kind, the size of the file in blocks, and the starting block location of the file (in base 10). All unused portions are identified as such. The user will be prompted for an RT-11 file name, a Pascal system file name, and finally a mode of transfer.

* GOTOXY BINDER * * Section 4.7 *

Version I.5 September 1978

This program alters the SYSTEM.PASCAL on the default P(refix disk. It prompts for 'local GOTOXY', a procedure which must be created and bound into the system (only once) in order to make the system communicate correctly with the screen.

Look at the file GOTOXY.TEXT on the release disk. This file contains a few procedures for doing GOTOXY cursor addressing on a few different CRT-type terminals. If the procedure needed is one of those, remove it from comments, comment out any others, recompile it, and run BINDER on it. BINDER is a self-instructing program.

If the GOTOXY cursor-addressing scheme for the terminal is not there, create one. The procedure may not be named GOTOXY because this identifier is predeclared at the "\$U-" level of compilation.

Possible error:

Nil memory reference at
compile time

Value range error when executing
BINDER

Fix:

Remove the program heading
and try again

(*\$U-*) should be the first
thing in the GOTOXY file

Assumptions:

- 1.) A screen terminal
- 2.) A PASCAL system
- 3.) The upper left-hand corner of the screen is X=0, Y=0.

- Notes -

```
(*U-*)
PROCEDURE FGOTOXY(X,Y: INTEGER);
BEGIN
  IF X < 0 THEN X := 0;
  IF X > 79 THEN X := 79;
  IF Y < 0 THEN Y := 0;
  IF Y > 23 THEN Y := 23;
  write(CHR(27), 'y', CHR(Y+32), CHR(X+32))
END;

BEGIN (* Dummy main block *)
END.
```

(*This routine should work for the DEC VT-52*)

```
(*U-*)
PROGRAM PASCALSYSTEM;
(*GOTOXY for SCRCC IQ12C*)
PROCEDURE IQ12OXY(X,Y: INTEGER);
VAR P: PACKED ARRAY[0..3] OF CHAR;
BEGIN
  IF Y > 23 THEN Y := 23;
  IF X > 79 THEN X := 79;
  IF Y < 0 THEN Y := 0;
  IF X < 0 THEN X := 0;
  P[0] := CHR(27);
  P[1] := '=';
  P[2] := CHR(Y+32);
  P[3] := CHR(X+32);
  UNITWRITE(2, P, 0);
END;
BEGIN END.
```

 * DUPLICATE DIRECTORY UTILITIES * * Section 4 B *

Version 1.5 September 1978

COPYDUPDIR

This program will copy the duplicate directory into the primary directory location. If the disk is not currently maintaining a current directory the program will tell you so.

To use this program e(x)ecute COPYDUPDIR. The program will ask for the drive in which the copy is to take place (4 or 5). If no duplicate directory is found it will tell you after you indicate the drive unit. If the duplicate is found then it will ask you if your sure you want to distroy the directory in blocks 2-5. A 'Y' will execute the copy any other character will abort the program.

MARKDUPDIR

This program will mark a disk that is currently not maintaining a duplicate directory so that it will. Caution must be exercised to be sure that blocks 6-9 are free for use. If they are not one must rearrange the files as to make them free. One can tell if there available by getting an E)xtended listing in the Filer and checking to see where the first file starts. If the first file starts at block 6 or the first file starts at block 10 but there is a 4 block unused section at the top, then the disk has not been marked. If however, the first file starts at block 10 and there is no unused blocks at the beginning of the directory then the disk has been marked.

SYSTEM. PASCAL	31	30-Aug-78	6	Codefile
----------------	----	-----------	---	----------

OR

<unused>	4		6	
SYSTEM. PASCAL	31	30-Aug-78	10	Codefile

Both of the above cases indicate disks that have not been marked. Below is the directory of a properly marked disk.

SYSTEM. PASCAL	31	30-Aug-78	10	Codefile
----------------	----	-----------	----	----------

To execute this program e(X)ecute MARKDUPDIR. The program will ask you which unit contains the disk to be marked (4 or 5). The program will check to see if it thinks that the blocks 6-9 are free. If the program doesn't think so it will ask you if you are sure they are free ? Typing 'Y' will execute the mark, any other character will abort the program. Be sure that the space is free before marking it as a duplicate directory.

* P-CODE DISASSEMBLER * * Section 4.9 *

Version I.5 September 1978

The disassembler inputs a standard UCSD code file and outputs symbolic psuedo-assembly (P-Code) along with various statistics concerning opcode frequency, procedure calls, and data segment references. The disassembler was originally written to collect statistics on opcode frequency, etc. as an aid in making architecture improvements. It has since been found helpful in debugging interpreters, optimizing programs, and provides a source of further information regarding some of subtleties of our implementation of Pascal. All statistics gathered are "static" as opposed to "dynamic". In other words the statistics are collected by making a pass through the code file instead of collecting them while the code file is actually running.

4.9.1 DISASSEMBLY

The Disassembler inputs a code file that has been generated by the UCSD Pascal Compiler. If a program USES a UNIT the disassembly will include the UNIT only if the code file has been linked. Assembly routines linked into a Pascal host will never be included in the disassembly.

The Disassembler is invoked by executing DISASM.I5 and requires the file OPCODES.I5 to be on the system disk. The Disassembler will first prompt for an input code file, the suffix .CODE being assumed and thus not required. The next question refers to the byte sex of the machine the code file is intended to run on, that is whether the first physical byte (byte 0) of a machine word is the most significant byte of the word. For more information, see section 3.6 BYTE-SWAPPING. For both currently supported CPU's, the PDP-11 and the BOBO families, physical byte 0 is the least significant byte. Next the prompt will be for an output file for the disassembled output. Since the output file is untyped, CONSOLE: or the PRINTER: (if it is on-line) may be used in preference to any other file. The final question at this stage is whether the user wishes to take control of the disassembly, i.e. decide which procedures are disassembled as opposed to all the procedures in the file.

The following question regards the collection of statistics on references to a particular Procedure's data segment. Should you decide to control the disassembly you will be warned that all statistics gathered are only gathered on those procedures which are disassembled. Next you will be taken into the Segment Guide. This level displays the segments you have by name and lets you decide on which one you are interested in. The Procedure Guide follows to let you decide on the particular procedure(s) that you wish to disassemble. Typing an "L" at this point will list the procedure(s) contained in this segment. A more complete description of this step

occurs in the next section. The Segment Guide may be re-entered by typing "G" in the Procedure Guide. Thus in this manner you may disassemble several procedures in several different segments without disassembling the entire file. The Segment Guide is exited by typing "Q".

```

1 1 1:D 0 (*$L CONSOLE:*)
2 1 1:D 1 PROGRAM DISASMDemo;
3 1 1:D 3 VAR I: INTEGER;
4 1 1:D 4 TOMORROW: BOOLEAN;
5 1 1:D 5 COMMENT: STRING;
6 1 1:C 0 BEGIN
7 1 1:C 0 I:=0;
8 1 1:C 5 TOMORROW:=FALSE;
9 1 1:C 8 REPEAT
10 1 1:C 8 I:=I+1;
11 1 1:C 13 WRITELN('Disassembly -- a step backwards...');
12 1 1:C 74 UNTIL TOMORROW;
13 1 1:C 77 END.

```

FIGURE 1 SAMPLE PASCAL PROGRAM

SEGMENT	PROC	BLOCK #	OFFSET#	OFFSET IN BLOCK=	HEX CODE
1	1	0(000):	BPT	7	D507
1	1	2(002):	SLDC	0	00
1	1	3(003):	SRO	3	AB03
1	1	5(005):	SLDC	0	00
1	1	6(006):	SRO	4	AB04
1	1	8(008):	SLDO	3	EA
1	1	9(009):	SLDC	1	01
1	1	10(00A):	ADI		B2
1	1	11(00B):	SRO	3	AB03
1	1	13(00D):	L0D	1	B60103
1	1	16(010):	LCA	42	'Disassembly -- a step backwards..
1	1	60(03C):	SLDC	0	00
1	1	61(03D):	CXP	WRITESTR	CD0013
1	1	64(040):	CSP	IDCHECK	9E00
1	1	66(042):	L0D	1	B60103
1	1	69(045):	CXP	WRITELN	CD0016
1	1	72(048):	CSP	IDCHECK	9E00
1	1	74(04A):	SLDO	4	EB
1	1	75(04B):	FJP	B	A1F6
1	1	77(04D):	RBP	0	C100

FIGURE 2 SAMPLE PROGRAM DISASSEMBLED

Figure 1 displays a sample Pascal program that has been listed during compilation. Figure 2 displays the disassembled code of the file generated by the compiler. The left 3 columns in figure 2 correspond to the 3 columns to the right of the line number in figure 1. They are segment number, procedure number, and offset within procedure, respectively. The offset is also given in hex in parentheses. A complete description of UCSD P-Code mnemonics is given in section 3.4. The actual code that exists in the file is given in hex in the rightmost column. The parameters to CXP's and CSP's are converted to the procedure name if it is a known system procedure or function. WRITESTR, WRITELN, and IDCHECK are some examples. The string operand for LCA is printed as a string as evidenced by the line with offset 16. Jumps have their operand(s) converted to an offset from the start of the procedure so that the offset may act as a label. Thus the B displayed in the operand field of the FJP at offset 75 really means a jump to the SLDO at offset B. This is also true of case jumps (XJP's). The block number and byte offset of the start of the procedure are given relative to the start of the code file. Thus this procedure starts at block 1, offset 0 of the code file. The segment dictionary resides in block 0 for all code files.

4.9.2 DATA SEGMENT REFERENCE STATISTICS

The fourth prompt the Disassembler provides is a question asking if you would like to keep track of all references to a particular procedure's data segment. The most common use of these statistics is in optimization of a given procedure's code file. By re-arranging the order of declaration of variables one may change the offset within a data segment that applies to a given variable. For p-machine architecture reasons the first 16 words offset into the data segment are the fastest and have optimized 1 byte instructions. Offsets from 17 to 127 result in instructions as least 2 bytes long, while references to greater than 127 require at least 3 bytes. By making the most frequently used variables have the smaller offsets one may save considerable code file space and possibly time during execution.

```

|Data Segment size:   45      Data references:   5      Lex level   0
|
|For segment DISASMDE Procedure # 1
|Offset(word)      Total      %
|      3            3        60.00
|      4            2        40.00
|
|
|

```

FIGURE 3 SAMPLE PROGRAM'S DATA SEGMENT STATISTICS

Figure 3 shows the data segment statistics for our sample program. Clearly there is little to be gained from optimizing such a small program but the general idea can still be presented. By using the compiled listing shown in figure 1 one can match offsets to variables as such:

variable	offset
I	3
TOMORROW	4
COMMENT	5

Now by using the figures in figure 3 one can see that offset 3 or the variable I occurs most frequently and thus deserves it's position. This same idea carried out on a large program may result in substantial size savings. Notice that offset 6 never occurs and thus is not included in the statistics in figure 3.

The prompt for the output file for these statistics occurs after the disassembly has been completed. If you elect to collect these statistics you will be taken into the Segment and Procedure Guides as described in the previous section except that the prompt requests the selection of a data segment on which to collect statistics. In the Procedure Guide, "L" gives a listing of all the procedures in the selected segment by number, lex level, and data segment size. After the selection of a data segment, processing continues, as described in the previous section, from the point after the data segment question.

4.9.3 OPCODE, PROCEDURE CALL, AND JUMP STATISTICS

These statistics are collected as an aid in optimizing the architecture of P-Code and although they are interesting to look at they are of no real use to the typical user. For this reason they will be described only superficially.

Each opcode is given with a complete breakdown of which bit was most significant for each operand on any given occurrence of the opcode. These are presented in terms of totals and percentages of the number of occurrences of the opcode. In addition a histogram of the opcode occurrence as a percentage of the total number of opcodes disassembled runs along the righthand margin. There is also a table of jumps in terms of the number of bits required to represent the distance of the jump for both positive and negative jumps. Finally there are counts of all procedure calls listed by segment and procedure number.

 * LIBRARY MAP UTILITY * * Section 4.10 *

Version I.5 September 1978

The program LIBMAP produces a map of a library (or code) file and lists the linker information maintained for each segment of the file.

The program first prompts for a library file name. As in the linker, this may be an asterisk to indicate "*SYSTEM.LIBRARY". Unlike the linker, however, the ".CODE" suffix may be suppressed by appending a period to the full file name.

Example

typing	references file
*	*SYSTEM.LIBRARY
FARKLE	:FARKLE.CODE
OLD.LIBRARY.	:OLD.LIBRARY

Typically, the map utility will be used to list library definitions but the option is available to include intra-library symbol references. Should this feature be desired, type a "Y" when queried for a reference list. A space (or carriage return) is considered a "N".

The user is now prompted for an output file name. (".TEXT" will be appended unless an extra period is used.) Several libraries may be mapped at the same time. To quit, type a carriage return when prompted for any file name.

A sample map follows

LIBRARY MAP FOR *SYSTEM.LIBRARY

```

S # 0:  MOVETO      separate procedure segment
        PSMATP      public ref
        PSYPOS      public ref
        MOVETO      separate proc   P #1
        PSXPOS      public ref
        GMOVETO     global addr     P #1,   I #0
        PSBUFP      public ref
        JUNK        private ref
        DRWLIN      global ref
        PSYPDS      public ref (2 times)
        LINETO      separate proc   P #2
        PSXPOS      public ref (2 times)
        GMOVETO     global ref
        GLINETO     global addr     P #2,   I #0
  
```

DRAWLINE separate proc P #3
 DRWLIN global addr P #3, I #0
 PSMATP public ref (2 times)
 CONCAT separate proc P #4

S # 1: PSGRAPHI library unit
 XROT constant value of 0
 MAXSTK constant value of 7
 MATSTK private ref (10 times)
 MOVETO external proc P #8
 LINETO external proc P #9
 CONCAT external proc P #12
 YROT constant value of 1
 ZROT constant value of 2
 PSXPOS public ref
 PSMATP public ref (7 times)
 PSYPOS public ref
 PSBUFP public ref (7 times)
 STKINX private ref (8 times)
 BUF1 private ref (4 times)
 BUF2 private ref (2 times)

S # 2: VPGRAPHI library unit
 NONE constant value of 0
 REVERSE constant value of 3
 SCREEN private ref (3 times)
 SCALE private ref (8 times)
 XCENTER private ref (2 times)
 YCENTER private ref (2 times)
 XCURR private ref (7 times)
 YCURR private ref (7 times)
 WHITE constant value of 5
 BLACK constant value of 6
 XHIVALUE private ref (4 times)
 YHIVALUE private ref (4 times)
 XLOVALUE private ref (5 times)
 YLOVALUE private ref (5 times)
 DRAW constant value of 1
 POINT constant value of 4
 ERASE constant value of 2
 DRAWLINE external proc P #8
 XSCREEN constant value of 320
 XSCALE private ref (3 times)
 XSHIFT private ref (2 times)
 YSCREEN constant value of 240
 YSCALE private ref (3 times)
 YSHIFT private ref (2 times)

S # 3: TURTLE library unit
 NONE constant value of 0
 WHITE constant value of 1
 REVERSE constant value of 3
 HEADING private ref (15 times)
 WANTCURS private ref (13 times)

SCALE private ref (8 times)
SCREEN private ref (3 times)
XCENTER private ref (2 times)
YCENTER private ref (2 times)
XCURR private ref (6 times)
YCURR private ref (6 times)
TOPEN private ref (4 times)
BLACK constant value of 2
XHIVALUE private ref (4 times)
YHIVALUE private ref (4 times)
XLDVALUE private ref (5 times)
YLDVALUE private ref (5 times)
XSCREEN constant value of 320
XSCALE private ref (3 times)
DRAWLINE external proc P #10
XSHIFT private ref (2 times)
YSCREEN constant value of 240
YSCALE private ref (3 times)
YSHIFT private ref (2 times)

S # 4: to S #15: are unused

* TABLE 1 * * EXECUTION ERRORS *

Version I.5 September 1978

0	System error	FATAL
1	Invalid index, value out of range (XINVNDX)	
2	No segment, bad code file (XNOPROC)	
3	Procedure not present at exit time (XNOEXIT)	
4	Stack overflow (XSTKOVR)	
5	Integer overflow (XINTOVR)	
6	Divide by zero (XDIVZER)	
7	Invalid memory reference <bus timed out> (XBADMEM)	
8	User break (XUBREAK)	
9	System I/O error (XSYIDER)	FATAL
10	User I/O error (XUIDERR)	
11	Unimplemented instruction (XNOTIMP)	
12	Floating point math error (XFPIERR)	
13	String too long (XS2LONG)	
14	Halt, Breakpoint (without debugger in core) (XHLTBPT)	
15	Bad Block	

All fatal errors either cause the system to rebootstrap, or if the error was totally lethal to the system, the user will have to reboot. All errors cause the system to re-initialize itself (call system procedure INITIALIZE).

* TABLE 2 * * IORESULTS *

Version I.5 September 1978

0	No error
1	Bad Block, Parity error (CRC)
2	Bad Unit Number
3	Bad Mode, Illegal operation
4	Undefined hardware error
5	Lost unit, Unit is no longer on-line
6	Lost file, File is no longer in directory
7	Bad Title, Illegal file name
8	No room, insufficient space
9	No unit, No such volume on line
10	No file, No such file on volume
11	Duplicate file
12	Not closed, attempt to open an open file
13	Not open, attempt to access a closed file
14	Bad format, error in reading real or integer
15	Ring buffer overflow

* TABLE 3 * * UNITNUMBERS *

Version 1.5 September 1978

NUMBER	VOLUME NAME
0	<empty>
1	CONSOLE
2	SYSTEM
3	GRAPHIC
4	floppy0
5	floppy1
6	PRINTER
7	available - <unimplemented>
8	REMOTE <reserved for future use>
9	block1
10	block2
11	block3
12	block4

Devices 9 - 12 are block-structured devices, in most cases (RK-05).

* TABLE 4 * * PENSTATES *

Version I.5 September 1978

DRAWLINE:

0	PENUP (picture will not change)
1	PENDOWN (force bits on)
2	ERASER (force bits off)
3	COMPLEMENT (XOR bits)
4	RADAR (scan for obstacle)

DRAWBLOCK:

0	OR <paint source onto destination>
1	COPY <source goes to destination>
2	COMPLEMENT <inverted source goes to destination>
3	EXCLUSIVE-OR <source exclusive-or destination goes to desination>

* TABLE 5 * * SYNTAX ERRORS IN UCSD PASCAL *

Version I.5 September 1978

The syntax errors this compiler gives are not the best it can do. When time comes available to do so, the error generation of the compiler is going to be seriously re-vamped.

- 1: Error in simple type
- 2: Identifier expected
- 3: 'PROGRAM' expected
- 4: ')' expected
- 5: ':' expected
- 6: Illegal symbol
- 7: Error in parameter list
- 8: 'OF' expected
- 9: '(' expected
- 10: Error in type
- 11: '[' expected
- 12: ']' expected
- 13: 'END' expected
- 14: 'i' expected
- 15: Integer expected
- 16: '=' expected
- 17: 'BEGIN' expected
- 18: Error in declaration part
- 19: error in <field-list>
- 20: '.' expected
- 21: '*' expected
- 22: 'Interface' expected
- 23: 'Implementation' expected
- 24: 'Unit' expected

- 50: Error in constant
- 51: ': =' expected
- 52: 'THEN' expected
- 53: 'UNTIL' expected
- 54: 'DO' expected
- 55: 'TO' or 'DOWNTD' expected in for statement
- 56: 'IF' expected
- 57: 'FILE' expected
- 58: Error in <factor> (bad expression)
- 59: Error in variable

- 101: Identifier declared twice
- 102: Low bound exceeds high bound
- 103: Identifier is not of the appropriate class
- 104: Undeclared identifier
- 105: sign not allowed
- 106: Number expected
- 107: Incompatible subrange types

108: File not allowed here
109: Type must not be real
110: <tagfield> type must be scalar or subrange
111: Incompatible with <tagfield> part
112: Index type must not be real
113: Index type must be a scalar or a subrange
114: Base type must not be real
115: Base type must be a scalar or a subrange
116: Error in type of standard procedure parameter
117: Unsatisfied forward reference
118: Forward reference type identifier in variable declaration
119: Re-specified params not OK for a forward declared procedure
120: Function result type must be scalar, subrange or pointer
121: File value parameter not allowed
122: A forward declared function's result type can't be re-specified
123: Missing result type in function declaration
124: F-format for reals only
125: Error in type of standard procedure parameter
126: Number of parameters does not agree with declaration
127: Illegal parameter substitution
128: Result type does not agree with declaration
129: Type conflict of operands
130: Expression is not of set type
131: Tests on equality allowed only
132: Strict inclusion not allowed
133: File comparison not allowed
134: Illegal type of operand(s)
135: Type of operand must be boolean
136: Set element type must be scalar or subrange
137: Set element types must be compatible
138: Type of variable is not array
139: Index type is not compatible with the declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter solution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be scalar
149: Index type must be integer
150: Assignment to standard function is not allowed

151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable cannot be formal or non-local
156: Multidefined case label
157: Too many cases in case statement
158: No such variant in this record
159: Real or string tagfields not allowed
160: Previous declaration was not forward

161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substitution of standard proc/func not allowed
165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
169: Error in base set
170: Value parameter expected
171: Standard file was re-declared
172: Undeclared external file
174: Pascal function or procedure expected
182: Nested units not allowed
183: External declaration not allowed at this nesting level
184: External declaration not allowed in interface section
185: Segment declaration not allowed in unit
186: Labels not allowed in interface section
187: Attempt to open library unsuccessful
188: Unit not declared in previous uses declaration
189: 'Uses' not allowed at this nesting level
190: Unit not in library
191: No private files
192: 'Uses' must be in interface section
193: Not enough room for this operation
194: Comment must appear at top of program
195: Unit not importable

201: Error in real number - digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range
204: 8 or 9 in octal number
250: Too many scopes of nested identifiers
251: Too many nested procedures or functions
252: Too many forward references of procedure entries
253: Procedure too long
254: Too many long constants in this procedure
256: Too many external references
257: Too many externals
258: Too many local files
259: Expression too complicated

300: Division by zero
301: No case provided for this value
302: Index expression out of bounds
303: Value to be assigned is out of bounds
304: Element expression out of range
398: Implementation restriction
399: Implementation restriction

400: Illegal character in text
401: Unexpected end of input
402: Error in writing code file, not enough room
403: Error in reading include file
404: Error in writing list file, not enough room
405: Call not allowed in separate procedure
406: Include file not legal

* TABLE 6 * * ASSEMBLER SYNTAX ERRORS *

Version I.5 September 1978

This section lists all the general errors found in the ERRORS file, specific machine errors are found in the sections below dealing with machine specifics.

- 1: Undefined label
- 2: Operand out of range
- 3: Must have procedure name
- 4: Number of parameters expected
- 5: Extra garbage on line
- 6: Input line over 80 characters
- 7: Not enough ifs
- 8: Must be declared in ASECT before use
- 9: Identifier previously declared
- 10: Improper format
- 11: EQU expected
- 12: Must EQU before use if not to a label
- 13: Macro identifier expected
- 14: Word addressed machine
- 15: Backward ORG not allowed
- 16: Identifier expected
- 17: Constant expected
- 18: Invalid structure
- 19: Extra special symbol
- 20: Branch too far
- 21: Variable not PC relative
- 22: Illegal macro parameter index
- 23: Not enough macro parameters
- 24: Operand not absolute
- 25: Illegal use of special symbols
- 26: Ill-formed expression
- 27: Not enough operands
- 28: Cannot handle this relative
- 29: Constant overflow
- 30: Illegal decimal constant
- 31: Illegal octal constant
- 32: Illegal binary constant
- 33: Invalid key word
- 34: Unexpected end of input - after macro
- 35: Include files must not be nested
- 36: Unexpected end of input
- 37: Bad place for an include file
- 38: Only labels & comments may occupy column one
- 39: Expected local label
- 40: Local label stack overflow
- 41: String constant must be on 1 line
- 42: String constant exceeds 80 chars
- 43: Illegal use of macro parameter
- 44: No local labels in ASECT
- 45: Expected key word

- 46: String expected
- 47: Bad block, parity error (crc)
- 48: Bad unit number
- 49: Bad mode, illegal operation
- 50: Undefined hardware error
- 51: Lost unit, no longer on-line
- 52: Lost file, no longer in directory
- 53: Bad title, illegal file name
- 54: No room, insufficient space
- 55: No unit, no such volume on-line
- 56: No file, no such file on volume
- 57: Duplicate file
- 58: Not closed, attempt to open an open file
- 59: Not open, attempt to access a closed file
- 60: Bad format, error in reading real or integer
- 61 *Nested macro definitions illegal*
- 62 *"=" or "f" expected*
- 63 *May not EQU to undefined labels*

Z80 Based machines

For constants, Hex is the default type,
 a 'B' defines binary ex. 10010B ,
 a '.' defines decimal ex. 5674. .

Location Counter (LC) = \$

All reserved words may not be used for any other purpose such as an identifier. For example, the reserved word "C" currently is being used as a register and in a condition code, therefore it may not be used for any other purpose (this is contrary to usual Zilog assembly language, but is restricted in the UCSD assembler).

Specific error messages:

- 76: Incorrect operand format
- 77: Close paren ")" expected
- 78: Comma "," expected
- 79: Plus "+" expected
- 80: Open paren "(" expected
- 81: Stack pointer "SP" expected
- 82: "HL" expected
- 83: Illegal "CC" condition code
- 84: Register "C" expected
- 85: Register "R" expected
- 86: Register "A" expected

PDP11 Based machines:

For constants, Octal is the default type for both input
and output.
a 'H' defines hexadecimal ex. 056H ,
a '.' defines decimal ex. 546. ,
a 'B' defines binary ex. 1001B .

Location Counter (LC) = *

Specific error messages:

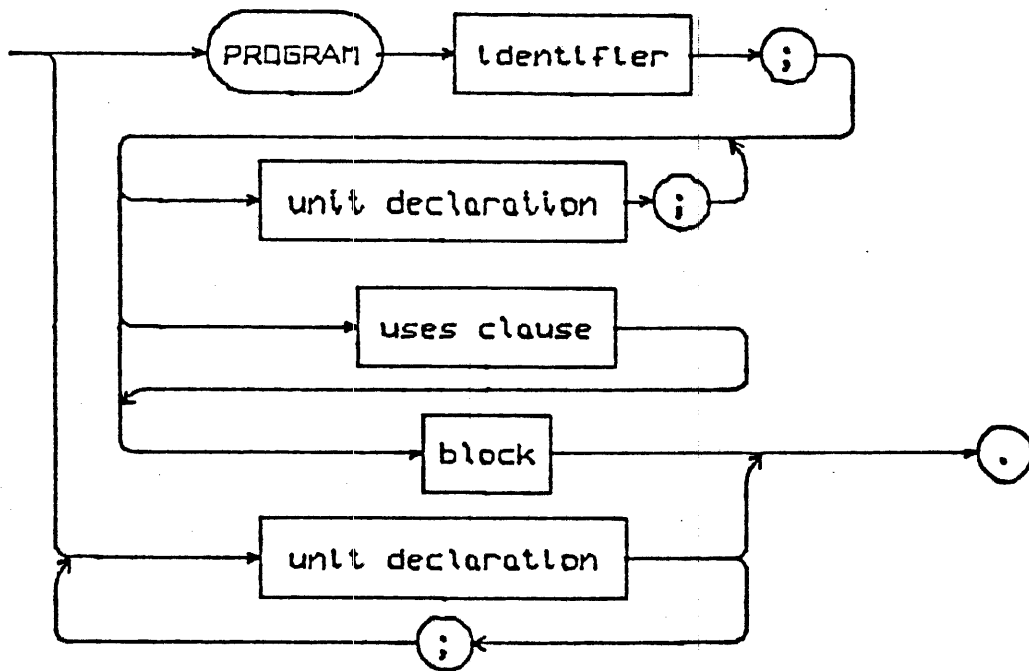
76: Closing paren ")" expected
77: Register expected
78: Too many special symbols
79: Unrecognizable operand
80: Register reference only
81: First operand must be a register
82: Comma expected
83: Unimplimented instruction
84: Must branch backwards to label

 * TABLE 7 * * American Standard Code for Information Interchange *

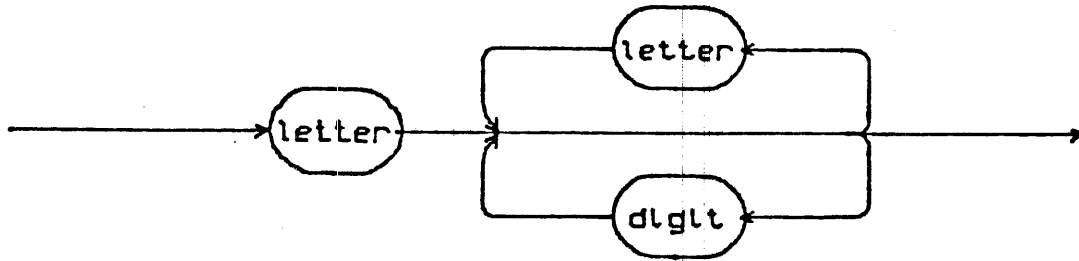
Version I.5 September 1978

0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	`
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EDT	36	044	24	\$	68	104	44	D	100	144	64	d
5	005	05	ENG	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	
29	035	1D	GS	61	075	3D	=	93	135	5D]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

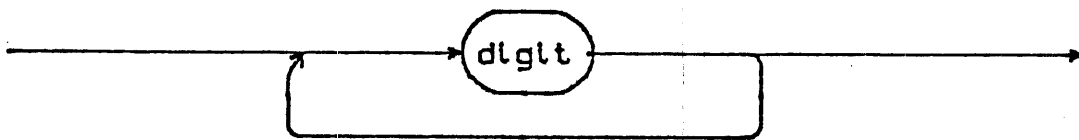
<compilation>



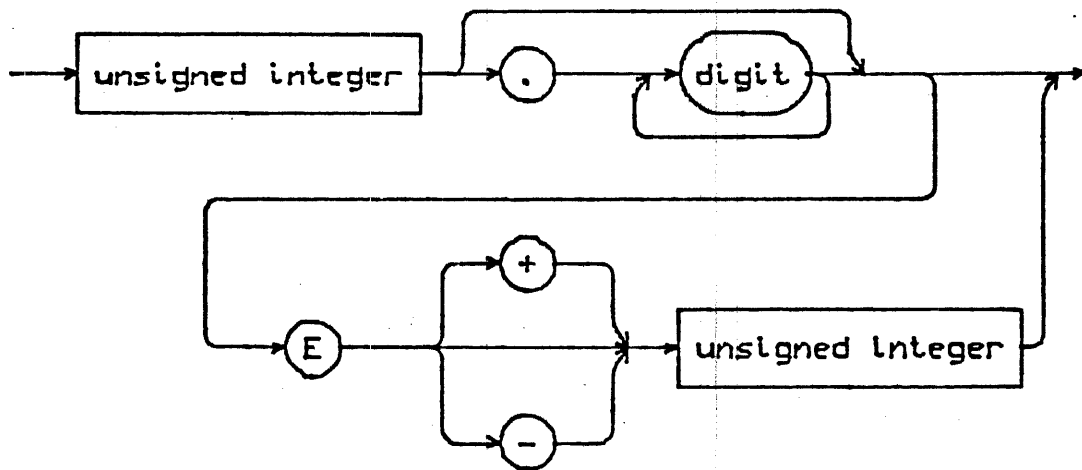
<Identifier>



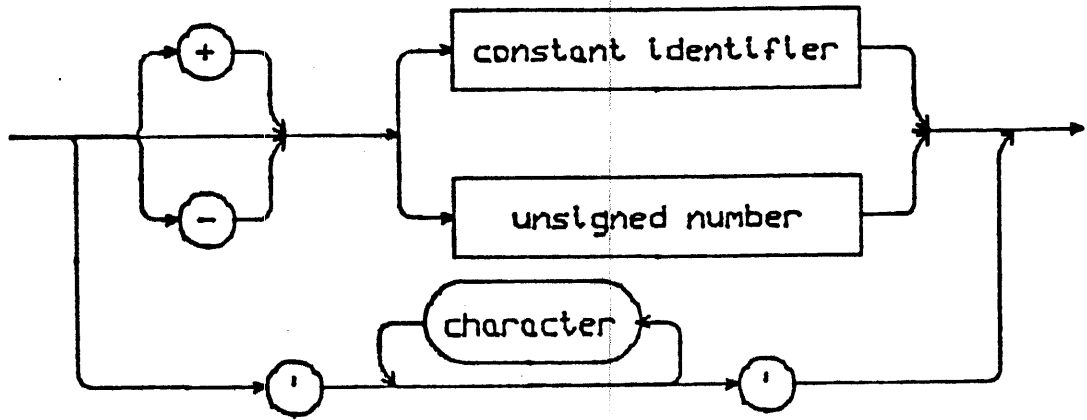
<unsigned integer>



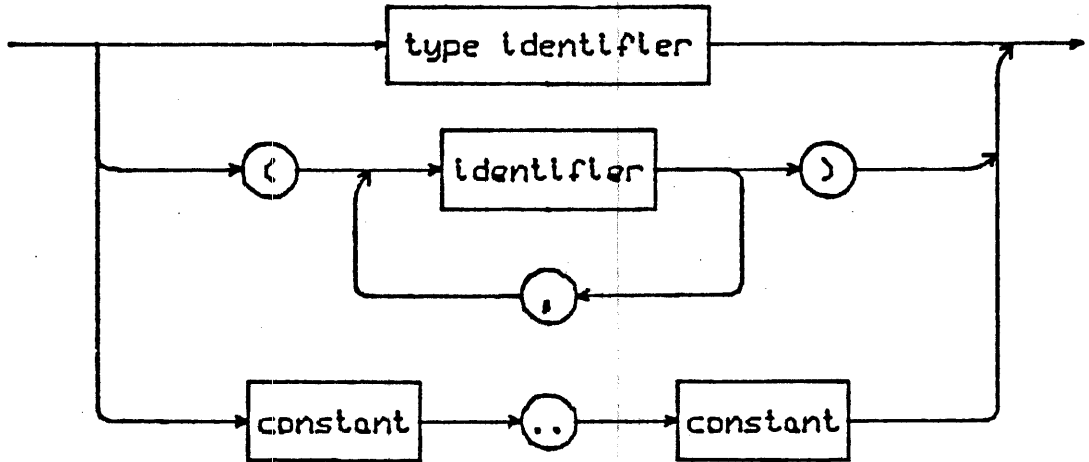
<unsigned number>



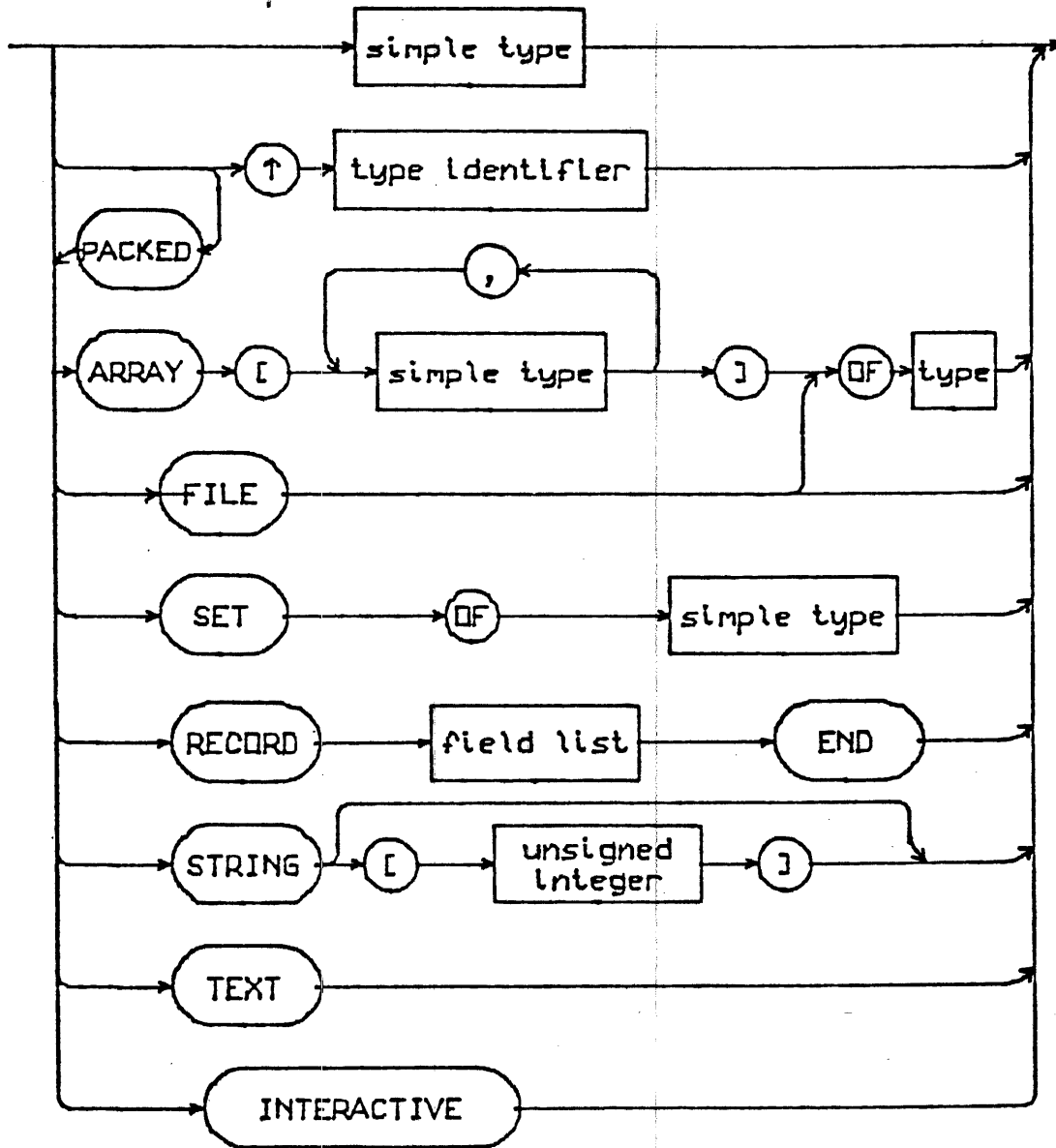
<constant>



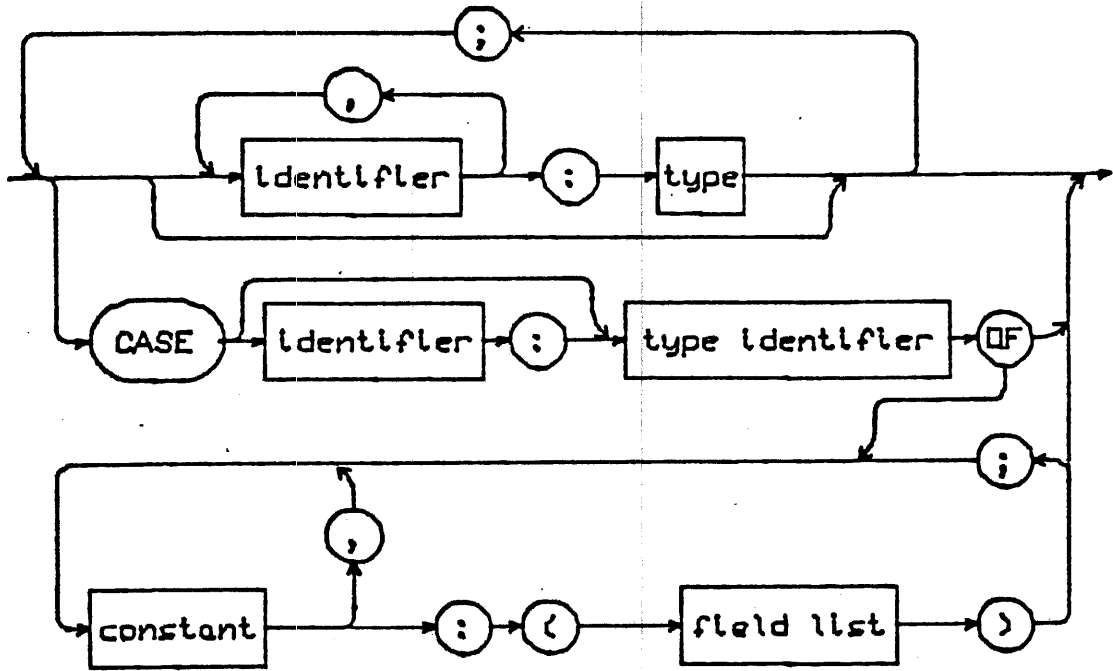
<simple type>



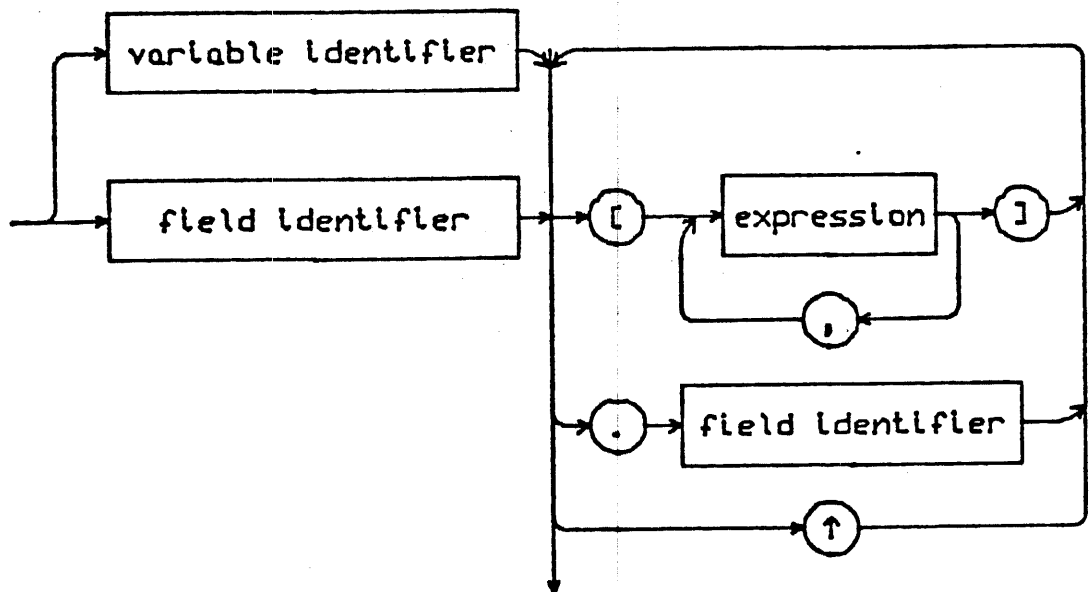
<type>



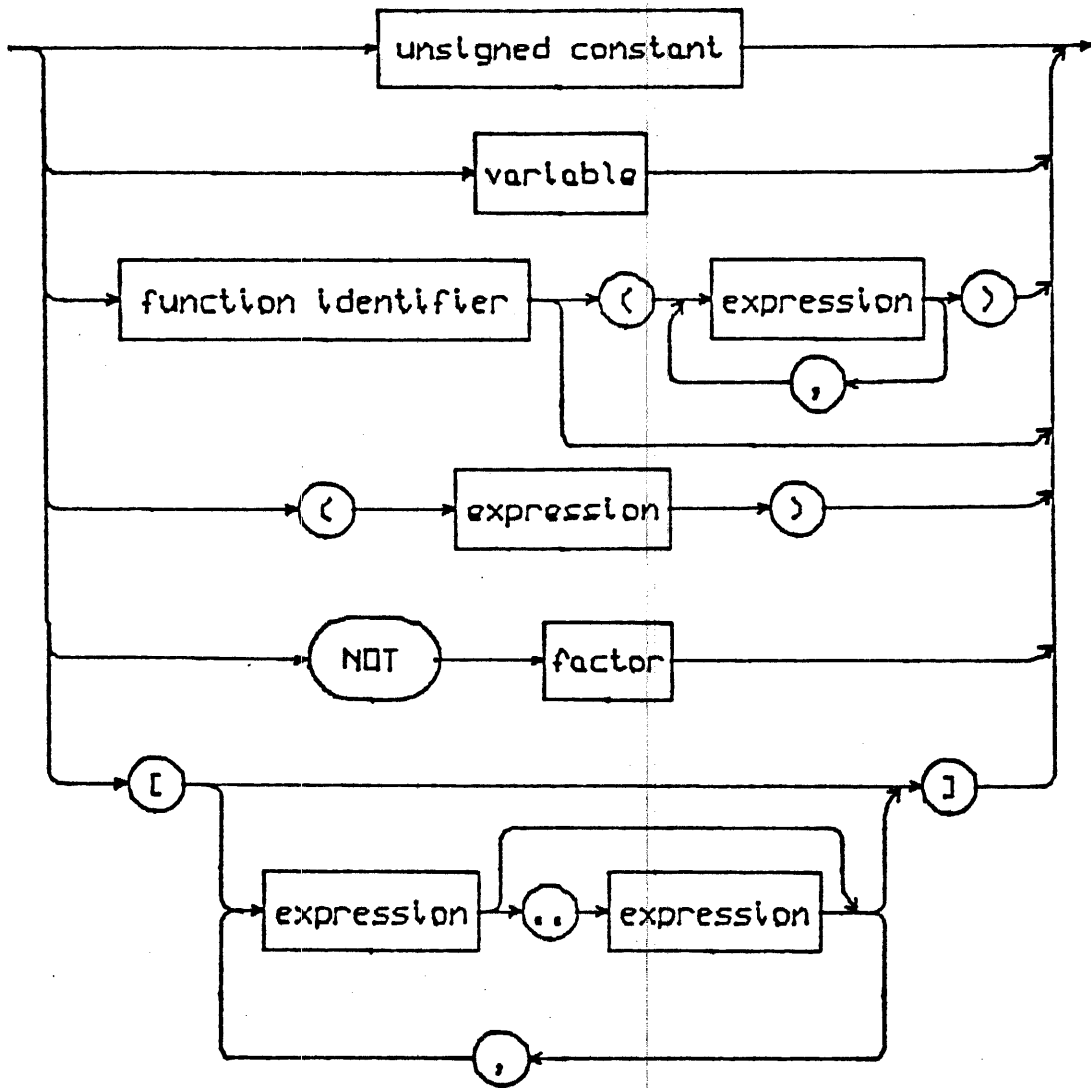
<field list>



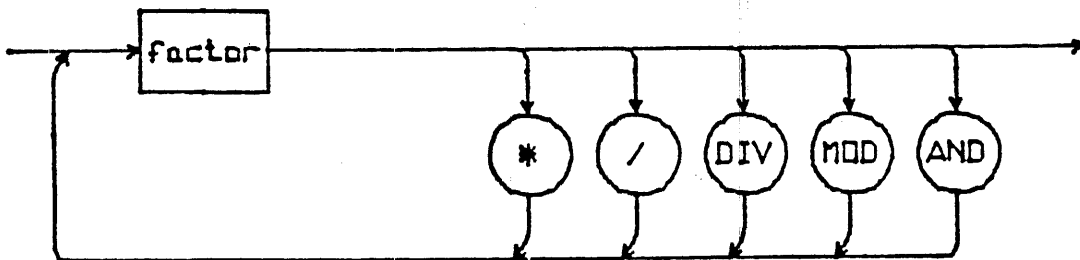
<variable>



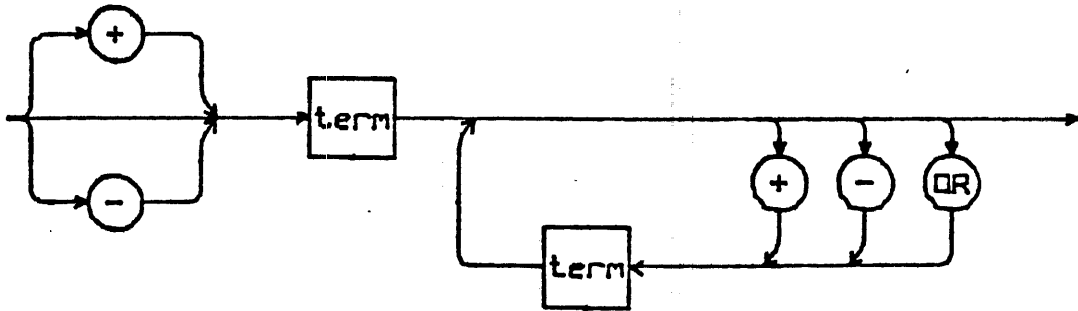
<factor>



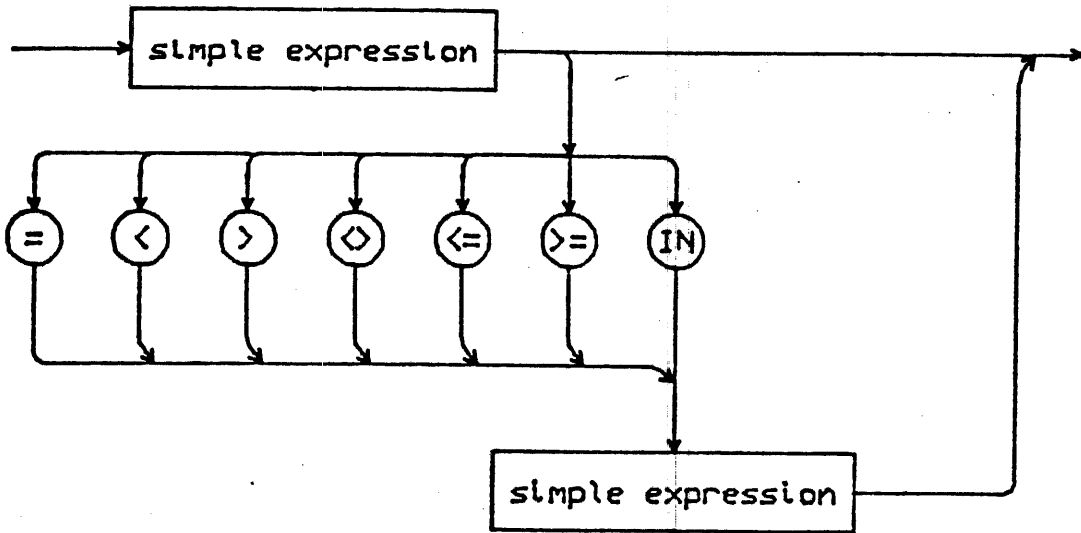
<term>



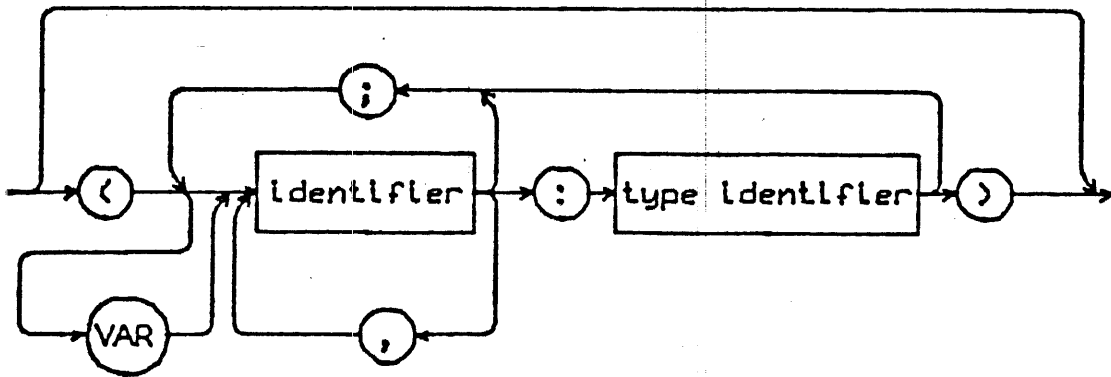
<simple expression>

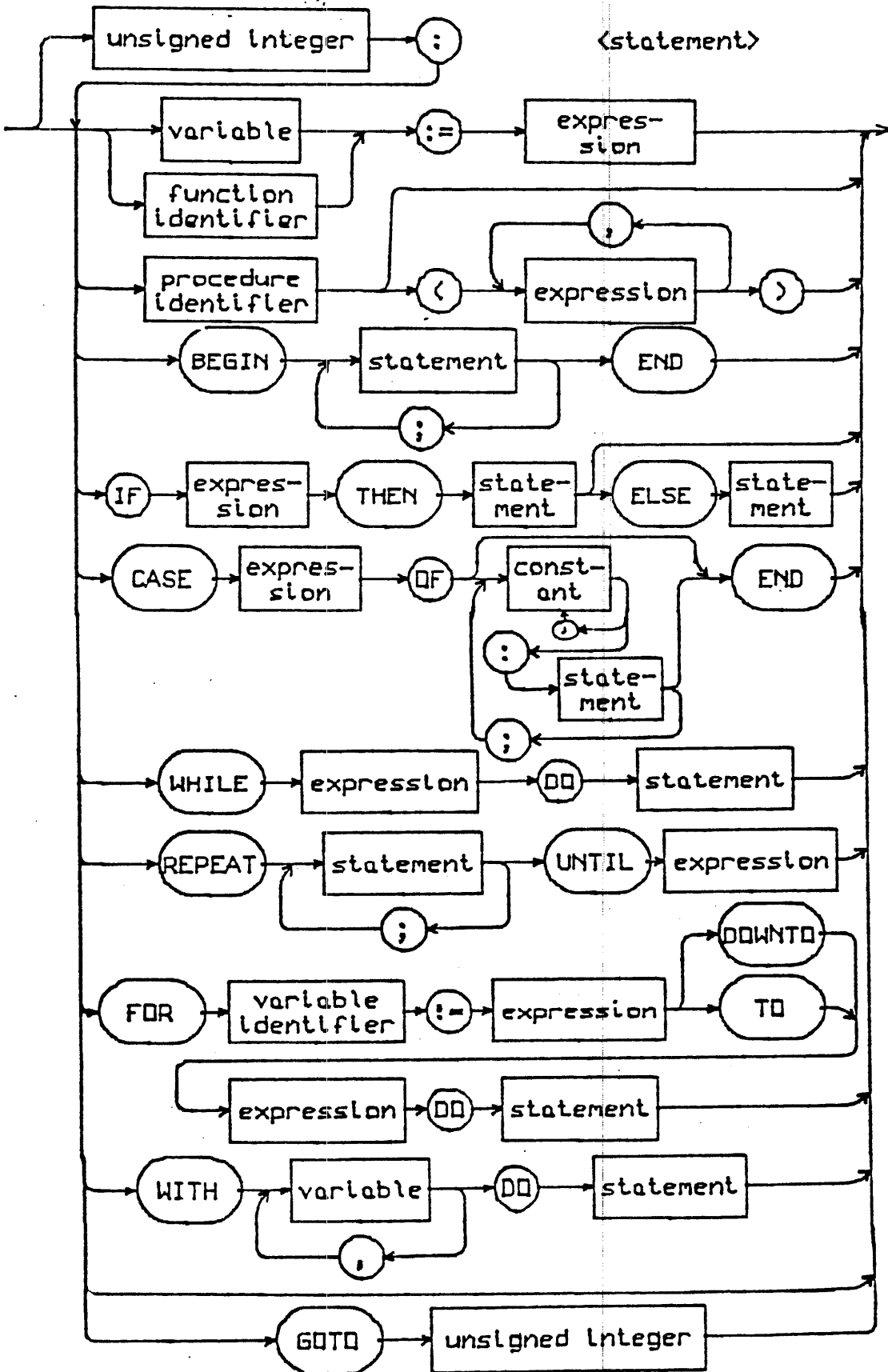


<expression>

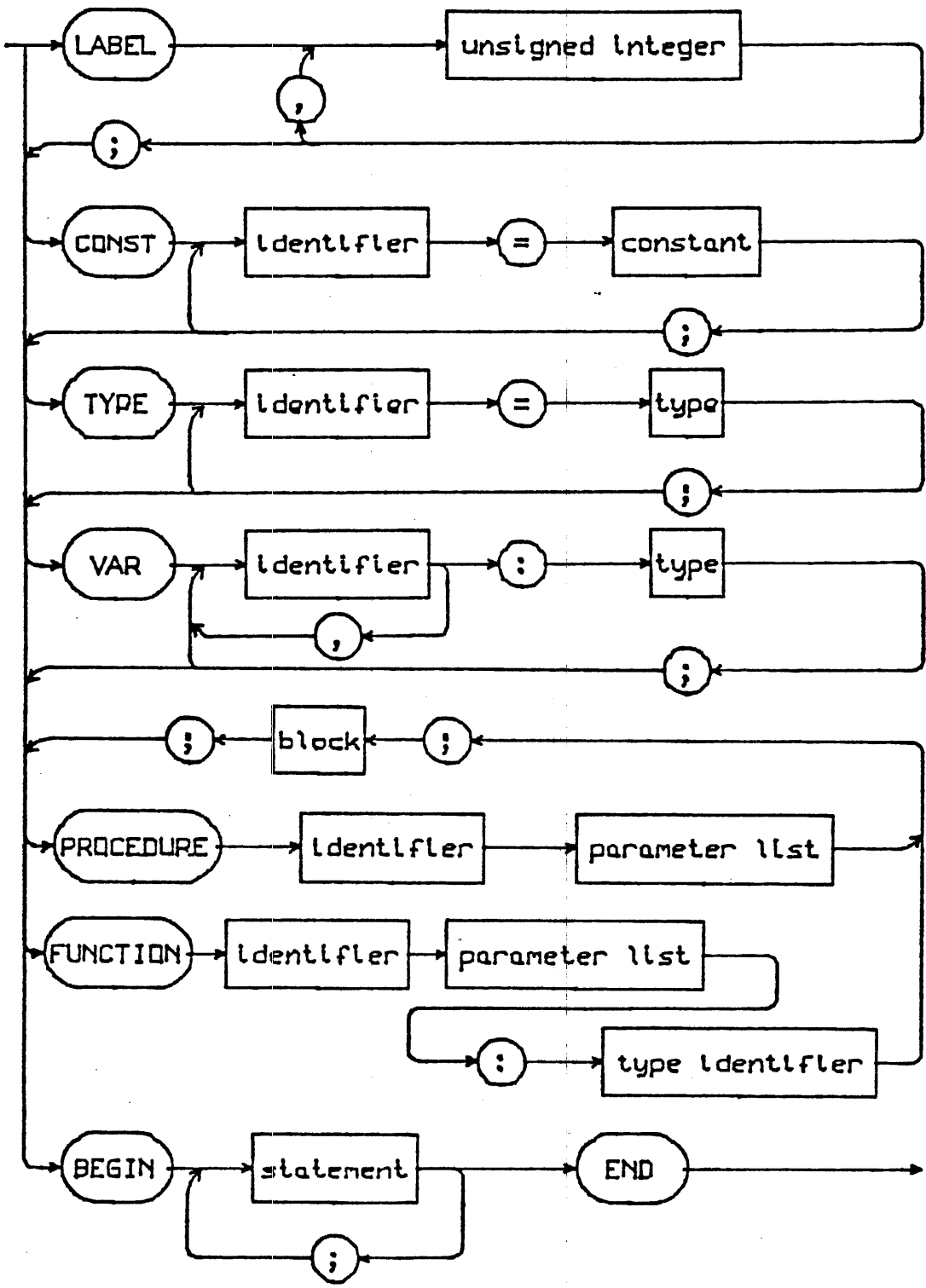


<parameter list>





<block>



* MATERIALS AVAILABLE * * Section A.1 *

As the UCSD Pascal system has grown, we have found that to distribute all of the software which is useful to all users for all systems, has become an unbearable task. To attempt to alleviate the large number of diskettes the release software requires, and to alleviate the number of pages of documentation sent to each subscriber, we have started to split the system into a number of seperately available sections.

The major section is the section which contains the operating system and all the support routines that go with it. We include a number of useful utilities which should enable the subscriber to do all types of developmental work. The master release (as from herein it shall be named) contains the interpreter for the initial system ordered, the UCSD Pascal operating system, the Pascal compiler, two text editors (one for screen devices, one for general purpose), a BASIC compiler, the Linker, the Assembler for the appropriate machine (at least). Other utilities include: a generalized file utility (the File handler), a generalize patch and dump routine, a set of programs to enable the subscriber to configure the system to run most intelligently with any terminal, a desk calculator, and a librarian.

Software which is not included in the master release is generally available from the IIS as a supplemental package at a nominal handling charge (dependent on the amount of material involved with the package). The sorts of software available are: interpreters for machines other than the machine the master release was ordered for, which will be accompanied by the assembler for that machine, in some cases we have assemblers for machines for which we do not yet have interpreters, program and data management systems, specifically a cross-referencer, and a pretty-printer. Also available, although not until some indeterminate time after the I.3 release, a Computer Aided Instruction packet. This may be available through the IIS, however it may be available only through the University of California Extension Studies Office. The CAI package consists of knowledge quizzes, and programming quizzes, and a record keeping system, all based on Kenneth L. Bowles book: (Micro)Computer Problem Solving Using Pascal.

* THE FIRST TIME THROUGH * * Section A.2.1 *

Version I.5 September 1978

Welcome to UCSD PASCAL. If you put the disk labelled "PASCAL:" in your booting drive, went through your normal boot-strapping procedure, and were greeted in a similar fashion, you do not need to read this section.

If this is not the case then here are a few of the problems we have encountered with I.4 coming up in strange and foreign lands:

- 1.) Some revisions of the LSI-11 refuse to boot with the clock running. If you have a switchable clock, turn it off to bootstrap; if and when the system greets you with the welcome message and the date, turn the clock back on.
- 2.) You have Andromeda floppy-disk drives. Currently you will be able to use only drive #0 unless the other drives have disks in them at bootstrap time. Drives that do not meet this condition will appear permanently off-line.
- 3.) You do not have enough memory. The minimum requirement for memory is 24K 16-bit words.
- 4.) You have a system configured for RK-05 hard-disk and you have an unformatted disk on line. The system will hang waiting for a reply from the disk which cannot be generated if the disk is unformatted. Take the disk off-line and try again.
- 5.) You have a system configured for RK and RX and the RX is not present. RX must be present.
- 6.) We haven't encountered your problem before. Call:

The number listed on the front page of this document.

* BOBO/ZBO WITH CP/M & 3740 DISKS * * Section A.2.2 *

Version 1.5 September 1978

THE CP/M IMPLEMENTATION OF UCSD PASCAL

BOOTING PASCAL

To get Pascal running under your version of CP/M, a two-disk bootstrap is used. First, boot CP/M in the usual manner. On the CP/M disk distributed with the Pascal system is a file called PASCAL.COM. PIP this file over to the booted disk, then execute it.

When the program asks for a Pascal disk, put the disk labeled PASCAL: in drive A and any disk in drive B. The system may not boot if there is no disk in drive B, or if you have a 1-drive system and your CP/M drivers wait on a request to drive B. Then hit [return]. In about 15 seconds the Pascal welcoming message should appear. (Note: we have discovered that some drives, possibly as a result of being double-buffered, cannot keep up with a 2 to 1 interleaving and hence are extremely slow. The bootstrap then may take about 30 or 40 seconds. We intend to alleviate this problem in the next release, but persons with such drives will have to bear with slow disk accesses for the present.)

If all has gone well, Welcome to the Wonderful World of Pascal. If not, please call to notify us of your problem.

MODIFICATIONS TO CP/M

The Pascal system will operate under an unmodified CP/M system, but it is advisable to create a special CP/M for use with Pascal in order to have Pascal running in the environment for which it was designed.

1. If there is no disk in a drive and an access is made from that disk, the driver should not wait to perform that access until a disk is inserted, as the Pascal system often attempts to read from empty drives when searching for a particular disk. Instead, simply return a 1 to indicate a bad I/O operation.
2. If you have a keyboard interrupt handler, it should recognize the character [ctrl-f] as a "flush-output" toggle and signal the character-out routine to gobble any characters until signaled again. When it receives another [ctrl-f] the keyboard handler should signal the output handler causing the output handler to resume outputting characters sent to it.

The keyboard interrupt handler should also recognize the character [cntrl-s] as a "stop output" toggle and wait until it receives another [cntrl-s] before allowing program execution to continue.

If your keyboard has no alphalock, the input driver can use any character not used for some other purpose as an alphalock toggle. [Cntrl-p], [return], [cntrl-i], [cntrl-s], [cntrl-f], [cntrl-c] or any character in SYSCOM^.CRTINFO should be excluded from consideration. We suggest [cntrl-a].

Pascal expects the tab character ([cntrl-i]) to cause the terminal cursor to advance to the nearest eight column. If the terminal does not do this itself, then the driver in the BIOS should.

CREATING A BOOTSTRAP ON A PASCAL DISK

Note: These instructions are for a standard BIOS with 512-byte blocks. For instructions for a non-standard BIOS, reference file READ.ME on the CP/M disk in the distribution packet.

On the CP/M disk are two programs, PGEN.COM and PINIT.ASM. The program PGEN.COM is a program used to write out a buffer (which will be filled by boot code and BIOS) to track 0. PINIT.ASM is the boot code that reads SYSTEM.MICRO from a Pascal disk, loads the BIOS into the correct place, and starts the interpreter's boot routine.

You must create a file PBOOT.HEX, which will require a slight modification of your current BOOT program. PBOOT will reside on track 0, sector 1 and, when executed, will load track 0, sectors 2 thru 13 into memory starting at location (MSIZE-4B)*1024 + 0BA00H, and jump to that location.

You then need to edit PINIT.ASM, changing MSIZE to match your system. Assemble the file, creating PINIT.HEX.

The next step is to stitch together the one-sector boot, the Pascal interpreter loader, BIOS, and the program to write this information out to sector 0. The following is a session with DDT that performs all this. This session was used to create a 48K system. User input is in lowercase, and comments are off to the right.

```
A>ddt pgen.com          ; load PGEN.COM into memory. PBOOT, PINIT,
                        ; and BIOS will be overlayed into PGEN's
                        ; data area, after which a memory image will
                        ; be saved.
```

```
DDT VERS 1.3
NEXT PC
0400 0100
```

```

-ipboot48.hex      ; set PBOOT48.HEX as input file
-h900 0            ; PBOOT starts at location 0, and we want to
                  ; read it in at location 900H

0900 0900
-r900             ; read in PBOOT
NEXT PC
0980 0000

-ipinit48.hex     ; set 'PINIT48.HEX' as input file
-h980 BA00        ; PINIT starts at location BA00H in a 48K system
                  ; (in general (MSIZE-4B)*1024 + BA00H), and we
                  ; want it at location 980H

C380 4FB0
-r4f80           ; read it in
NEXT PC
0A7d BA00

-ibios48.hex     ; and lastly read BIDS into location DB0H
-hd80 be00
C380 4FB0
-r4f80
NEXT PC
0F76 0000
-[cntrl-c]      ; leave DDT...

A>save 16 pgen48.com ; ...and save the program.

A>pgen48         ; sample execution of the program...

PGEN VI. 0

PUT BOOTER?(Y/N)y

WRITING BOOTER TO DRIVE A, TYPE RETURN ; put a Pascal disk (preferably a
; copy of the master) in drive A
; before hitting [return].

AGAIN?(Y/N)n
GET BOOTER?(Y/N)n
REBOOTING CP/M, TYPE RETURN           ; put the CP/M disk back in drive A
; before hitting [return].

A>

```

* DIFFERENCES AMONG IMPLEMENTATIONS * * Section A.3 *

Version 1.5 September 1978

The following is a list of differences between PDP11 Pascal and BOBO/Z80 Pascal, the items describe the way it is on the BOBO/Z80, and how that differs from the documented system.

1. The definition of div is different (thereby changing the values returned by mod):

$a \text{ div } b = \text{floor}(a/b)$
 $a \text{ mod } b = a - b*(a \text{ div } b)$

2. The I/O drivers are all written for synchronous operation. This means that [break] has no effect. [Ctrl-s] and [ctrl-f] will not perform as described unless you have a keyboard interrupt handler, and this handler is modified as specified below in Modifications to CPM.

This also means that UNITBUSY, UNITCLEAR, and UNITWAIT are meaningless. (In the future it may be possible to use the UNITBUSY and UNITCLEAR operations on the keyboard, but this is currently infeasible.)

3. The interpreter is called SYSTEM.MICRO instead of SYSTEM.INTERP.
4. The CP/M implementations have bootstraps that are not accessible to Pascal, hence the program BOOTER.CODE will not work. See the appropriate section of this document for instructions on copying and/or creating a bootstrap.
5. There are no turtle graphics procedures in the interpreter. Users with bit-mapped graphics devices are advised to see section 3.1 of the documentation for a Pascal version of DRAWLINE.
6. There are no long integer functions available with the Z80/BOBO system. They will be available in later releases.

* CHANGES MADE IN RECENT RELEASES * * Section A.4 *

Version I.5 September 1978

SUMMARY OF DIFFERENCES BETWEEN UCSD PASCAL RELEASES I.4 AND I.5

The following additions, improvements and/or corrections apply to Version I.5. Reference the (section #) preceding each entry for a more detailed description. For information regarding differences between previous releases refer to the system documentation for those releases.

(1.1)
OPERATING SYSTEM

- (—) All fields of SYSCOM (system communication area) that can be set in the utility SETUP are initialized at boot time using *SYSTEM.MISCINFO (if present).
- (2.1.1) The bug in the string intrinsic POS has been fixed.
- (1.1) C(ompile will now prompt the user for the file to compile if the workfile is empty.
- (1.8) There now exists a new command called L(link at the command level of the system that directly invokes the new utility *SYSTEM.LINKER.
- (1.9) There now exists a new command called A(ssem at the command level of the system that directly invokes the new assembler.
- (1.1) If a file SYSTEM.STARTUP exists on a given disk, that file will be run as a user program at initialize time.
- (1.1) R(un directly invokes *SYSTEM.LINKER if it is needed by the user program. It assume use of *SYSTEM.LIBRARY for external linkage.
- (1.1) X(ecute will not run code files which need to be L(inked. An error message will appear.
- (1.2) The file handler is now a separate file called *SYSTEM.FILER.

(---) Backspacing and are now allowed when reading integers from Unit #1 (CONSOLE:). However, backspacing over the sign, if any, is not permitted.

(1.2) FILE HANDLER

Substantial modifications have been made in the syntax of user responses to filer prompts. For nearly all commands there exists the option of using either of two wildcard symbols enabling extended control over activity within the filer. In general, the symbol "=" will allow selective control over files within the L(dir, C(hange, R(emove, and T(ransfer commands. The "?" symbol is similar to "=" with the addition that it will cause the filer to prompt the user for each task to be performed.

Q(et command now allows use of appended ".TEXT" and ".CODE" suffixes in file names and ignores them.

S(ave command will now allow the current workfile to be saved on a disk other than the system volume.

E(and L(dir now require an appended ":" after literal volume I.D.'s. Selective listing of directory subsets is allowed through use of the wildcard symbol = in conjunction with file prefix and suffix string patterns. Directory listings may be sent to a volume other than CONSOLE: by following the source volume name with ', <volume id>'.
'

C(hange command will now allow the user to change selected file prefix and suffix string patterns within groups of filenames containing the chosen patterns through use of the wildcard symbols = or ?.

R(emove command allows selective removal of groups of files using the = or ? symbol in a manner similar to the C(hange command. To selectively remove any or all of the files on a given volume the user may type <vol.prefix> ? and will be prompted for each file on the disk. Typing R(emove) <vol.prefix> now will result in no action. R(emove) <vol.prefix> = will remove ALL files on the disk. All commands resulting in the potential removal of more than one file will prompt the user with "Update directory?" following "removal" of file names.

T(ransfer command functions in a manner similar to the C(hange command. When performing a disk to disk transfer using one drive it will now ask for the file name to be transferred to before the source disk is removed. It is now possible to selectively transfer any or all of the files on a disk by typing

<vol.prefix> followed by "?" or "=" in a manner similar to the R(emove command. The user will be prompted for each file and is given the option of transferring.

Z(ero command will now prompt the user with the present number of blocks allocated the the disk in the directory, if a valid number exists, and will ask if the same number of blocks is wanted. If the response is No (or there was no previous "# of blocks") then the user may enter the appropriate number of blocks. The Z(ero will be aborted if a bad # of blocks is specified.

N(ew command will now check for a ".BACK" file corresponding to the current workfile and will ask if the user wishes this file to be removed. (This is for use in conjunction with the new L.2 (large file) EDITOR.

The new command ? will result in display of the prompt-line extension:

File: B(ad-blks, E(xt-dir, K(runch, M(ake, P(refix, V(olume, X(amine, Z(ero

Typing any non-command key will redisplay main promptline.

EDITORS (Sections 1.3 and 1.4)

Three different editors are currently provided with the UCSD PASCAL system: YALOE, "EDITOR"(E.6), and the new L.2 EDITOR. EDITOR is a substantially more powerful (and even easier to use) editor than YALOE, but it makes some assumptions about the run-time environment. The L.2 EDITOR (eventually to become the standard release editor) will handle files of arbitrary size, however it is in its experimental form and recommended for brave users only.

EDITOR requires a reasonably powerful CRT terminal with the following features:

- XYADRESSING - go directly to a given row and column on the screen
- NDFS - non-destructive forward space (the inverse of back-space)
- LF - down one line (and if at the bottom of the screen scrolls up)
- RLF - reverse line feed (up one line; not required to reverse scroll)

(EDITOR no longer requires Erase-to-end-of-screen,
Erase-to-end-of-line, or Home facilities.)

Typing "E" at the main command level will execute the file SYSTEM.EDITOR. Selection of either VALUE or EDITOR(E.6 or L.2) as the system editor is made in the Filer by Changing the selected file's name to SYSTEM.EDITOR.

Proper use of EDITOR requires that the system disk be left on-line while editing.

The E.6 EDITOR has the following differences from the previously released E.4 EDITOR:

- (1.3.3) The C(opy command now requires the user to specify whether the copy is to be made from the B(uffer (as in the old C(opy command) or from another F(ile. Copying from a file allows the option of copying subsets of the file by specifying markers.
- (1.3.3) A(djust now enables L(eft and R(ight justification as well as C(entering of text lines.
- (---) Automatic date-stamping of files. The first date the file was created and the last date that it was updated are displayed in the E(nvironment.

The following is a brief summary of the differences between the E.6 editor and the L.2 (large file) editor (for more information see section 1.3.5):

- (1.3.5) The L.2 EDITOR does not write to SYSTEM.WRK.TEXT unless a new workfile has been created. Instead, upon entering the editor the file to be read from is renamed with a .BACK suffix and a workfile is created with the old file's name.
- (1.3.5) New commands to be used in conjunction with large file capability are B(anish L(eft or R(ight, and N(ext B(ack or F(oward or S(tart or E(nd.
- (1.3.5) F(ind and R(eplace will prompt user if target not found and the file extends beyond the editor buffer(i.e., if it is a "large file").
- (1.3.5) Changes within E(nvironment:
 - Ability to set tab stops.
 - Lists names of markers.
 - Lists number of pages in Left and Right stacks of large files, in buffer and number of pages available on disk.

(1.5)

DEBUGGER

(1.5) The debugger now works as claimed in the system documentation.

PASCAL COMPILER

- (2.2) Lowercase characters are now allowed within all identifiers and reserved words, but are converted to upper case (i. e., Hello is equivalent to HELLO). The break character '_' is also allowed (anywhere a digit is allowed in an identifier) and is ignored.
- (3.3.3) There now exists the facility for using "Long Integers" for business applications. The standard type INTEGER has been extended and the standard arithmetic operators +, -, *, DIV, and unary plus and minus are allowed for use with long integers (as well as the TRUNC and STR intrinsics).
- (3.3.2) A substantial new addition to capabilities of programming in UCSD PASCAL is the facility for linkage to separately compiled "UNIT's" and external assembly language routines. A UNIT is a library module which may be imported for use by PASCAL programs. It incorporates the use of public and private declarations and definitions. The introduction of UNITS to UCSD PASCAL introduces new syntax for the language including the new reserved words:
- UNIT
 - INTERFACE
 - IMPLEMENTATION
- and USES.
- (3.3.2) PASCAL programs may now access external assembly language routines through the use of an EXTERNAL declaration which resembles the FORWARD declaration.

(1.9) LINKER

SYSTEM.LINKER is a new system utility made available to allow the linkage of separately compiled PASCAL UNITS as well as access in PASCAL to assembly language routines, and linkage from assembly language to assembly language.

(4.2) LIBRARIES

The file SYSTEM.LIBRARY is available for use in conjunction with SYSTEM.LINKER. The old LINKER.CODE has been replaced by LIBRARY.CODE which allows the user to build libraries containing utility routines.

(2. 1. 1)
INTRINSICS

The procedure STR has been added and is used to convert integers or long integers to their character string representation.

UTILITY PROGRAMS

Several new UTILITY PROGRAMS have been added. Reference also the TABLE OF CONTENTS and the UTILITY DOCUMENT(Section 4).

- (4. 3) NEW SETUP.
- (4. 5) REVISED PATCH.
- (4. 8) COPYDUPDIR.
- (4. 8) MARKDUPDIR.
- (1. 9) ASSEMBLERS. (LSI-11, 8080, Z80)
- (4. 9) DISASSEMBLER.

* INDEX * * Section B *

Version I.5 September 1978

ARRAY, 117
ASSEMBLER, 4, 99, 100, 114, 284
BAD BLOCK SCAN, 26
BANISH, 55
BLOCK, 117
BLOCKNUMBER, 117
BLOCKREAD, 124, 140, 156
BLOCKWRITE, 124, 140, 156
BOOTSTRAP, 45, 227
BREAKPOINT, 77
CASE STATEMENTS, 135
CHANGE, 19
CHARACTER, 117
CLOSE, 124, 149, 156
COMPILED LISTING, 84
COMPILER, 3, 81, 283
CONCAT, 119, 157
CONDITIONAL ASSEMBLY, 111
CONTROL CHARACTERS, 59
COPY, 51, 120
CP/M, 5, 273
CRAWL, 72
CURSOR, 31, 36, 62
DATE, 25
DEBUGGER, 4, 71, 82, 283
DELETE, 34, 39, 40, 51, 52, 120, 157
DESTINATION, 117
DIRECTIVES, 105
DIRECTORY, 16, 18, 284
DISK ERROR, 26
DISK SIZE, 29
DISK SPACE, 27
DLE, 163
DRAWBLOCK, 129, 157
DRAWLINE, 129, 157, 159
EDITOR, 3, 31, 281
EOF, 125, 138, 141
EOLN, 125, 138, 141, 148
EXAMINE, 26, 72, 74
EXECUTE, 4
EXIT, 142, 157
EXPRESSION, 117
EXTENDED LIST, 18
EXTERNAL, 95, 102, 173
FILE, 123, 125, 148
FILEID, 117
FILENAMES, 7, 11, 31
FILER, 2, 3, 7, 280

FILES, 139
FILLCHAR, 132, 146, 157
FIND, 42, 43, 51
FORWARD, 173
FUNCTION, 107
GENERAL ERRORS, 261
GET, 13, 125
GOTO, 82, 142
GOTOXY, 133, 157, 222, 235, 281
GRAPHICS, 129, 159
HALT, 133, 157
HEAP, 136
IDSEARCH, 157
IMPLEMENTATION, 167
INCLUDE, 83, 100, 115
INDENTATION CODE, 163
INDEX, 117
INITIALIZE DISKS, 28
INPUT, 138, 149
INSERT, 33, 37, 52, 120, 157
INTERACTIVE, 148
INTERFACE, 167
INTRINSICS, 156
IO-ERROR, 125, 249, 251
IORESULT, 83, 125, 157, 184
JUMP, 52
KEYBOARD, 138, 149
KRUNCH, 27
L2 EDITOR, 52
LENGTH, 119, 153, 157
LIBRARIAN, 283
LIBRARY, 173
LINKER, 4, 95, 172, 283
LIST DIRECTORY, 16, 18
LOCK, 124
LOG, 133
LONG INTEGERS, 120, 179, 283
LSI11, 1
MACRO, 104
MACROS, 109
MAKE, 28
MARK, 133, 157
MARKERS, 36, 47, 52
MEMORY ALLOCATION, 136
MEMORY MANAGEMENT, 133
MOVELEFT, 131, 157
MOVERIGHT, 131, 157
NEW, 15, 138
NEXT, 55
NORMAL, 124
NUMBER, 117
OUTPUT, 138, 149
PACK, 147
PACKED ARRAYS, 144

PACKED RECORDS, 146
PACKED VARIABLES, 144
PAGE, 126
PASCAL, 1
PATCH, 284
PDP-11, 99
PDP11, 1, 5, 186
PENSTATES, 255
POS, 119, 157
PREFIX, 25
PROCEDURE, 107
PROGRAM HEADINGS, 147
PSEUDO COMMENT, 71, 82
PSEUDO-OPS, 105
PURGE, 124
PUT, 125
PWROFTEN, 133, 157
QUIET, 85
QUIT, 15, 50, 52, 62
RADAR, 159
RANGECHECK, 85
READ, 126, 148, 154
READLN, 148
RELBLOCK, 117
RELEASE, 133, 157
REMOVE, 20
REPLACE, 42, 44, 52
RESET, 123, 148, 149, 150, 157
RESTRICTIONS, 156
REWRITE, 123, 148, 149, 150, 157
RT-11, 233
RUN, 3
SAVE, 14
SCAN, 131
SCREEN, 118
SCREEN CONTROL, 5, 85, 133, 221, 235, 284
SEEK, 126, 140, 158
SEGMENT PROCEDURE, 150, 165
SETS, 151
SETUP, 284
SIMPLVARIABLE, 117
SIZE, 118
SIZEOF, 133, 146, 158
SOURCE, 118
STR, 120, 158, 180
STRING, 118, 119, 283
STRINGS, 152
SWAPPING, 86
SYNTAX ERRORS, 257
SYSCOM, 81
SYSTEM COMPILATION, 86
SYSTEM LIBRARY, 4, 71, 84, 87, 95, 106
SYSTEM.WRK.CODE, 3, 35, 81, 96, 100
TEXT, 148, 163

TIME, 133, 158
TITLE, 118
TOKEN, 42
TRANSFER, 21
TREESEARCH, 158
TRUNC, 180
UNIT, 87, 97, 167
UNITBUSY, 123, 158
UNITCLEAR, 124
UNITNUMBER, 118, 253
UNITREAD, 123, 158
UNITWAIT, 124, 158
UNITWRITE, 123, 158
UNPACK, 147
UNTCLEAR, 158
USE LIBRARY, 87
USES, 168
VALID, 118
VOLUME, 25
VOLUME NAMES, 7, 253
VOLUMES, 15
WALK, 72
WHAT, 15
WILDCARDS, 11
WORD PROCESSING, 38, 46, 47, 52
WORKFILE, 3, 8, 32, 35, 59, 71, 81, 96, 100, 279
WRITE, 126, 155
WRITELN, 155
Z80, 1, 5, 99, 186
ZERO, 28