UCSD (MINI-MICRO COMPUTER) PASCAL
RELEASE VERSION 1.4 JANUARY 1978

Institute for Information Systems
UCSD Mailcode C-021
La Jolla, CA 92093
(714) 452-4723 (714) 452-4526

```
**************************************
* UCSD (MINI-MICRO COMPUTER)  PASCAL *
* REVISED VERSION  I.4b   APRIL 1978 *
* Institute for Information Systems  *
* UCSD Mailcode C-021                *
* La Jolla, CA  92093                *
* (714) 452-4723      (714) 452-4526 *
**************************************
```

```
***************************************************************
* Copyright (c) 1978 Regents of the University of California, *
*   San Diego Campus.  This software, its source, object, and *
*   all other forms, is the property of the Institute for     *
*   Information Systems and may be used or copied by others   *
*   only with written authorization from the Institute for    *
*   Information Systems.                                      *
***************************************************************
```

NEW IMPLEMENTORS: See "THE FIRST TIME THROUGH"

DISCLAIMER: These documents and/or the software they describe are subject to change and/or correction without notice.

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 Corp., General Automation Corp., The UCSD Computer Center, grants from the University of California Instructional Improvement Program, Tektronix Corp., Micropolis Inc., Computer Power and Light.

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:

S. Dale Ander, Lucia A. Bennett, Charles "Chip" Chapin, Gary R. Dismukes, Julie E. Erwin, Shawn M. Fanning, Joel J. McCormack, Mark D. Overgaard, Keith A. Shillington, Roger T. Sumner, Dennis J. Volper

Software Authors:

S. Dale Ander, Charles "Chip" Chapin, J. Greg Davidson, C. Richard Grunsky, Robert J. Hofkin, Richard S. Kaufmann, Peter A. Lawrence, Joel J. McCormack, Mark D. Overgaard, Keith A. Shillington, Roger T. Sumner, David B. Wollner.

Collected and Edited by:

Keith Allan Snillington

****

* TABLE OF CONTENTS *

****

Version I.4b April 1978

SECTION PAGE

0 PREFACING NOTES

1 INTRODUCTORY NOTES 1
2 BRINGING UP THE PASCAL SYSTEM 3
1 ON PDP-11 3
2 ON 8080/Z80 SYSTEM WITH CP/M AND 3740 DISKS 5
3 ON COMPAL-80 WITH MICROPOLIS MINI-FLOPPY DRIVES 9
3 DIFFERENCES AMONG IMPLEMENTATIONS FOR DIFFERENT PROCESSORS. 11
4 CHANGES MADE IN RECENT RELEASES (I.4, I.4b) OF THE SYSTEM 13

1 THE SYSTEM

1 INTRODUCTION AND OVERVIEW 19
2 FILE HANDLER 25
3 SCREEN ORIENTED EDITOR 31
1 INTRODUCTION 31
2 GETTING STARTED 33
3 DETAILED DESCRIPTION OF COMMANDS 37
4 REFERENCE 53
4 YET ANOTHER LINE ORIENTED EDITOR - YALOE 55
5 DEBUGGER 65
6 PASCAL COMPILER 75
1 COMPILE TIME OPTIONS 77
7 BASIC COMPILER 83

2 UCSD PASCAL

1 INTRINSICS 89
1 STRING 91
2 INPUT/OUTPUT 95
3 TURTLE GRAPHICS 99
4 LOW LEVEL GRAPHICS 101
5 CHARACTER ARRAY MANIPULATION 103
6 MISCELLANEOUS 105
2 DIFFERENCES BETWEEN UCSD'S PASCAL AND STANDARD PASCAL 107

3 IMPLEMENTORS' GUIDES

1 DRAWLINE 133
2 FILE FORMATS 137
3 SEGMENT PROCEDURES 139
4 INTERPRETER NOTES 141
5 INTRODUCTION TO THE PASCAL PSEUDO-MACHINE 157

4 UTILITY PROGRAMS

1 CALCULATOR 179
2 THE LINKER 181
3 SETUP - SYSTEM RECONFIGURATION 185
4 BOOTSTRAP COPIER 195
5 PATCH 197
6 REMOTE PORT TRANSCEIVER KIT 199
7 DEMONSTRATION PROGRAMS 201
8 RT11 TO PASCAL CONVERSION KIT 205
9 HEX-OCTAL-DECIMAL CONVERSION KIT 207
10 CHARACTER SET EDITOR 209
11 FOTOFILE DEVELOPER 211
12 CROSS REFERENCER 213
13 SOURCE COMPARATOR 215
14 GOTOXY PROCEDURE BINDER 217
15 CRASHED DISK RECOVERY PROGRAM 219

5 INSTRUCTIONAL SYSTEM

1 INSTRUCTIONAL SUPPORT (OVERVIEW) 221
2 STRUCTURE OF A UNIT 223
3 DESCRIPTION OF AUTHOR PROCEDURES 225
4 DESIGNING (AND DOCUMENTING) A LESSON FRAME 231
5 A PROCTOR'S GUIDE TO USING THE BOOKKEEPER 233

6 TABLES

1 EXECUTION ERRORS 241
2 IORESULTS 243
3 UNITNUMBERS 245
4 PENSTATES 247
5 SYNTAX ERRORS LOCAL TO UCSD'S PASCAL 249
6 AMERICAN STANDARD CODE for INFORMATION INTERCHANGE 251
7 UCSD PASCAL SYNTAX DIAGRAMS 253

```
********************* ***************
* INTRODUCTORY NOTES * * Section 0.1 *
********************* ***************
```

Version I.4b   April 1978

I.4b is the first UCSD PASCAL release which actually supports multiple types of processors (in particular the PDP-11 and the 8080/Z80). (There are implementations on several other processors, but none of them are directly supported by the PASCAL Project.)

As later portions of this document will detail, 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, all the PASCAL software will also run on that new real machine.

So 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. Section 0.3 lists the differences between the PDP-11 and 8080/Z80 interpreters. Most of these are sophisticated and/or relatively unused features that have not yet been implemented for the 8080/Z80.

Another class of differences stems from variations in the system I/O environments rather than in the host processor. Included here are differences in system console terminal types (eg. 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. The utility program "SETUP" (see Section 4.3) is provided to adapt the PASCAL system to the idiosyncrasies of the available terminal. Differences in mass storage medium (eg. floppy disk vs cartridge disk vs mini-floppy) and interface/controller (eg. programmed I/O vs DMA) are harder to deal with.

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 incompatibile devices but make no claim to support users who want to develop their own such drivers. Section 0.2.1 describes the process of bringing up a PDP-11 version of the system and some of the difficulties that may arise.

The situation in the 8080/Z80 world is much more chaotic. It would just not be practical for the Project to write and support drivers for the vast multitude of 8080/Z80 I/O environments that exist. Therefore 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.

Our dominant mode of distribution for 8080/Z80 systems will be on 3740 compatible floppy disks. One of the distributed disks 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. Section 0.2.2 details 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.

Currently the only specialized 8080/Z80 environment that we support is the COMPAL 80 with Micropolis mini-floppy drives. Section 0.2.3 describes the idiosyncrasies of that environment.

The final subsection (0.4) summarizes differences among the most recent releases of the PASCAL system.

```
************************** *****************
* THE FIRST TIME THROUGH * * Section 0.2.1 *
************************** *****************
```

Version I.4   January 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 document.

If this is not the case then here are a few of the problems we have encountered with I.3 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.) We haven't encountered your problem before. Call:

Institute for Information Systems (714) 452-4723.

- Notes -

```
************************************ *****************
* 8080/Z80 WITH CP/M & 3740 DISKS * * Section 0.2.2 *
************************************ *****************
```

Version I4.b April 1978

A discussion of the CP/M implementation of UCSD Pascal follows.

Booting Pascal

To first get Pascal running under your version of CP/M, a two-disk bootstrap is used. The first step is to boot CP/M in your 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 advised that you create a special CP/M for use with Pascal in order to have Pascal running in the environment it was designed for.

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 [cntrl-f] as a "flush-output" toggle and signal the character-out routine to gobble any characters until signaled again. When it receives another [cntrl-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-48)*1024 + 0BA00H, then 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-48)*1024 + BA00H), and we
                        ;   want it at location 980H
C380  4F80
-r4f80                  ; read it in
NEXT   PC
0A7d  BA00

-ibios48.hex            ; and lastly read BIOS into location D80H ?
-hd80 be00
C380  4F80
-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>
```

- Notes -

```
************* *****************
* COMPAL-80 * * Section 0.2.3 *
************* *****************
```

Version I.4b   April 1978

The COMPAL-80 implementation of the Pascal system.

Booting:

Put the disk labeled PASCAL into drive 0, push the RESET button, and type F400G. In about 15 seconds the Pascal system should greet you.

Copying a booter:

The Pascal system uses a slightly different data layout within a sector than the standard Micropolis format. For this reason creating a booter and formatting a new disk are done at the same time. Simply use the DISKCOPY program supplied with BASIC to copy the disk labeled PASCAL ontc another mini-floppy. The new disk is now a valid Pascal disk with a booter and can be Z(eroed in the filer.

General:

1. [Cntrl-a] is used as an alphalock on keyboards without such a key.
2. The size of a Micropolis Mod II disk is 600 Pascal blocks.
3. Up to four disk drives can be supported and are equated to Pascal units 9..12.
4. The screen appears as a small Datamedia screen to the Pascal system, which means it has erase-eol, erase-eos, cursor positioning, etc.
5. [Cntrl-f] and [cntrl-s] work.
6. Size of the typeahead buffer is 31 characters.
7. Output to screen truncates after 64 characters. If WIDTH is set to 0 in SETUP, wrap-around occurs.

- Notes -

* DIFFERENCES AMONG IMPLEMENTATIONS FOR DIFFERENT PROCESSORS * * Section 0.3 *

Version I4.b April 1978

Differences between PDP11 Pascal and 8080/Z80 Pascal.

1. The definition of *div* is different (thereby changing the values returned by *mod*):
    a *div* b = floor(a/b)
    a *mod* b = a - b*(a *div* b)

2. The following floating point routines are not implemented:
    sin, cos, atan, exp, ln, log, sqt

3. The I/O drivers are all written for synchronous operation. This means that [break] has no effect. [Cntrl-s] and [cntrl-f] will not perform as described unless

    a. you have a keyboard interrupt handler, and this handler is modified as specified below in *Modifications to CPM*, or
    b. you have a COMPAL-80 system.

    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.)

4. The interpreter is called SYSTEM.MICRO instead of SYSTEM.INTERP.

5. Neither the CP/M nor the COMPAL-80 implementations have bootstraps that are 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.

6. There are no turtle graphics procedures in the interpreter. Users with bit-mapped graphics devices are advised to see section 3.1 of the documenta' 'on for a Pascal version of DRAWLINE.

- Notes -

```
*********************************** ***************
* CHANGES MADE IN RECENT RELEASES * * Section 0.4 *
*********************************** ***************
```

Version I4.b April 1978

SUMMARY OF DIFFERENCES BETWEEN UCSD PASCAL RELEASES I.3 AND I.4

The following additions, improvements and/or corrections apply to Version I.4. Reference the (section #) preceeding each entry for a more detailed description.

FILE HANDLER

(2.1) T(ransfers of large files are supported for single-drive systems. XFER program is no longer available for this purpose.

(1.2) T(ransfers endangering the directory of the destination volume are prefaced with the warning:
"Risk dir of <VOLID>?"

(1.2) Similarly the Z(ero command asks for verification: "Destroy <VOLID>: ? " (if disk to be zeroed is named).

(1.2) Z(ero command offers "duplicate directory" option.

(1.2) "Zero what unit?" has been replaced with "Zero dir of what volume?" in the Z(ero command.

(1.2) '#' followed by <hardware unit number> is interchangeable with VOLID throughout the system.

(1.6) &
(2.1.2) RESET and REWRITE are intended to replace OPENOLD and OPENNEW respectively, which are being "phased out" of the system RESET has an optional second parameter of type STRING.

(1.2) &
(6,TAB 3) RK-05s are supported as Units #9-12.

(1.2) Date is stored on system disk and remains as set until changed by D(ate command.

(---) Screen is no longer automatically cleared between successive commands in the Filer. Screen erase is accomplished by typing <sp> or <cr>. Suggestion: type <sp> before T(ransferring to CONSOLE device.

(---) Files are no longer extended by the system if an attempt is made to write beyond end-of-file.

(---) Blocks 0 and 1 of .TEXT files are not transferred to non-block-structured devices.

(1.2) <cr> no longer indicates default volume. Instead <cr> simply returns user to Filer command level. Note (in particular for L,B,E,X,K and Z Filer commands):
'*' denotes system disk
':' (implying empty VOLID) denotes default disk.

(1.2) When opening files for output with the [<# of blocks>] option, '*' substituted for <# of blocks> causes the second largest or half the largest area available (whichever is bigger) to be allocated.

<u>EDITORS</u> (Sections 1.3 and 1.4)

Two different editors are currently provided with the UCSD PASCAL system: YALOE and "EDITOR". EDITOR is a substantially more powerful (and even easier to use) editor, but it makes some assumptions about the run-time environment.

EDITOR requires a reasonably powerful CRT terminal with the following features:

ERASEEOS - the capability to erase from the cursor to the end of the screen

ERASEEOL - erase from the cursor to the end of the line

XYADRESSING - go directly to a given row and column on the screen

NDFS - non-destructive forward space (the inverse of back-space)

HOME - goes to upper left-hand corner

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)

Typing "E" at the main command level will execute the file SYSTEM.EDITOR. Selection of either YALOE or EDITOR as the system editor is made in the Filer by C(hanging the selected file's name to SYSTEM.EDITOR.

Notes: Currently YALOE can handle larger files than EDITOR. In future releases EDITOR may handle arbitrarily large files.
Proper use of EDITOR requires that the system disk be left on-line while editing.

DEBUGGER

(1.5) W(alk, C(rawl and B(reakpoint facilities have been implemented.

PASCAL COMPILER

(1.6) COMPILE-TIME OPTION changes:

- (1.6) '+' is assumed if neither '+' nor '-' appears after the option letter in a compiler-option comment.
- (1.6) "L"ist may specify a <file name>.
- (1.6) "Q"uiet option is provided to suppress output to the CONSOLE device during compilation.

(2.2.6) Standard type INTERACTIVE (vs TEXT) is introduced.

(---) Standard constant MAXINT = 32,767 is provided.

(2.1.2) READ(STRING) will read up to the end-of-line character and set EOLN(FILEID) true. Subsequent READs of STRING variables will return the null string until a READLN or READ(CH) is executed.

(2.2.2) Pecularities of .TEXT format (ie. blank compression codes and special first page) will be transparent to READS and WRITES on files with logical records of type CHAR and titles with the .TEXT suffix.

(2.2.2) '{' and '}' are accepted as comment delimeters in addition to '(*' and '*)'.

(---) '@' notation for pointers is no longer valid. Use '^' only.

(2.2.14)
& (3.5) Segment procedures may be declared forward succesfully.

(2.2.1) Semicolon before the "END" in a CASE statement is optional.

(2.2.16) String comparison is lexicographic, eg. :
'ABCD' < 'XYZ' despite comparitive lengths of strings.

(1.1.2) Typing "E" when an error is found during compilation invokes the system editor, whereas previously typing <sp> to continue or <esc> to abort were the only alternatives.

BASIC COMPILER

(1.7) A BASIC compiler is now provided.

INTRINSICS

(2.1.6)
&(4.14) GOTOXY intrinsic is provided for screen cursor addressing.

(2.1.2) SEEK instrinsic allows random access to a logical record.

(2.1.2) Optional fifth parameter to UNITREAD and UNITWRITE is now of type INTEGER not BOOLEAN.

(2.1.2)
& (2.2) RESET and REWRITE are intended to replace synonomous intrinsics OPENOLD and OPENNEW respectively, which are being "phased out" of the system. RESET has an optional second parameter of type STRING.

IMPLEMENTORS' GUIDE

(3.4) NOTES on the Pascal INTERPRETER are provided.

(3.6) Files with the reserved suffix .TEXT may include blank compression codes.

UTILITY PROGRAMS

(4) &
(1.1) Many new UTILITY PROGRAMS have been added. Reference also the INTRODUCTION and OVERVIEW document (Section 1.1) for a brief description.

INSTRUCTIONAL SYSTEM

(5) An INSTRUCTIONAL SYSTEM is now provided.

TABLES

(6.6) An ASCII-HEX-OCTAL-DECIMAL table is now provided.

MISC

(----) Textual run-time errors are written to CONSOLE device provided the system disk is accessible during execution.

SUMMARY OF DIFFERENCES BETWEEN UCSD PASCAL RELEASES I.4 AND I.4b

1. The Z80/8080 PASCAL system has been released.

2. A new system file, SYSTEM.MISCINFO, must be present at boot time. This file contains information about the terminal attached to the system and is the file now written into when the P(ermanent command in SETUP is executed.

3. The compiler is now a separate file, SYSTEM.COMPILER, and has had a few modifications made to it:
   a. It will allow the compilation of small programs in 48K bytes, as previously claimed.
   b. Formfeeds immediately following a [return] are accepted in the input file.
   c. A semi-colon immediately before the END in a record declaration is allowed.
   d. The pre-defined function MEMAVAIL: INTEGER is implemented; it returns the number of words available for use between the stack and the heap.
   e. The pre-defined procedure EXIT has been extended to allow the name of the program as its parameter. The old syntax, EXIT(PROGRAM), is still valid.

4. Formatting of real numbers to ASCII characters has been changed (substantially improved).

```
****************************** ***************
* INTRODUCTION AND OVERVIEW * * Section 1.1 *
****************************** ***************
```

Version I.4   January 1978

1. INTRODUCTION

The U.C.S.D. PASCAL system described in the following set of documents 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". Software maintenance and enhancement is made relatively straightforward by the fact that, except for the P-machine interpreter (and a few run-time support routines for efficiency), all of the system software is written in Pascal.

The current system now runs on the Digital Equipment Corporation PDP 11 series. Implementations for the Zilog Z80 and the 8080 microprocessor are currently being developed, expected early this year. The system is designed to be used primarily with a CRT terminal acting as the CONSOLE device; the system is flexible enough, however, to be reconfigured for slower hard-copy terminals as well. For further information regarding compatability between various types of equipment and this system see the "SETUP" document in Section 4.3. These documents are 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 U.C.S.D. Pascal and Standard Pascal see Section 2.2.

2. U.C.S.D. PASCAL SYSTEM: AN OVERVIEW

The structure of the U.C.S.D. Pascal system is best conceptualized in terms of the "tree-like" structure diagram figure 0.1 at the end of this sub-sectiion.

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(dit, R(un, F(ile, C(ompile, X(ecute, D(ebug, I(nit, H(alt
```

If the user types "F" he will "descend" a level within the structure diagram into a level called the "Filer". Upon entering the Filer, the user will receive another prompt-line detailing the set of commands which are available to him at the Filer level of the system. One of the Filer level commands is Q(uit. This command causes the user to exit from the Filer level and "ascend" back to the outermost command level of the system. At this point in time, the user is back to the level in the system from where he started after bootstrapping the machine. Some commands within the system prompt the user for the name of some disk file. In the case of these commands, 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, then the backspace key (or equivalent key depending upon the present system configuration) may be used to "back over" and erase the erroneous part. The 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 that he does not wish the system to accept any file name whatsoever, then he may "escape" from this command 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. For example, in the current release this is particularly true at the Filer level of the system. (A complete list of commands at the Filer level may be found in the Section 1.2).

A concept central to the design of the entire U.C.S.D. 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. The workfile is not necessarily just one file on the user's disk, but can be a number of files (usually source & code) which together comprise the "workfile". A user is allowed only one workfile at any one time. Therefore if a user wishes to work with a new workfile (i.e. go on to greener pastures) while at the same time preserving the contents of his current workfile, he must "save" the contents of the workfile under a separate file name on his disk by using the S(ave command in the Filer level of the system. Likewise, old workfiles may be retrieved from the disk and loaded into the workfile using the G(et command in the Filer level.

3. OUTERMOST LEVEL COMMANDS: AN OVERVIEW

A. E(dit

This command is invoked by typing "E" while at the outermost command level of the system. This command 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, along with many other powerful commands. (See Section 1.3 for details.) The workfile text (if present) is read into the editor buffer.

B. F(iler

This command 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 floppy disk. Some typical commands are the L(dir and T(ransfer commands. 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(ransfer 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. Also, as mentioned in the OVERVIEW section, there are commands associated with the "getting", "saving", and "clearing" of the user's workfile. (For more documentation on the Filer level of the system see Section 1.2 below).

C. C(ompile

This command calls the Pascal compiler into memory and causes the contents of the current workfile to be compiled. If an error in the program within the workfile is detected, the compiler will stop and display the error and the surrounding text of the program. By typing a space, the user can cause the compiler to resume the compilation. Typing an <esc> will cause the compiler to abort & return to C(ommand level. Typing 'E' will call in the editor, and if the system editor is the screen editor, the cursor will be placed 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 U.C.S.D. 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 file currently exists, the compiler is called in the same manner as described in C above. 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.) 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, Q(uit the Filer, and then R(un the program.

F. 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 just 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. (Further documentation on the interactive Debugger can be found in the Section 1.5 below.)

G. I(nit and H(alt

The I(nit command causes the system to re-initialize.

The H(alt command causes the computer to halt.

4. UTILITY PROGRAMS INCLUDED IN THIS RELEASE: AN OVERVIEW

Included in this release of the U.C.S.D. Pascal system is the following set of utility programs:

A. Calculator

Disk file title: CALC.CODE

This is a Pascal program which allows a user to use the computer as a calculator to make quick mathematical computations. See Section 4.1 below for further details.

B. Linker

Disk file title: LINKER.CODE

This is a program used to link together segment procedures anc/or functions which have been compiled separately. See Section 4.2 for further details.

C. Setup

Disk file title: SETUP.CODE

This is a Pascal program which can be used to reconfigure the system for use on different terminals or devices. See Section 4.3 for further details.

D. Booter

Disk file title: BOOTER.CODE

This is a Pascal program which copies the bootstrap from any one floppy disk to another. This program is designed to be used with one disk drive. See Section 4.4 for further details.

Note: BOOTER.CODE will not work on Z80 or 8080 microprocessors because the bootstrap area on those systems is not accessible in Pascal.

The number of utility programs has grown past the scope of this sub-section. For a complete list of the utility programs now available with your 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 welcomed contributions. A separate paper by K. Bowles on Software/Courseware exchange is available upon request. This paper proposes a mechanisim for the exchange of software and courseware.

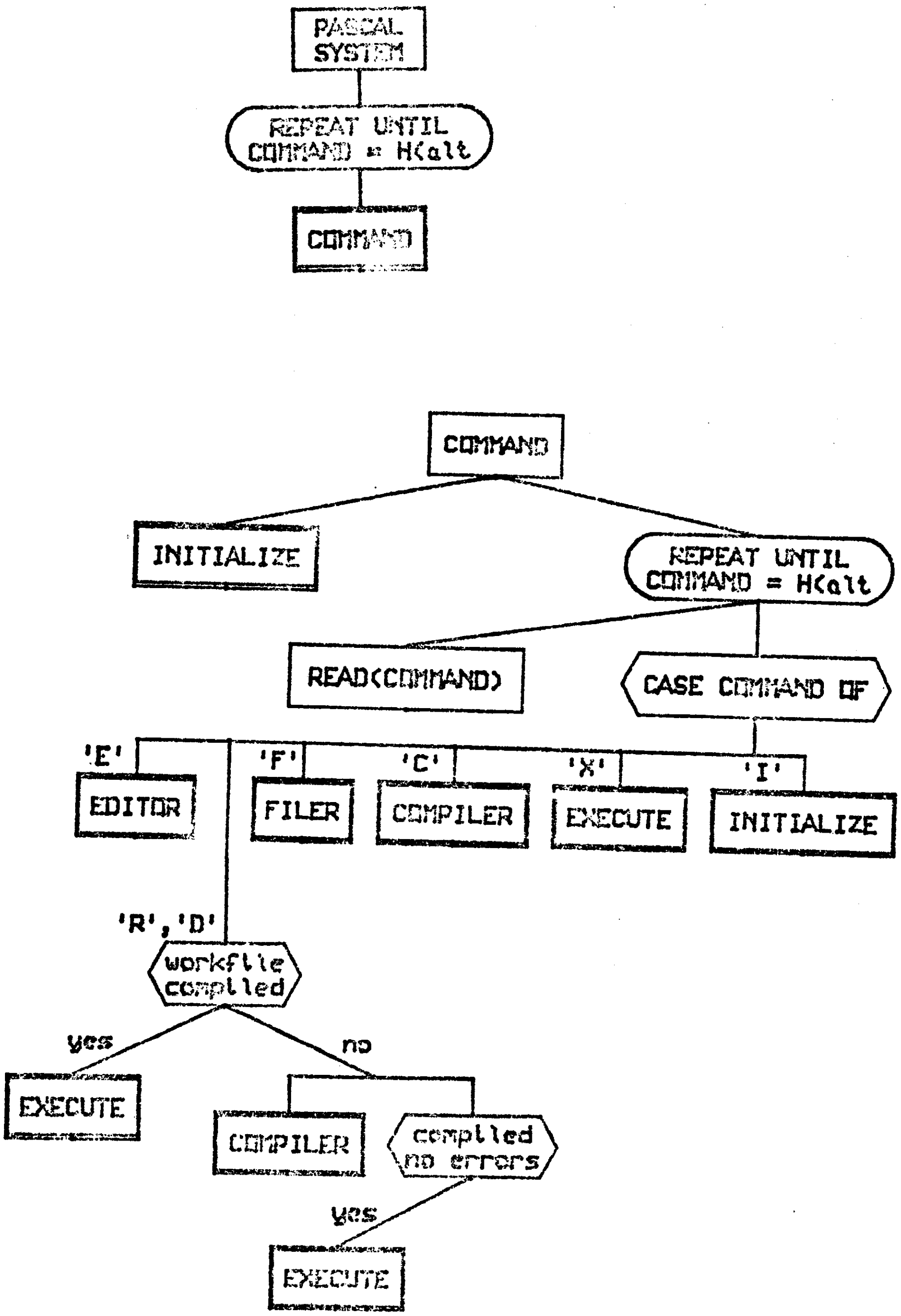

PASCAL SYSTEM
REPEAT UNTIL COMMAND = H(alt
COMMAND
COMMAND
INITIALIZE
REPEAT UNTIL COMMAND = H(alt
READ(COMMAND)
CASE COMMAND OF
'E'
EDITOR
'F'
FILER
'C'
COMPILER
'X'
EXECUTE
'I'
INITIALIZE
'R','D'
workfile compiled
yes
EXECUTE
no
COMPILER
compiled no errors
yes
EXECUTE

Figure 0.1

* FILE HANDLER * * Section 1.2 *

Version I.4 January 1978

File Names and Structure

Files are maintained in 512-byte physical blocks similar to those used with the PDP11 line of computers. Initially, the layout of these blocks on a floppy disk will use alternate 128-byte sectors to retain compatibility with PDP11 files. However, we anticipate using the system with high performance floppy disk drives on which adjacent sectors may be used; the system will provide this capability as an option. Media other than flexible diskettes will be made available as the system evolves, such as the RK-05 disk, which the system is capable of dealing with in a somewhat limited capacity.

Each file is stored in a contiguous area of the disk and is pointed to by the disk directory. Each file is identified by a unique string of up to 15 characters containing letters of the alphabet, digits, and the special character period ("."). Following are examples of legal file names:

```
WHOPPER
ONE.TEXT          (8 characters long)
ONE.CODE
ONE.1.CODE        (10 characters long)
LONG.FILE.NAME
```

The system will translate lower case letters to upper case and will remove blanks and non-printing characters for storage of a directory title. The user may employ the period character to indicate hierarchic relationships among files and/or to distinguish several related files of different types. If the last identifier following a period in a file title is one of several reserved words, the file will be assumed to be formatted a--ording to the named file type. For example, ONE.A.CODE might be the compiled object code file associated with a source program in ONE.A.TEXT. The file types currently defined are GRAF, FOTO, BAD, TEXT, and CODE.

The reserved suffixes for filenames are:

.TEXT	Editor files. Editable, compilable, listable.
.CODE	Code files. Runnable. eXecutable.
.FOTO	Screen image file. Bit data.
.GRAF	Editable vector lists.
.BAD	Nonreadable files. Cannot be moved.

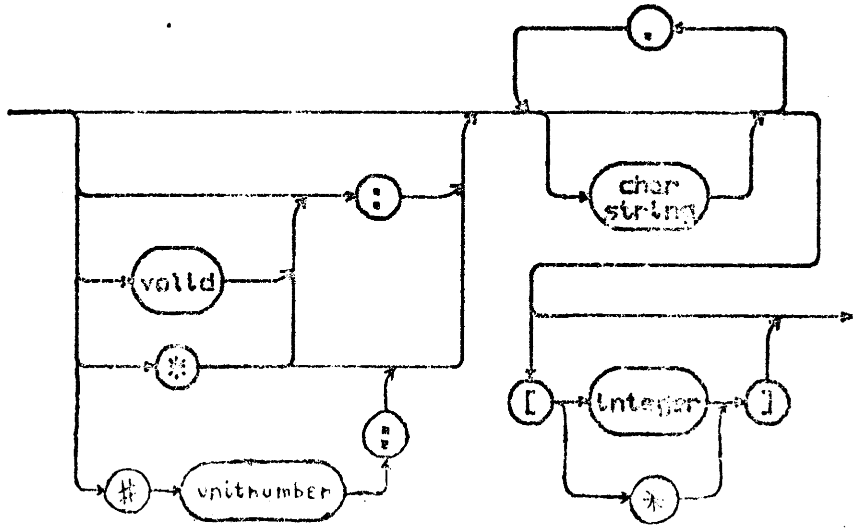
Each disk has a Volume Name associated with it. A volume name consists of up to 7 alphanumeric characters. The disk from which the system is initialized is called the "System Disk", and its volume name may be abbreviated "*".

Non-file structured devices (line printers, terminals, etc.) also have volume names. Thus all I/O occurs to or from 'volumes' (which may or may not have individual files). Throughout the system (in the file handler as well as user programs) files may be associated with actual areas on a disk or with other physical devices. The reserved volume names used to refer to these devices are as follows:

CONSOLE:	screen & keyboard with echo
SYSTERM:	screen & keyboard without echo
GRAPHIC:	the graphic 'side' of the screen (for 8510a's)
PRINTER:	the line printer
REMOTE:	for future expansion

One may define a particular volume to be the 'default volume'. The P(refix command at the FILE level is used for this purpose. It allows the user to set the default volume name which is attached to filenames. The System Disk is assumed to be the default volume immediately after bootloading. The default volume is the volume assumed in all file titles where no explicit volume is given. The syntax for a file name is as follows:

FILE TITLE



This syntax diagram is just like those published with the documentation on the PASCAL language. The use of "*" or "*:" preceding a file name refers to the system disk. A volume name, if given, must be separated from the file name with a ":".

When specifying file titles for output you may put the file in the first area of adequate size by adding [<# of blocks>] after the file name. <# of blocks> is the area size you would like the file to fit into. This number can be equal to or larger than the file length. If it is zero or omitted then the file will be put into the largest empty place available. If <# of blocks> is the character "*", either the second largest or half of the largest area available is allocated, whichever is larger. (RT-11 users are familiar with this scheme.)

All devices that may be on-line have built-in Unit Numbers predesignated by the system. Reference Table 3. (page 245)

File Level Command

Many of the following commands prompt the user for one or two file titles. Responding to any request for a file title by typing just a carriage return causes the command to return the user to the F(iler level. In the case of commands that permanently alter the state of a file, the user may be prompted to verify that the requested action is really wanted. If "Y" (for "yes") is typed, the Filer will proceed to do the specified action. Any other response to this prompt will result in a return to the main File level, with the action not occuring. When a volume name is requested, ':' implies the default prefix volume, '*' implies the booted volume, '#n' implies the volume unit-number n.

G(et) Opens the requested file with an implicit ".TEXT" suffix as the work file. The file with an implicit ".CODE" suffix is also gotten if one exists.

S(ave Removes old file by that name, renames SYSTEM.WRK files to that name.

N(ew) Clears Workspace.

L(dir) Lists the directory of the volume specified after the prompt. '*' infers the root, or booted device, ':' infers the default device, null exits to F(iler.

C(hange) Changes the title of a file or volume name of a disk to a new name.

R(emove) Removes the indicated file from the directory on the volume. NOTE: To remove SYSTEM.WRK.TEXT and/or SYSTEM.WRK.CODE the N(ew command should be used, or the system may get confused. R(emoving a volume name causes it to go off-line.

T(ransfer) Copies the contents of the first specified file to the second specified file. The second specified file is created as a new file. Note: Files may be transfered to volumes that are not directory structured, such as CONSOLE and PRINTER, by just specifying the volume name followed by a colon ":". If you transfer to a volume with a directory on it by specifying only its volume name, i.e. "T VOLA:,VOLB:<cr>", you will be asked if you wish to risk the directory of VOLB. This is to ensure that you indeed want to transfer the specified source to VOLB:, a process which wipes out the current directory of VOLB. Single-drive transfers may be accomplished by the following sequence:

1.) Type 'T' for transfer.
2.) Ensure that source disk is in drive.
3.) Type source filename <cr>.
4.) Wait for prompt "to what file?".
5.) Ensure that destination disk is in drive.
6.) Type destination filename <cr>.
7.) Follow prompting messages until F(iler prompt returns.

D(ate) Displays current date and enables you to change it. The format for the date is given in the prompt line. This date will be associated with any files saved in the current session and will show up by those files when using the L(dir) or E(xtended list) commands. The date is stored on the system (*) disk and remains the same until changed with the D(ate command.

W(hat) Informs the user if his workfile exists, is saved or not, and what its name is.

Q(uit) Returns the user to the main command level.

The following commands will not appear on the promptline due to lack of space, and it is assumed that they will be used only by experienced users:

P(refix) Changes the current default to the volume specified after the prompt.

M(ake) Allows the user to create a new file under the name given after the prompt, followed by the number of blocks wanted within square brackets, e.g. MYFILE[20]. This command is useful for filling unpleasant gaps in the directory, if you are being selective as to where files go.

V(olumes) Displays the names and associated unit numbers of volumes currently on-line. The name of the system volume will be preceded by a '*', the default volume by a 'P', and any other volume which is directory structured by a '#'.

B(ad blocks) Checks each block on the indicated volume for unrecoverable errors and lists the number of each bad block.

E(xtended list) Lists directory in more detail than the L 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.

X(amine) First asks the name of the volume then the block range to be examined, (eg. 35-63 or just 18). If any files are in danger of being removed by this process, it will inform you of such an event and ask if you want to risk losing these files. An ensuing scan process reports located bad areas which it wants to "mark bad". A Y(es response from the user will initiate M(aking a .BAD file over those areas.

K(runch) Crunches or compresses the files on the specified volume so that free blocks are combined into one area. WARNING: It is advisable to do a B(ad block scan, and then if necessary an X(amine, prior to K(runching because K(runch will try to move unmarked bad blocks and/or put good files into bad areas. Do not disturb the disk until K(runch tells you it has completed its task. K(runching the system volume may require rebootstrapping.

Z(ero) Re-initializes the indicated volume by zeroing out the directory and giving the disk a new name. You will be prompted for the volume to zero; reply with a volume name or equivalent unit number (type "#n") of the disk to be zeroed. If it has a name, regardless of how you specified it, you will be asked "Destroy dir?", verifying that you indeed want to zero this disk. You will also be asked "Duplicate Dir?". This is an option available for disks upon which you want to keep a redundant copy of the directory. Specifying yes (typing "Y") at this time will cause the directory of this disk to be written in two locations every time the directory is written out. This option makes recovery from directory failure an easier task. The consequence of this is a slight slowing of the system at directory writing time. You will be asked to enter the size of the volume (you will be given a clue as to what number to enter). Then you will be asked for the new name of the disk and asked to confirm it one more time. Finally the new directory gets written onto the disk.

- Notes -

```
*************************** *****************
* SCREEN ORIENTED EDITOR * * Section 1.3.1 *
*************************** *****************
        Version I.4      January 1978
```

Introduction

The Scope of This Document

This document describes the Screen Oriented Editor (Version [E.4]). The purpose of the document is to provide for the user of the Editor an introduction and a reference. The document itself is divided into four sections. The first is this introductory section which describes the philosphy behind what the Editor does. The second is a tutorial section 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. The fourth section is a quick reference section.

The Concept of a 'Window' on the Program

The Screen Oriented Editor is specifically designed for use with Video Display Terminals. Two of the chief properties of those terminals are: 1) that they display a fixed amount of material at a time, that is, one screenful; and 2) that they are readily updated. The Editor is designed to take as much advantage of those display properties as is possible and in particular to use the updating property to keep in front of the user the current status of the portion of the program near which he is working. On entering any file the Editor displays the start of the file in the upper left corner of the screen. If the file is so long that it will not all fit onto the screen only the first portion appears. The whole file is there but you can only see a portion of it through the 'window' of the screen. Indeed the whole file is accessible by the Editor commands and when any Editor command takes the user to a position in the file which is not displayed, the "window" is updated to show a portion of the file near the place to which the user has moved.

The Concept of a Cursor

Moving around in the file is done with the use of the cursor. The cursor represents your exact position in the file. The window you are able to see is a portion of the file which is near the cursor. To see another portion of the file you merely 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 which you have changed. In those cases the portion of the screen where the cursor stopped is displayed. In no editing case is it necessary for the user to operate on portions of the text he cannot see on the screen, but in some cases it is optional.

The Concept of a Prompt Line

The Editor, consistant with the rest of the Pascal System, displays a prompt line as the top line of the screen. The purpose of the prompt line is 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. The entry or Command level of the Editor displays the following prompt line:

```
>Edit: A(djust C(py D(lete F(ind I(nsrt J(mp Rplace Q(uit X(chng Z(ap  [E.4]
```

Getting Started with the Screen Oriented Editor

Entering the Workfile and Getting a Program

When you first come into the Editor you may be asked:

No workfile is present. File? (<ret> for no file)
You may answer this question two ways:

1) With a name (like "STRING1<ret>"). This means that you wish to get a file from your disk so you can modify (edit) it.

What you see on the screen after typing the name is a copy of the text of the first part of the file. For example if your disk had a file called STRING1 which contained a program called STRING1 and after the "No workfile..." prompt line you typed "STRING1<ret>", the program shown in Figure 2.1 could appear 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 is called <ret> on the prompt line). This means that you wish to start an entirely new file from scratch.

The only thing visible on the screen after doing this is the editor prompt line. You have started a new workfile and currently have nothing in it. You will probably wish to type "I" and start inserting a program or text of your own design.

Workfiles: If a workfile already exists then no questions are asked. The workfile is displ yed and can be modified. The workfile can be cleared so that you can start a file by using the N)ew command in the Filer.

Moving the Cursor

As mentioned above: the cursor is the center for all editing activity. Therefore in order to edit, it is necessary to move the cursor. There are many commands that move the cursor; we will start with the simplest. On your keyboard are four keys with arrows (they look may like triangles) on them. These four keys will move the cursor. The <up-arrow> will move the cursor up one line, the <right-arrow> will move the cursor right one space and so forth.

If you try experimenting with the cursor in a program such as STRING1 you will notice that the cursor does not like to be outside of the text of the program. For example if you are after the "N" in "BEGIN" (see Figure 2.2 below) and push the <right-arrow>, you will notice the cursor moves to the "W" in "WRITE". Similarly if you are at the "W" in "WRITE('TOO WISE ');" and use <left-arrow> you will move to after the "N" in "BEGIN".

Figure 2.2

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

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

Let us go through the cursor moves needed to get ready to modify STRING1 to write SMART instead of WISE. You will want to change the "WRITE('TOO WISE ');" found in the third line to a "WRITE('TOO SMART ');". To do this you must get the cursor to the right spot.

For example: if the cursor is at the "P" in "PROGRAM STRING1;". You need to go down two lines, so, press the down arrow 2 times. To mark the positions the cursor occupies they are labeled a,b,c in Figure 2.3. "a" is the initial position of the cursor; "b" is where the cursor is after the first <down-arrow>; "c" is after the second <down-arrow>.

Figure 2.3

```
aROGRAM STRING1
bEGIN
c WRITE('TOO WISE ');
```

Similarly you can move it to the right until it sits at 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".

Using I(nsert

The next thing you wish to be able to do is to insert something into the text. The Command level prompt line reminds you that to I(nsrt (insert) an item you need to type "I". Let's go through the process of insertion using the STRING1 program which you saw above. In order to insert it is first necessary to move the cursor to the place you wish to make the insertion. Earlier, you have moved the cursor to the "W" in "TOO WISE"; now, if you type "I" you will make an insertion before the "W". (If the cursor was at the "S" you would be making the insertion in front of the "S".) To help to remind you of the context of the insertion, the Editor displays the last part of the line on the right side of the screen. After you type "I" the following prompt line should appear on the screen:

```
>Insert: text {<bs> a char,<del> a line} [<etx> accepts, <esc> excapes]
```

If that prompt line did not appear at the top of your screen you are NOT in insert mode and cannot insert. You may have typed a wrong key.

Assuming you were at the "W", did properly type "I" and got the insert prompt line now you may insert "SMART" by typing the five letters "SMART". They will appear on the screen as you type them.

Now there remains one more important step. You have the choice indicated at the end of the prompt line: you can push the <etx> key and accept the insertion, or you can push the <esc> key and the insertion will disappear.

Figure 2.4 (Screen after typing "SMART")

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

Figure 2.5 (Screen after <etx>)

```
-------------------------------------------------------------------------
BEGIN
  WRITE('TOO SMARTWISE ');
-------------------------------------------------------------------------
```

Figure 2.6 (Screen after <esc>)

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

It is legal to insert a carriage return. This is done by typing <return> while in the INSERT mode. It causes the Editor to start a new line.

Using D(elete

The DELETE mode works like the INSERT mode. Having inserted the SMART into the STRING1 program and having pushed <etx> you now wish to delete the WISE. Step 1 is to move the cursor to the first of the items you wish to delete. Step 2 is to type a "D" to put the Editor into DELETE mode. The following prompt line should appear:

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

Now every time you type <space> a letter will disappear. So in our example typing 4 spaces will cause the "WISE" to disappear. Then you have the same choice as in insert. You can type <etx> and the proposed deletion is made or you can type <esc> and the proposed deletion reappears and remains part of the text.

As in insert it is legal to delete a carriage return. Go to the end of the line, enter DELETE mode, and <space> until the cursor moves to the beginning of the next line.

At this point you have learned sufficient commands to edit any file you desire. There are many more commands in the Editor which make editing easier. These are described in the next section of this document.

Leaving the Editor and Updating the Workfile

When you have finally finished making all the changes and additions you desire you will wish to exit the Editor and "save" a copy of the modified program. The Editor prompt line shown above reminds us that one of the options is "Q" for Q(uit.

Typing "Q" will cause the prompting display shown in Figure 2.7.

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 your modified file on your disk is to type "U" for U(pdate. This will cause a workfile to be saved. With the workfile thus saved it is possible to use the R(un command (provided of course your file is a program). It is also possible to use the S(ave option in the Filer to save your modified file in your library before you use the Editor to modify or create another file.

```
************************************ *****************
* DETAILED DESCRIPTION OF COMMANDS * * Section 1.3.3 *
************************************ *****************
```

A Detailed Description of Each Command with Examples

INTRODUCTION

Command level: The outer or entrance level of the Editor is called the Command level. From this level each of the Editor commands can be reached. Whenever you enter or return to the Command level the Editor redisplays the "Edit:" prompt line shown in section 1.

Certain options affect many commands. For conciseness these are grouped together in this introduction. Detailed descriptions of the commands follow this introduction. These are grouped in the following scheme: First the moving commands (cursor moving commands, Jump, Page, Equals), second the text changing commands (Insert, Delete, Zap, Copy, eXchange), third the Find and Replace commands, fourth the formatting commands (Adjust, Margin), and fifth the miscellaneous commands (Set, Verify and Quit).

Command and Mode: At the Command level there are many options. For convenience we will refer to some of these options as commands and some of them as modes depending upon the appearance of a prompt . If an option executes a task and returns control to the Command level we will call that option a command. If an option issues a prompt and gives the user another level of options we will call that option a mode.

Repeat-factors: Many of the commands allow repeat-factors. A repeat-factor is applied to a command by typing a number immediately before issuing the command. The execution of the command is repeated for the number of times indicated by the repeat-factor. For example: typing "2 <down-arrow>" will cause the <down-arrow> commmand 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.

The cursor: It should be remarked that the cursor is never really "at" a character. The cursor is only allowed to be "between" characters. When we say the cursor is at the letter "R" it is actually between the letter "R" and the letter in front of it. You can notice this most clearly on the insert command when it inserts in front of the character the cursor was "at". On the screen the cursor is placed "at" "R" when it is really before "R" to make it easier to display.

The <arrow> keys: The four keys which this document refers to as <up-arrow>, <down-arrow>,<left-arrow> and <right-arrow> are implementation dependent. The implementor of the PASCAL system has the option using the utility program SETUP, which is described in the User Documents, to redefine the keys of his choice as the keys the Editor takes to be the <arrow> keys. Consult local documentation in case of such a redefinition.

Direction: Certain commands are affected by direction. If the direction is forward then they operate forward through the file. Forward is the standard direction of reading English. If the direction is backward then they operate in the opposite direction through the file. When direction affects the command it is specifically noted. The user may change the direction by typing the appropriate commands.

MOVING COMMANDS

Basic moving commands:

<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 the direction. "<" indicates backward and ">" indicates forward. On entering the Editor the direction is forward. The direction can be changed whenever you type the appropriate command and the "Edit:" prompt line is present. The period and the comma are allowed to change direction 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.

The Editor, for user convenience, maintains the column position of the cursor while you are using <up-arrow> and <down-arrow>, however, 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

JUMP mode is reached by typing "J" while at the Command level. This is indicated on the prompt line by "J(mp". On entering JUMP mode the following prompt line appears:

```
>JUMP: B(eginning E(nd M(arker <esc>
```

You may jump to the beginning of a file by typing "B", to the end of a file by typing "E" or to a marker in the file by typing "M". "B" (or "E") jumps you to the beginning (or end) of the file and displays the edit prompt line. Typing "M" causes the Editor to display the prompt line:

Jump to what marker?

You must enter the name of the marker 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.

Of course to be able to jump to a marker you first must Set the marker. See the SET mode for how to do this.

Page

PAGE command is executed by typing "P" while at the Command level.

PAGE command moves the cursor one whole screenful up or down depending on the direction of the arrow at the beginning of the prompt line. The cursor moves to the start of the line. A <repeat-factor> may be used before this command to go several pages.

Equals

EQUALS command is executed by typing "=" while at the Command level.

EQUALS command causes the cursor to jump to the beginning of the last string which was inserted, found or replaced. 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.

TEXT CHANGING COMMANDS

Insert

INSERT mode is reached by typing "I" while at the Command level. This is indicated on the prompt line by "I(nsrt". On entering INSERT mode the following prompt line appears:

```
>Insert: Text {<bs> a char,<del> a line}  [<etx> accepts, <esc> escapes]
```

As described in section 1.3.2 one of the options here is to type in text followed by <esc> or <etx>. If you have inserted a character which you didn't want, it is possible to get rid of it without leaving the INSERT mode by back-spacing over it. The INSERT prompt line indicates this by "<bs> a char". If you want to get rid of the entire line which you just typed, type <del>. The INSERT prompt line indicates this by "<del> a line".

When you type <return> INSERT starts a new line at the level of indentation specified by the options you have turned on in Environment section of the SET mode. See the section on the SET mode for to how set these options.

Auto-indent:

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 next line at the first position. Note: if Filling is True the first position is what you set as the Left-margin.

Filling:

If Filling is True then the Editor will insist that all your insertions are between the right and left margins by automatically inserting <return>'s between "words" whenever you would have otherwise exceeded the right margin 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 then Auto-indent controls the Left-margin while Filling controls the Right-margin. In any case you can directly change the level of indentation by using the <space> and <backspace> keys immediately after a <return>. Important: you can do this only immediantly after a <return>.

Example 1: With Auto-indent on 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 you make an insertion and accept it that insertion is available for use in the COPY mode. If you enter the INSERT mode and <esc> there is no string available for COPY.

Delete

DELETE mode is reached by typing "D" while at the Command level. This is indicated on the prompt line by "D(lete". On entering DELETE mode the following prompt line appears:

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

When you enter DELETE, the Editor remembers where the cursor is. That position is called the anchor. The object for you is to bracket the text you want to delete between the cursor and the anchor using the normal moving commands. As you bracket text it will disappear from the screen. To accept the deletion, type <etx>; to escape, type <esc>.

If you type <etx> the Editor saves everything which was deleted for COPY to use; if you type <esc> 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. The line 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 prc¸ram 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 we may use the COPY routine to copy the two deleted lines at any place to which we move the cursor.

Figure 3.3

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

Figure 3.4

```
-------------------------------------------------------------
PROGRAM STRING2:
BEGIN
END.
-------------------------------------------------------------
```

Zap

The ZAP command is executed by typing "Z" while at the Command level. This is indicated on the prompt line by "Z(ap".

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 you are zapping more than 80 characters you are asked to verify.

Repeat-factors and Zap: If you do a FIND or a REPLACE 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 you have deleted by using the ZAP command is available for use with the COPY command.

Copy

The COPY command is executed by typing "C" while at the Command level. This is indicated on the prompt line by "C(py".

On executing the COPY command 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 copying 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: If you have accepted a deletion the buffer is loaded with what you have deleted; if you have escaped from a deletion the buffer is loaded with what would have been deleted if you had accepted.

2)INSERT: If you have accepted an insertion the buffer is loaded with what you have inserted. If you have escaped from an insertion the copy buffer is empty.

3)ZAP: If you have used the ZAP command the buffer is loaded with what you have deleted.

The copy buffer is of limited size. Whenever you have deleted so much text that the buffer will not hold all of it. The Editor will warn you upon your typing <etx> with the line:
There is no room to copy the deletion. Do you wish to delete anyway? (y/n)

Exchange

EXCHANGE mode is reached by typing "X" while at the Command level. This is indicated on the prompt line by "X(chng". 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 you type. For example in the file in Figure 3.5 with the cursor at the "W" in WISE, typing "X" to put you in EXCHANGE mode, 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                    Figure 3.6
--------------------          -------------------
WRITE('TOO WISE ');           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 your changes part of the file.

Note: You may not type past the end of the line or type in a carriage return.

FIND AND REPLACE

In both modes the use of a <repeat-factor> is valid. 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 and 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. 1) The terminating delimiter of the string will be the second occurance of the delimiter used as the st ¬ting delimiter. For example: when in REPLACE mode the following command is valid and will replace the first occurance of the character "[" with the character "]": "<[<)])". Here "<" and ")" are the delimiters.

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: If the direction is forward, both modes will operate from the point at which the cursor is toward the end of the file. If the direction is backward both will operate from the point at which the cursor is toward the beginning of the file. If the direction is backward the target pattern will be found if the beginning of the pattern is at or in front of the cursor.

Literal and Token mode: If you are in Literal mode the Editor will look for occurances of the target string. If you are in Token mode the Editor will look for isolated occurances of the target string. The Editor considers a string isolated if it is surrounded by any combination of delimiters. For example, in the expression below the string "HEIGTH" is isolated by the delimiters "=" and "*".

```
AREA:=HEIGTH*WIDTH;
```

If you wish token mode you type "T" after the prompt line and before the target string. If you wish Literal mode you type "L" in the same place. If you do not type either value the mode will be set to the default value found in the Environment. If the default value is Literal the prompt line will remind you that if you wish Token you must type "T" by displaying "T(ok". If the default value is Token the prompt line will remind you that you must type "L" if you wish Literal mode by displaying "L(it". Token mode ignores spaces within strings. In token mode both "(',')" and "(',')" are considered to be the same string.

The Same option: In both commands you may type "S" instead of any of the delimited strings. The "S" indicates to the Editor that it is to use the same string as previously used. 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

FIND mode is reached by typing "F" while at the Command level. This is indicated on the prompt line by "F(ind". 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 occurance of the <target> string starting with the current position and going in the current direction. The number "n" is the <repeat-factor> and is shown on the prompt line in the brackets "[]". The arrow at the beginning of the prompt line always gives the current set direction.

Example 1: In the STRING1 program with the cursor at the first "P" in PROGRAM STRING1 Type "F". Then when the prompt appears type "'WRITE'". YOU must type the single quote marks. The prompt line should now appear as:

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

When you type the last quote mark the cursor will jump 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

REPLACE mode is reached by typing "R" while at the Command level. This is indicated on the prompt line by "R)place". On entering REPLACE mode one of the two prompt lines in Figure 3.11 appears. In this example we have assumed that a <repeat-factor> of four was entered.

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 SIZEQX:INTEGER;" to "VAR SIZEYZ:INTEGER;". If we had not been in a literal mode it would not have found the string QX because it is not a token. It was part of the token SIZEQX.

Example 2: In Token mode REPLACE ignores spaces between tokens when looking for patterns to replace. For example if you had the lines on the left hand side of Figure 3.9 and you typed: "2RT/(',')/.LN." The prompt line should appear as:

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

and immediately after you typed the last period it would change those two lines to those on the right hand side of Figure 3.10.

Figure 3.10

```
-----------------------------------------------------------------------
    WRITE(',');                                   WRITELN;
    WRITE( ',');                                  WRITELN;
-----------------------------------------------------------------------
```

V)fy: The verify option permits you to examine each occurance of the <targ> string (up to the limit set by the repeat factor) and decide if that occurance 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; typing a space will cause the REPLACE mode to search for the next occurance provided the repeat factor has not been reached. The repeat factor counts the number of times an occurance is found, not the number of times you actually type "R".

FORMATTING COMMANDS

Adjust

ADJUST mode is reached by typing "A" while at the Command level. This is indicated on the prompt line by "A(djst". 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 you may use the <right-arrow> and <left-arrow> commands to move the whole line. Each time you type a <right-arrow> the whole line moves one space to the right. Each <left-arrow> moves it one to the left. When you have the line adjusted to the desired indentation press <etx>. You cannot <esc> from this mode.

In writing a PASCAL program you may find yourself with a whole sequence of lines to adjust. For example when you find you need an additional BEGIN -END sequence you will want to adjust all the lines in between two spaces to the right. This is done easily. Adjust one line, then use <up-arrow> (<down-arrow>) commands and the line above (below) will be automatically adjusted by the amount of adjustment on the line from which you came.

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

Example: Starting with the cursor anywhere in the line "WRITE('TOO WISE ');" of the program shown in Figure 3.11, type the series of commands: "A",<right-arrow>,<down-arrow>,<down-arrow>, "3"<right-arrow>,"2",<down-arrow>,<etx>. The adjusted text is shown in Figure 3.12.

Figure 3.11

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

Figure 3.12

```
---------------------------------------------------------------------
PROGRAM STRING2;
BEGIN
   WRITE('TOO WISE ');             A <right-arrow> adjusts this line right one
   WRITE('YOU ARE');      Adjusted by amount of the above line on <down-arrow>
      WRITELN(',');      Adjusted one by line above and 3 by "3",<right-arrow>
      WRITELN('TOO WISE ');   Adjusted four by line above on "2",<down-arrow>
      WRITELN('YOU BE')                Also adjusted four by "2",<down-arrow>
END.
---------------------------------------------------------------------
```

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.

<u>Margin</u>

MARGIN command is executed by typing "M" while at the Command level. MARGIN is an Environment dependent command, that is, it may only be executed when Filling is set to True and Auto-intent 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                Left-margin 10
Right-margin 72              Right-margin 70
Paragraph-margin 8           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 occuring between two blank lines. to MARGIN a paragraph move the cursor to anywhere in that paragraph and type "M". If you are doing an exceptionally long paragraph it may take several seconds before the routine is ready to redisplay the screen.

Command Characters: 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.

MISCELLANEOUS COMMANDS

Set

SET mode is entered by typing "S" while at the Command 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 you are editing it is particularly convenient to be able to jump directly to certain places in a long file. The Editor enables you to set markers into your file at places of your choosing. Once you have set these markers it is possible to jump to them using the M(arker option in the JUMP mode. When in the SET mode you type "M" for M(arker, the following prompt line appears:

Name of marker?

At this point you will wish to enter the name of the marker. You may enter any string followed by a <return>. The marker will be entered at the position of the cursor in the text; therefore, first move the cursor to the desired position then set the marker. (If the marker already existed, it will be reset.)

Figure 3. 15

```
PROGRAM STRING1
BEGIN
  wRITE('TOO WISE ');
END.
```

Example: With the cursor at the position shown in Figure 3. 15 and with the Editor at the Command level, the following commands illustrate the setting of a marker. The lower case character represents the position of the cursor.

You type:	Prompt line displays:
	>Edit: A(djst......
"S"	>Set: M(arker E(nvironment <esc>
"M"	Name of Marker?
"* GO<ret>"	>Edit: A(djst......

After the <return> the cursor goes back to its original place in the text and the marker has been set. Having set the marker, you may JUMP from any place in the file to the marker named "* GO" and the cursor will move directly to the "W" in "WRITE".

E(nvironment:

There are several different uses for editing on the computer. Text editing and program editing are the two chief ones. Certain options make it more convenient to write programs while other options make it more convenient to edit text. The Editor enables the user to set the environment which the user determines to be most convenient for him. When in the SET mode you type "E" for E(nvironment, the screen display is replaced with the following prompt shown in Figure 3. 16.

Figure 3.16

```
---------------------------------------------------------------------
>Environment: {options} <etx> or <sp> to leave
     A(uto indent  True
     F(illing      False
     L(eft margin  0
     R(ight 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 you may change any or all of the options. The options shown are the default options which you have upon entering the Editor on the Terak 8510A. Implementations for other machines may have different defaults.

The Options:

A(uto indent:

Auto-indent affects only the INSERT mode of the Editor. Auto-indent may be set to True (turned on) by typing "A","T". Auto-indent may be set to False (turned off) by typing "A","F".

F(illing:

Filling affects the INSERT mode and allows the MARGIN command to function. Filling is set to True (turned on) by typing "F","T". It is set to False by typing "F","F".

L(eft margin
R(ight 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. End the positive integer with a <space>. The positive integer you typed should replace the old value for the L(eft margin in the prompt shown in Figure 3.16. Setting new values for the other margins is done similarly using the letters "R" and "P". All positive integers with less than four digits are valid margin values.

As an example you could set the options to the following values:

```
    L(eft margin  10
    R(ight margin 60
    P(ara margin  0
```

C(ommand ch:

The Command character affects the MARGIN command and the Filling option in the INSERT mode as described in those sections. You may 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.

T(oken def:

This option affects FIND and REPLACE. Token is set to True by typing "T","T" and to False by typing "T","F". If Token is True then Token is the default. If Token is False then Literal is the default.

Verify

The VERIFY command is executed by typing "V" while at the command level. Verify permits you to verify the status of the Editor by causing the screen display to be updated. The Editor attempts to adjust the window so that the cursor is at the center of the screen.

Quit

QUIT mode is reached by typing "Q" while at the Command level. This is indicated on the prompt line by "Q(uit". 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
----------------------------------------------------------------------
```

You must select one of the four options by typing U,E,R or W.

U(pdate:

This causes you to leave the Editor after writing the file you have just modified into the workfile. The file is stored as SYSTEM.WRK.TEXT and 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.

E(xit:

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

R(eturn:

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

W(rite:

This option puts up a further prompt:

Figure 3.18

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

You are able to cause the Editor to write the modified file to any file name of your choosing. If you cause it to write to the name of an existing file the modified file will replace the old file. After the file has been written to the disk, the EDITOR will ask you to enter E(xit or R(eturn. If you R(eturn the cursor is returned to the exact place of typing "Q". This command can be aborted by typing a carriage return instead of a file name. Aborting will return you to the Editor.

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

<u>Reference Section</u>

```
<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
```

A(djust: 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.

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

D(elete: Treats the starting position of the cursor as the anchor. Use any moving commands to move the cursor. <etx> deletes everything between the cursor and the anchor.

F(ind: 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 <del> to reject part of your insertion.

J(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)pdate, 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.

eX(change: Exchanges the current text for the text you type in while in this mode. You can only do one line. <back-space> cause 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.

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

Version I.4   January 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 interpretted 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 reenter 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.)

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 by 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 TERAK 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.

COMMAND ARGUMENTS

A commmand 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.

0   '0' 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.

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      GSTRING$$
```

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.

THE TEXT BUFFER

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

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

THE CURSOR

The "cursor" is a logical entity which is where, in your text, the next command will happen. 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.

INPUT/OUTPUT COMMANDS

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

The L(ist 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

```
*W<file title>$
```

File title is any legal file title as decribed 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 by 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. If it can't find it it appends a '.TEXT' and tries again.

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

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

Executing the QU 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 QE to exit the editor.

The QR command is used on the occasions when a Q 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.

CURSOR RELOCATION COMMANDS

J(ump, A(dvance, B(eginning, G(et, 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, they search the current Text Buffer 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.
```

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'. You should then or as soon as possible type two successive <esc>s. To continue, type I to go back into 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 is left at the first character following the deleted text.

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

*B$FHOSE $=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 is left at 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 is left immediately following the changed text.

*0CAPPLES$$ Replaces the characters from the beginning of the line up to the cursor with 'APPLES', (equivalent to using 0X).

*BGHOSE$=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 is left positioned after 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'.

***0XTEXT$$** Exchanges the current line from the beginning to the cursor with the string 'TEXT', (equivalent to using 0C).

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

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 is left positioned 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 0U.

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'i, 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 specfied 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.

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:(eturn 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 -

```
************************ ***************
* INTERACTIVE DEBUGGER * * Section 1.5 *
************************ ***************
```

Version I.4   January 1978

To facilitate the debugging of Pascal programs, an interactive debugger is included in the system. In order to use it is recommended that two compiler options be turned on in your program.

The first is "D+" which causes conditional halts to be generated. These halts are necessary for use of the Crawl, Walk or Breakpoint commands. The other is "L+", which causes a compiled source listing of your program to be written to your disk. The debugger uses this file, SYSTEM.LST.TEXT, while in the CRAWL and WALK mode or when a breakpoint is executed. "D+" causes a slightly larger codefile to be created, "L+" requires space on disk, but these options can be turned on and off repeatedly, allowing one to surround troublesome pieces of code, without generating extremely large code or listing files.

Sample program to be debugged:

```
 1    1     1:D      1 (*$D+,L+*)
 2    1     1:D      1 PROGRAM BUG;
 3    1     1:D      3 VAR I:  INTEGER;
 4    1     1:D      4
 5    1     2:D      1 PROCEDURE DIVO;
 6    1     2:D      1 VAR J:  REAL;
 7    1     2:C      0 BEGIN
 8    1     2*C      0   J := 5 / I;
 9    1     2*C     11 END    (* DIVO *);
10    1     2:C     26
11    1     1:C      0 BEGIN
12    1     1*C      0   I := 0;
13    1     1*C      7   DIVO;
14    1     1*C     11 END    (* BUG *).
```

The source listing above of program BUG is interpreted as follows: The first number of each line is the line number, the second is the segment number and the third is the procedure number. The letter after the colon or star indicates whether the offset represents a code ('C') or data ('D') offset. When a star '*' appears after the procedure number it means that that line has at least one conditional halt associated with it; otherwise a ':' appears. If the offset is a code offset, it represents the offset in the code segment for that procedure of the first instruction generated for that line. If the offset is a data offset the number represents the word offset in the data area where storage for that line of the procedure begins.

To use the debugger type D(ebug instead of R(un at the system command level. The program will be compiled if necessary, and the debugger will print a message with the release number and date of release.

PASCAL INTERACTIVE DEBUGGER -January 1978

You will be in the EXAMINE mode and its prompt will be displayed. In EXAMINE mode you can peruse portions of memory, set or clear breakpoints, resume execution or exit your program. You may also enter WALK or CRAWL mode in which your program will be executed one statement at a time. To begin execution you have three options:

1) R(esume -runs program normally until a BREAK or breakpoints are encountered or a non-fatal run time error occurs.

2) C(rawl -executes program on a one statement at a time basis, waiting for input from you between steps.

3) W(alk -executes program one statement at a time at an adjustable rate.

CRAWL MODE

Prior to execution of a statement, information about that statement is displayed. If SYSTEM.LST.TEXT exists, the compiled listing line containing that statement is displayed, otherwise the line #, the segment and procedure # the statement is in, and the code offset of the first instruction of the statement is printed. You then have two options:

1) type [space] if you wish to execute the line and continue
2) type 'Q' to leave the CRAWL mode and enter the EXAMINE mode

WALK MODE

This mode is similar to CRAWL in that information about each statement is displayed just prior to execution of that statement. When you type 'W' to go into WALK mode a prompt will appear on the top line of the screen.

DELAY:

An integer should then be entered, and the debugger will use this as the number of seconds (on an LSI-11) to delay between executing each statement in the program. Use the BREAK key (sometimes unreliable in PDP-11 systems) to get back to the EXAMINE mode.

EXAMINE MODE

NOTE: To use this mode and to aid in debugging a program with the debugger it is almost a necessity to have a compiled source listing of the 'bugged' program in order to find names of variables with their offsets, the numbers belonging to procedures and the code offsets for each line in the program.

In EXAMINE the following will be displayed when the divide by zero occurs in the program BUG. (One way to get to this point is to use the R(esume command and then type [space] when prompted to do so.)

```
>EXAMINE: 1..9 (links, M(ove, <, >, L(ink, D(ata, S(tack, H(eap
E(rase, U(pdate, <ctrl-U(p>, <ctrl-D(own>, C(rawl, W(alk, R(esume, <esc>

Proc   2    Caller   1    Parent   1    Param   0    Data     2
 Seg   1       Seg   1       Seg   1    Stack  10     IPC     9    Depth  0
Brkpnts:                                Defaultlink =  Dynamic
TYPE      ID     PROC# OFFSET    ADDR   INTEGER   OCTAL   HEX   LO  HI   CHAR
Data                 2      1  116560    -25228  116564  9D74  164 235    t?
Data                 2      2  116562         0  000000  0000  000 000    ?
     *
```

Floating point overflow/underflow

The cause of entering the EXAMINE mode is displayed on the bottom line of the screen. It is either some type of execution error, a user break, termination of CRAWLING or WALKING mode, or execution of a breakpointed statement. In this case it was a floating point error (a divide by zero to be specific).

The procedure in which the error occurred is given by PROC and the SEG# below it. Here there are 2 and 1 respectively. This can be seen from looking at the source listing to be procedure DIVO. The caller of DIVO and its parent are the same: the main body of BUG. In BUG, as in most user programs, the main body is procedure #1, in seg#1. When you are moving up and down the dynamic or static chains PROC and SEG refer to the procedure you're at in the dynamic chain (the current procedure) and CALLER and PARENT are in reference to this procedure.

The amount of memory (in words) allocated for this procedure's PARAMeters is 0, the STACK size is 10 words (The current release may be inaccurate) and the DATA segment is 2 words. The STACK portion is the evaluation stack where values are put during expression calculations by the compiler. The DATA area is used for storage of declared variables and any temporary variables generated. For more detailed explanations refer to section 3.5 in the documentation, INTRODUCTION TO THE PASCAL PSEUDO-MACHINE. The procedure DIVO has one local variable, R: REAL declared which takes two words in memory because it is REAL. By looking at the source listing you can see that J is stored starting at offset 1, and then by looking at offsets 1 and 2 in the memory display area of the screen you can see the value of J. To determine just where the value of a variable is stored (i.e. its offset) one must understand the algorithm the compiler uses. Consider, for example the following declaration:

```
VAR I, J, K: INTEGER;
```

Variable K would be stored at offset 1 (not 3), J would be at offset 2 and I at offset 3.

For this declaration

```
VAR A: ARRAY[0..2] OF INTEGER;
```

A[0] would be at offset 1, A[1] at offset 2 and A[3] at 3.

Parameters are always stored directly from left to right. Functions values always occupy offsets 1 and 2.

IPC is the interpreter program counter and by looking at this number and searching for it (or the number that's closest to it but still smaller) in the source listing you will have the line in which the error occurred. In this case the IPC is 11 which corresponds to line 8 in program BUG.

DEPTH tells you where the current procedure is in the 'call' chain with respect to the procedure in error (which is always at the bottom of the call chain and therefore at depth 0). In other words, DEPTH is the number of dynamic links above the halted procedure.

DEFAULTLINK is the type of LINK you will traverse when using the traversal commands. A DYNAMIC link points to a procedure's caller and a STATIC link to its parent.

The '*' shows where the D(ata, S(tack, or H(eap command will write when used.

COMMANDS IN EXAMINE MODE

1) # links -entering a number between 0 and 9 will move you that many links up or down the dynamic or static chain. The direction in which you will go is determined by the first character of the EXAMINE prompt. '>' indicates traversal will be in the direction of older calls (if dynamic) or ancestors (if static); '<' indicates traversal towards more recent calls. Note: Traversal towards descendants is disallowed. The type of links you will traverse, STATIC or DYNAMIC, is specified to the right of DEFAULTLINK.

2) <esc> -typing this will return you to the system command level.

3) L(ink -this command toggles the DEFAULTLINK from DYNAMIC to STATIC and vice-versa.

4) D(ata -used for examining the DATA and parameter segment of a procedure. This command has 4 parameters that can be specified if you don't want to use their default values. The debugger will prompt you for them. Typing a [cr] at any point tells the debugger to use the default values for the rest of the parameters. Typing a [space] delimits a parameter and lets the debugger prompt you for the next one.

a) OFFSET: default value is last offset displayed plus 1. Beginning value is 1. You change the offset by entering an integer.

b) LENGTH: beginning default value is the minimum of the Buffer size for the memory display (15 for 24 line screens) and DATA plus PARAM. After that it is the last length specified in a D(ata or S(tack command. LENGTH determines the number of words to be displayed.

c) PROC: the number of the current procedure is the default value. Any procedure that is up the call chain from PROC may be specified.

d) SEG: default value is the segment the current procedure belongs to. Enter the value of the segment you want if it's not the default.

If the debugger finds the specified procedure it will display the data, wrapping around to the top and erasing information in the memory display buffer if necessary. When an offset displayed is larger than PARAM plus DATA for a procedure the message

Warning - offset too large

will appear on the bottom line of the screen and the invalid data will not be displayed.

5) STACK -this command is used for examining the stack area belonging to a specified procedure. Parameters are specified in the same manner as in the D(ata command, but the first offset is 0 not 1.

6) MOVE -a command used to find a specified procedure and make it the current procedure. This command has two parameters:

a) PROC-procedure number of the desired procedure. Default is the number of the bombed procedure (the one at the bottom of the call chain). To use the default just type [ret], and the normal search described below will be by-passed. Otherwise enter an integer. Type [ret] now to use the default segment number SEG, otherwise type [space].

b) SEG-segment number where desired procedure resides. Default segment number is that of the current procedure.

Once the parameters have been set the debugger will then search up the dynamic links starting at the caller of the current procedure. (This implies you can never move to the current procedure since the debugger won't find it.) If the specified procedure is found it becomes the current procedure and the information in the prompt line will be updated, otherwise the current procedure remains unchanged.

7) RESUME -resume normal execution of the program where the debugger was invoked.

8) CRAWL -resume execution of the program in CRAWL mode at the point in the program where the debugger was invoked.

9) WALK -resume execution of program in WALK mode starting where the debugger was invoked.

10) UPDATE -refreshes the memory display buffer. S(tack, D(ata and H(eap commands save the procedure numbers and offsets displayed in the memory buffer. When 'U' is typed the buffer is erased and the saved numbers are used to look up the information belonging there. If any of the information belongs to procedures that are below the current procedure in the call chain then UPDATE will not be able to refresh that part and will say

Proc not found

11) ERASE -clears the memory display buffer on the screen.

NOTE: neither the UPDATE nor ERASE commands affect main memory but simply the memory display buffer.

12) '<' or ',' -changes the direction of link traversal to be down the call chain, i.e. go towards the callees

13) '>' or '.' -changes the direction of link traversal to be up the call chain, i.e. go towards callers

14) HEAP -asks for an octal address, and a length. That portion of memory will then be displayed.

15) <CR> -clears the line with the '*' and moves down one line.

16) <CTRL-U> -moves the '*' up one line.

17) <CTRL-D> -moves the '*' down one line.

18) BREAKPOINT -asks if you want to S(et or C(lear a breakpoint.

SET: asks for a line number. Enter a line that has a '*' in the compiled listing. Whenever a statement in that line is about to be executed, the debugger is called.

CLEAR: asks for a line number. Enter <cr> to clear all breakpoints, or the line number of an active breakpoint.

This final section of the document will describe how one could use the commands described above to diagnose the fault (floating point error) that occurred in program BUG.

We know that the error occurred on line 8

```
J := 5 / I;
```

and the cause was a floating point error. With the debugger we might want to look and see just what the value of I is. From the source listing one can see that I is stored in procedure #1, segment #1 at offset 3. Let us use the M(ove command to go to procedure 1 and then the D(ata command to look at the value of I.

1) type 'M' for move, then a 1 for procedure 1 and then [ret] since the default segment # is 1 which is what we want.

```
Proc: 1[ret]
```

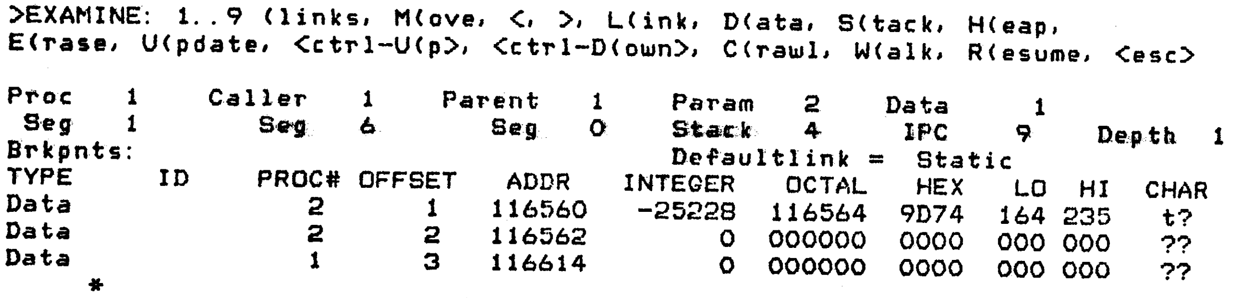
2) the three line description of the current procedure will be updated as shown below, the memory display buffer is unchanged so far.

3) type [ret]. This will enter a blank line in the memory display buffer. This step is not necessary.

4) type 'D' for data, enter 3 for OFFSET then [space], 1 for LENGTH and then [ret].

```
Offset: 3  Length: 1[ret]
```

The screen will look as follows:



```
>EXAMINE: 1..9 (links, M(ove, <, >, L(ink, D(ata, S(tack, H(eap,
E(rase, U(pdate, <ctrl-U(p>, <ctrl-D(own>, C(rawl, W(alk, R(esume, <esc>

Proc    1     Caller   1     Parent   1     Param    2     Data      1
 Seg    1        Seg   6        Seg   0     Stack    4     IPC       9     Depth  1
Brkpnts:                                    Defaultlink =  Static
TYPE      ID      PROC# OFFSET    ADDR    INTEGER    OCTAL    HEX    LO  HI    CHAR
Data                  2      1  116560     -25228   116564   9D74   164 235    t?
Data                  2      2  116562          0   000000   0000   000 000    ??
Data                  1      3  116614          0   000000   0000   000 000    ??
     *
```

An alternate way to look at the value of I would have been to just use the D(ata command and specify 1 for the procedure number rather than using the default value. Note that this would not change the value of the current procedure as the above method does.

```
Offset: 3  Length: 1  Proc: 1[ret]
```

Things to note:

PARAM is equal to two. This is because the system predeclares the two parameters INPUT and OUTPUT for you.

The integer value of I is zero just as it should be.

- Notes -

```
******************* ***************
* PASCAL COMPILER * * Section 1.6 *
******************* ***************
```

Version I.4   January 1978

The U.C.S.D. Pascal compiler is invoked by using the C(ompile command of the outermost level of the U.C.S.D. Pascal system. It assumes you have a workfile, either created by the editor, or G(otten in the F(iler.

The U.C.S.D. Pascal complier is a one-pass recursive descent compiler. It generates codefiles to run directly on the Pascal interpretive machine. The compiler is based on the P2 portable compiler from Zurich.

Unless the SLOWTERM Boolean inside of the system communication record SYSCOM^ 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 Q+ compiler option, which is dicussed in the section of this document entitled "Compile Time Options" which appears below). Below is an example of the output which appears on the CONSOLE device:

```
PASCAL compiler [I.4]
<   0>....................
P1
<  19>............................................
P2
<  61>............................................................
TEST
< 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 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 were detected, the the compiler will write a codefile onto the disk called *SYSTEM.WRK.CODE. This is the codefile which is executed if the user then types the R(un command. (For further details on the system commands, see INTRODUCTION AND OVERVIEW Sec. 1.1.)

Should the compiler detect a syntax error, the text surrounding the error and an error number together with the marker '<<<< ' will point at the symbol in the source where the error was detected (unless both the Q and L options are set, in this case the compilation will continue, with the syntax error going to the listing file, and the console remaining undisturbed). The compiler will the 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.

There are a few syntax errors added to the U.C.S.D. compiler which are not listed on pages 119-121 in Jensen and Wirth. A list of these additional syntax errors appears in TABLE 5. All error numbers will be accompanied by a textual message upon entry to the editor.

```
************************ *****************
* COMPILE TIME OPTIONS * * SECTION 1.6.1 *
************************ *****************
```

Compile time options in the U.C.S.D 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 U.C.S.D.'s compiler control comments is essentially as described in Jensen and Wirth. However the actual options and the letters associated with those options bear only occasional resemblance to the options listed on pages 101 and 102 of Jensen and Wirth. Also, if a '+' or '-' didn't appear after an option letter, '+' is assumed. The following sections describe the various options currently available to the user of the U.C.S.D. Pascal compiler.

D:

This option is used to cause 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 of this documentation entitled "DEBUGGER" for details)

Default value: D-

D-: causes the compiler to not emit 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.

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 '-' then the control comment will affect the boolean variable IOCHECK within the compiler. The alternative use of 'I' in a compiler control comment is to cause the compiler to include a different source file into the compilation at this point. The syntax of this include-file mechanism will be discussed after the discussion of the IOCHECK option.

(IOCHECK OPTION)

Default value: I+

I+: instructs the compiler to generate code after each statement which performs any I/O, which checks 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 check 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 G+, or L+, etc. can follow the filename. Note that if you have a file name which starts with '+' or '-' as the first character of the filename, you must insert a blank 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 at opening the include file fails, the compiler will concatenate a ".TEXT" onto the file-name and try again. If this second attempt fails, or some I/O error occurs at some point while reading the include file, the compiler will respond with a fatal syntax error.

The compiler will also relax the requirements of the order in which declarations must be made for included files, so that it is possible to 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 will also result 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 ɔmpiled 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*)
```

See the section of this document describing the include file mechanism for a complete description of the syntax for specifying a file-name inside of a control comment.

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 outputing a "C" for code and a "D" for declaration. 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 tell you 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 then the compiler will output addtitional code to perform checking on array subscripts and assignments to variables of subrange types.

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 set.

U:

This option sets the boolean variable SYSCOMP in the compiler. This boolean variable 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 you might expect them to. Not recommended for non-systems work without knowing why it does what it does.

- Notes -

```
************************ ***************
* UCSD BASIC COMPILER * * Section 1.7 *
************************ ***************
     Version I.4     January 1978
```

Introduction

This document has been designed for programmers who are already familiar with Basic. The intent of this document 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.

This document is divided into three sections. The first contains a brief description of the features included in UCSD Basic. The second contains the descriptions of the features unique to UCSD Basic. The third contains 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 of this document, all of them are noted or recapped in the second section.

The UCSD BASIC Compiler is invoked like a user program (with the eXecute command: X BASIC.COM). It immediately prompts for the name of the source file to be compiled. If no file is given (ie. an immediate carriage return response to the prompt), the current workfile is used as the source. Next the compiler prompts for a codefile name. This file will contain the results of the compilation and can be executed like any other user program, when the compiler has completed its translaion.

A Basic Description of Features Included

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.

Variable Names

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

Intrinsic Arithmetic Functions

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.

Intrinsic String Functions

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 occurance 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 anu 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. If you wish to have a file created by new to be editable with either of the system editors, you must append '.text' to the file title.

These functions return IORESULT as described in section 2.1.

Programming Statements

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 occurances of the enclosing quotation mark prints a single mark of that type.

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

Each execution of the loop increments the loop counter "var" by the amount of expression three. 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; if the variable is a real, the expression must evaluate 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 consistant 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.

Unique Features of UCSD Basic

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.

Intended Features

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.

To run a BASIC program

Create the BASIC program using one of the system text editors. Execute the file BASIC.COM, you will be asked for an input file, typing carriage return with no file name will cause the BASIC compiler to assume the workfile. You will also be asked to specify an output file, typing carriage return with no filename will cause the BASIC compiler to generate its output to the filename B. If your program compiles with no syntax errors, you can run it by eXecuting the code file generated by the basic compiler. If there are syntax errors in your program the ensuing steps should be obvious.

```
********************* ***************
* SYSTEM INTRINSICS * * Section 2.1 *
********************* ***************
```

Version I.4   January 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 particularily 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 {metabrackets} 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 expres on 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:
BOOLEAN 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 9600 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]

```
********************  ****************
* STRING INTRINSICS * * Section 2.1.1 *
********************  ****************
```

Version I.4   January 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';
PATTORN := 'TAL';
WRITELN(POS(PATTORN,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('IO',ID));
WRITELN(ID);
```

Will print:

```
INSERT DEMONSTRATIONS
```

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 <u>which means it probably becomes wrong</u>. 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 I.4   January 1978

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

These procedures open files for reading and writing. They 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 IORESULT. 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 ',,' will be sufficient. (See UNITBUSY and UNITWAIT.)

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[0],
                     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[0]);
```

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 for now is undefined as to what it will do..... The intent is to lock a file with the minimun 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 CLOSEed 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. While doing puts or writes at the end of a file, if the file cannot be expanded to accomodate 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 EOLN(FILEID) true. This means that any subsequent READs of STRING variables will return the null string until a READLN or READ(chararacter) is executed.

There are three files of type INTERACTIVE which are predeclared for you: 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 his output.

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

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

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

This procedure changes the file pointers so that the next GET or PUT from/to the file will happen to the INTEGERth record of FILEID. Records in files are numbered from 0. A GET or PUT <u>must</u> 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.

```
******************* *****************
* TURTLE GRAPHICS * * Section 2.1.3 *
******************* *****************
      Version I.4     January 1978
```

Section 2.1.3 is intentionally missing from the I.4 set of documents. Turtle Graphics are not ready for general release as of Version I.4; however, we do have a Turtle Graphics package for Terak 8510a users, which may be obtained upon special request. We plan in some future release to have a Turtle Graphics package which will be modifiable for any graphics screen.

Thank you for your patience.

- Notes -

```
*********************************** ***************
* LOW LEVEL GRAPHICS INTRINSICS * * Section 2.1.4 *
*********************************** ***************
```

<u>CAUTION</u>:

These routines do no range checking of the parameters they are passed. If any of the paramters are "out of range", these routines will happily move bit patterns throughout main memory, much to the dismay of the operating system and your program.

See Table 4 for modes and penstates for these intrinsics.

The DRAW intrinsics are available only for the Terak 8510a in this release. Additional display units will be supported in later releases, but no details are currently available. Probable implementation(s): Tektronix 4006.

```
PROCEDURE DRAWBLOCK ( SOURCE, SCREEN, ROWSIZE, STARTX, STARTY,
                      SIZEX, SIZEY, COPYX, MODE );
                  (* none of these are optional *)
```

This procedure is written for the Terak 8510a graphic display mode. The TERAK screen displays words consecutively with the most significant bit of the word on the right. DRAWBLOCK will work only on screens whose graphics operates in this manner. WARNING: No range checking is performed.

DRAWBLOCK transfers a bit matrix SOURCE, which starts on an word boundary, to a specified point (STARTY, STARTX) in the bit matrix SCREEN. All parameters are integers except SCREEN, which is a bit matrix of width ROWSIZE (i.e. BITMAP:PACKED ARRAY[0..MAXROW] OF PACKED ARRAY[0..ROWSIZE-1] OF BOOLEAN;). The SOURCE is SIZEX bits wide by SIZEY bits high. The first COPYX bits of each row are copied into the destination. MODE is defined in TABLE 4.

```
PROCEDURE DRAWLINE ( RANGE, SCREEN, ROWWIDTH, XSTART, YSTART,
                     DELTAX, DELTAY, PENSTATE );
                 (* none of these are optional *)
```

In order the parameters are: INTEGER IDENTIFIER, ARRAY IDENTIFIER, and (the remaining six) INTEGER EXPRESSION. RANGE will contain the results of a Radar scan. This parameter is untouched unless PENSTATE is sent as 4. The value returned is the number of dots that would have been drawn before encountering an obstacle. SCREEN may be subscripted to determine a starting position in the array. ROWWIDTH is the width of SCREEN in number of words; this determines how DRAWLINE will consider the rectangularity of the array. XSTART is the starting horizontal coordinate; YSTART is the starting vertical coordinate. DELTAX is the distance to move in the horizontal plane. DELTAY is the distance to move in the vertical plane. PENSTATE controls the action taken; see TABLE 4.

```
****************************************** ****************
* CHARACTER ARRAY MANIPULATIONS INTRINSICS * * Section 2.1.5 *
****************************************** ****************
```

Version I.4   January 1978

CAUTION:

These intrinsics are all byte oriented. Use them with care; read the descriptions carefully before trying them out. No range checking of any sort is performed on the parameters passed to these routines. Therefore 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 PTERADOCTYLE 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 end of both arrays.

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 ARAY: PACKED ARRAY [1..30] OF CHAR;

          (*123456789a123456789b123456789c*)
   ARAY:  |THIS IS THE TEXT IN THIS ARRAY|
             MOVERIGHT(ARAY[10],ARAY[1],10);
   ARAY:  |NE TEXT INE TEXT IN THIS ARRAY|
             MOVELEFT(ARAY[1],ARAY[3],10)
   ARAY:  |NENENENENENENETEXT IN THIS ARRAY|
             MOVELEFT(ARAY[23],ARAY[2],8);
   ARAY:  |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.

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 PWROFTEN (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; if the Debugger is not in core when this occurs, therefore, 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.

* DIFFERENCES BETWEEN U.C.S.D. PASCAL AND STANDARD PASCAL* * Section 2.2 *

Version I.4 January 1978

This document is a description of the various differences between Standard Pascal and U.C.S.D. Pascal. The Standard Pascal referred to by this document is defined in PASCAL USER MANUAL AND REPORT (2nd edition) by Kathleen Jensen and Niklaus Wirth (Springer-Verlag, 1975).

This document is intended to act as a summary and quick referance guide which mentions 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.

Many of the differences mentioned above lie in the area of FILES and I/O in general. It is recommended that the reader first concentrate upon the sections of this document which describe the differences associated with the standard procedures EOF, EOLN, READ, WRITE, RESET, and REWRITE.

SUMMARY OF TOPICS IN THIS DOCUMENT

1. CASE STATEMENTS

2. COMMENTS

3. DYNAMIC MEMORY ALLOCATION

4. EOF

5. EOLN

6. FILES

   A. INTERACTIVE FILES
   B. UNTYPED FILES
   C. RANDOM ACCESS OF FILES

7. GOTO AND EXIT STATEMENTS

8. PACKED VARIABLES

   A. PACKED ARRAYS
   B. PACKED RECORDS
   C. USING PACKED VARIABLES AS PARAMETERS
   D. PACK AND UNPACK STANDARD PROCEDURES

9. PARAMETRIC PROCEDURES AND FUNCTIONS

10. PROGRAM HEADINGS

11. READ AND READLN

12. RESET

13. REWRITE

14. SEGMENT PROCEDURES

15. SETS

16. STRINGS

17. WRITE AND WRITELN

17.5 EXTENDED COMPARISONS

18. MISC. IMPLEMENTATION SIZE LIMITS

19. SUMMARY OF U.C.S.D. INTRINSICS

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, then 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 116-11U 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. 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 of this documentation entitled "Pascal Compiler" for details on compiler control comments.)

Note that if the beginning of the comment is delimited by the "(*" symbol, then the end of the comment must be delimited by the matching "*)" symbol, rather than an occurance of the "}" symbol. In the case where the reverse is true, (i.e. 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 it encounters a comment symbol, it scans the text for the matching comment symbol. i.e. the following text will result in a syntax error:

```
(* THIS IS A COMMENT  (* NESTED COMMENT *)  END OF FIRST COMMENT *)
                                              ^error here.
```

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 as described below is only an approximation to the function of DISPOSE in that 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 cause changes 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. The fact that HEAP was declared to be a pointer to an INTEGER is not really important. In fact 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 as is traditional. (Declaring a pointer variable to be a pointer to an INTEGER proves to be a particularly handy construct for deliberately interferring with the contents of memory which is otherwise inaccessable). Below is a pictorial description of the heap at this point in the program's execution:

```
TOP OF HEAP --> !______________________! <--- HEAP
                !                      !
                !                      !
                !                      !
                !  contents of heap at !
                !    start of program  !
                !                      !
```

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:

```
TOP OF HEAP ---> !______________________!
                 !                      !
                 !          P^          !
                 !______________________! <--- HEAP
                 !                      !
                 !  contents of heap at !
                 !    start of program  !
                 !                      !
```

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 done a series of calls to the standard procedure NEW between the calls to MARK and RELEASE, then the effect would have been that the storage occupied by several variables would be released at once. Also 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 because of this that 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" which point to areas of memory which are no longer a part of the defined heap space.

4. EOF(F)

To set EOF to TRUE for a textfile F which is 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 of this documentation)

For any FILE F, if F is closed, then EOF(F) will return the value TRUE. If EOF(F) is TRUE , and F is a FILE of type TEXT, then 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,...) then the data thereby obtained is not valid.

When a user program starts execution, the system performs a RESET on the predeclared files INPUT, OUTPUT, and KEYBOARD. (See section of this document on the procedure 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.

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 in regard to 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 to cause EOLN(F) to become 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 you type a space, followed by the carriage return, then EOLN will remain FALSE and another READ will take place.

6. FILES

In regard to the subject of files, the I.4 release contains several changes from the I.3 release. These changes were made in order to bring U.C.S.D. Pascal closer to the standard definition of the language. These changes include the addition of a new file <type> called INTERACTIVE (described in section A below) and the introduction of the use of the standard predeclared identifiers RESET and REWRITE as synonyms for the U.C.S.D. intrinsics OPENOLD and OPENNEW.

As mentioned in the above paragraph, the I.4 Pascal compiler will continue to support the use of OPENOLD and OPENNEW by treating OPENOLD as being equivalent to RESET and OPENNEW as being equivalent to REWRITE. In later releases the predeclared identifiers OPENOLD and OPENNEW will be "phased- out" and RESET and REWRITE used in their place. (See the sections of this document entitled "RESET" and "REWRITE" for further details)

A. INTERACTIVE FILES

As of the I.4 release of the system, a new predeclared file kind INTERACTIVE has been added. Files of <type> INTERACTIVE behave exactly as files of <type> TEXT were defined in previous releases of the system. The standard predeclared files INPUT and OUTPUT will allways 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. Additionally, 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 a more detailed discussion of files of <type> INTERACTIVE see the section of this document entitled "READ AND READLN" and "RESET".)

B. UNTYPED FILES

U.C.S.D. Pascal has one type of file declaration which in not found in the syntax of Jensen and Wirth. This type of file declaration 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 occured.

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 parameter being the file identifier, and the second parameter is 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 RANDOMACCESS;
   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 repoint 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 EOF 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 EOF 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.

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 executed 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 Q; FORWARD;

  PROCEDURE P;
  BEGIN
    READLN(T);
    WRITELN(T);
    IF T[1]='#' THEN EXIT(Q);
    WRITELN('LEAVE P');
  END;

  PROCEDURE Q;
  BEGIN
    P;
    WRITELN('LEAVE Q');
  END;

  PROCEDURE R;
  BEGIN
    IF CN <= 10 THEN Q;
    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
```

then the following output will result:

```
THIS IS THE FIRST STRING
LEAVE P
LEAVE Q
LEAVE R

#
LEAVE R

LAST STRING
LEAVE P
LEAVE Q
LEAVE R
```

The EXIT(Q) statement causes the PROCEDURE P to be terminated followed by the PROCEDURE Q. Processing continues following the call to Q 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(Q) statement is not executed processing proceeds normally through the terminations of procedures P and Q.

If the procedure identifier passed to EXIT is a recursive procedure then the most recent invocation of that procedure will be exited. Also, if in the above example program, one or both of the procedures P and Q had declared and opened some local files, then an implicit CLOSE(F) is done when the EXIT(Q) statement was executed, just as if the procedures P and Q had 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. However, the routine use of the EXIT statement is discouraged.

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. The net result is that E becomes an unpacked array of 40 words. On the otherhand, the PACKED ARRAY F is an array occupying 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;
```

then F and E would have had identical configurations.

In short, the reserved word PACKED only has true significence 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 in fact 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 of 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, then 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 16 STRINGS in this document.

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 of this documentation entitled "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 in 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 occurance 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 the RECORD A is as completely packed as possible. In B, the F field is not packed and therefore occupies two 16 bit words. In contrast A.F has all of its fields packed into one word. However, 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 in the above example, 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 amoung 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.

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>.

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 affect 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 predeclared files are: INPUT, OUTPUT, and KEYBOARD and are defined to be of <type> INTERACTIVE. If the program wishes to declare any additional files, then these file declarations must be declared together with the program's other VAR declarations.

11. READ AND READLN

Given the following declarations:

```
VAR CH: CHAR;
     F: TEXT;   (* TYPE TEXT = FILE OF CHAR *)
```

then 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. However, 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, with the difference being that the definition of the statement READ(F,CH) is the following two statement sequence which is the reverse of the sequence specified by the standard definition for files of <type> TEXT:

```
GET(F);
CH:=F^;
```

The difference mentioned above affects the way in which EOLN must be used within a program which is reading from a textfile of type INTERACTIVE. As mentioned in Section 5 of this document, EOLN becomes true only after reading the end of line character which is a carriage return. When this end of line character is read, EOLN is set to true and the character returned as a result of the READ will be a blank. In the example program fragments below, the left fragment is an example program taken from Jensen and Wirth in which 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 performes a similiar 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. (In previous releases, this function was performed by the U.C.S.D. intrinsics OPENOLD and OPENNEW. The I.4 Pascal compiler still supports the use of OPENOLD and OPENNEW by treating these predeclared identifiers as synonomous with RESET and REWRITE respectively. Eventually OPENOLD and OPENNEW will be "phased-out" and no longer supported by the Pascal compiler.)

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 which is returned as a blank 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". The textfile X could likewise 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 defined to allways 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 the other sections of this document describing FILES, EOF,EOLN, WRITE AND WRITELN, and RESET. Additonal documentation on the U.C.S.D. intrinsics can be found in Section 2.1.2 of this documentation entitled "INPUT/OUTPUT INTRINSICS".

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 defined by 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 to 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 this alternative form, RESET has two parameters, the first parameter is the file identifier, the second parameter is either a STRING constant or variable which corresponds to the directory filename of the file being opened. For further documentation regarding the use of RESET to open a file see section 2.1.1 of this documentation entitled "INPUT/OUTPUT INTRINSICS".

13. REWRITE(F)

The standard procedure REWRITE is used to open and create a new file. REWRITE has two parameters, the first parameter being the file identifier, the second parameter 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 the equivalent action as performed by the U.C.S.D. OPENNEW intrinsic and will eventually replace OPENNEW as the intrinsic used to open a previously non-existent file. For further documentation regarding the use of REWRITE to open a file, see section 2.1.1 of this documentation entitled "INPUT/OUTPUT INTRINSICS".

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 all at once.

The following program 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 NUNDER TWO
I'M DONE
```

For further documentation on SEGMENT PROCEDURES, their use and the syntax governing their declaration see Section 3.3 of this documentation entitled "SEGMENT PROCEDURES".

15. SETS

U.C.S.D. Pascal supports all of the constructs defined for sets on pages 50-51 of Jensen and Wirth. 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. However, the underlying type of both sets is the type INTEGER, which by the above definition of compatability, 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.
```

However, in the following example the construct

I = J

is not legal since the two sets are sets 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.
```

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. This default maximum LENGTH 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 appear below:

```
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, for example, 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 don''t work too 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 > 'SAMETHING' THEN
  WRITELN(S,' is greater than SAMETHING');
 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 SAMETHING
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 procedue READ and READLN, all characters up to the end of line charater (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 and a description of the U.C.S.D. STRING intrinsics see Section 2.1.1 of this documentation entitled "STRING INTRINSICS".

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 as the result of writing out the value of a BOOLEAN variable.

For a description of WRITE statements of variables of type STRING see Section 2.1.1 of this documentation entitled "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 only work if the ARRAY is a PACKED ARRAY OF CHAR. See the section of this document on PACKED VARIABLES for further information about packing.

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, then leading blanks are written. If the field width is smaller than the dynamic length of the string then the excess characters will be truncated on the right.

17.5 EXTENDED COMPARISONS.

U.C.S.D. Pascal allows = and <> comparisons of any array or record structure.

18. MISC. 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 16383 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.

19. 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	Results in a halt in 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.
IORESULT	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 withing a file.
SIZEOF	2.1.6	Function returning the number of bytes allocated to a variable.
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.4   January 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 (DELTAY / DELTAX) <= 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;
ROWSIZE, 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*);
```

- Notes -

**************** ***************

* FILE FORMATS * * Section 3.2 *

**************** ***************

Version I.4   January 1978

Code files are documented in Sections 3.4 and 3.5.

Text files are of the format:

```
<1024 bytes> header page, information for editors.
<1024 byte pages> where a page is defined:
        <[DLE][indent][text][CR][DLE][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  FOTOFILE: PACKED FILE OF SCREEN;
```

or something similar, which takes up the same dimensional space.

Data files are up to the user.

- Notes -

```
***************************  ***************
* SEGMENT PROCEDURE NOTES *  * Section 3.3 *
***************************  ***************
     Version I.4     January 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, in that 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 4.2) 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.

Reference Section 3.5, INTRODUCTION TO THE PASCAL PSEUDO MACHINE, for further details and examples.

```
******************************* ***************
* PSUEDO-MACHINE ARCHITECTURE * * Section 3.4 *
******************************* ***************
```

Version I.4   January 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.4 release and will only be available sometime later.

I. 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 funct'¬n 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.)
SEG: 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.

II. 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.)

GDIRP: 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, SEG, JTAB: copies of the BASE, MP, SEG 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.

HLTLINE: 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.

III. EXCEPTION HANDLING - XEQERR.

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 XEQERR routine. This routine

1) enters the error code into SYSCOM^.XEQERR.
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.

IV. 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 arrange 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:
                 15                                            0
                 -----------------------------------------------
      word 1:    !                 low mantissa                !
                 -----------------------------------------------

                 15 14                     7  6                0
                 -----------------------------------------------
      word 0:    !s !       exponent         !   high mantissa  !
                 -----------------------------------------------

Z80/8080:
                 15                     8  7                   0
                 -----------------------------------------------
      word 1:    !       low mantissa      !   middle mantissa  !
                 -----------------------------------------------

                 15 14                  8  7                   0
                 -----------------------------------------------
      word 0:    !s !   high mantissa      !       exponent     !
                 -----------------------------------------------
```

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+15) 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 many 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.

V. 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.

VI. 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 anscestor chain, traversed using the MSCW.MSSTAT links.

Mnemonic	Op-code	Parameters	Full name and operation
(* V.A Variable fetching, indexing, storing, and transfering *)			
(* V.A.1 One word loads and stores *)			
(* V.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.
(* V.A.1.b Local one word loads and store *)			
SLDL1 .. SLDL16	216 .. 231		Short load local word. SLDLx fetches the word with offset x in MP activation record and pushes it.

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.

(* V.A.1.c Global one word loads and store *)

SLDO1 .. SLDO16	232 .. 247		Short load global word. SLDOx fetches the word with offset x in MP activation record and pushes it.
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.

(* V.A.1.d Intermediate one-word loads and store *)

LOD	182	DB,B	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.

(* V.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.

(* V.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 similiar block. Transfer the block from the stack to the destination block.

(* V.A.3 Byte arrays *)

BYT	210		Byte conversion. Convert word pointer tos to a byte pointer. (NOP on the PDP11 and Z80/8080 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 similiar 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.

(* V.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 so as 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.

(* V.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 similiar block. Transfer the block from the source to the destination.
SIND0 .. SIND7	248 .. 255		Short index and load word. SINDx indexes the word pointer tos by x words, and pushes the word pointed to by the result.
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.

(*V.A.6 Dynamic variable allocation and deallocation *)

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-nil, cut NP back to GDIRP and set GDIRP to nil. 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 nil

if necessary, then store NP into word pointed to by tos.

RLS 158 32

Release heap. Set GDIRP to nil, then store word pointed to by tos into NP.

(* V.B Top of stack arithmetic and comparisons *)
(* V.B.1 Logical *)

LAND 132

Logical and. And tos into tos-1.

LOR 141

Logical or. Or tos into tos-1.

LNOT 147

Logical not. Take one's complement of tos.

EQUBOOL 175 6
NEQBOOL 183 6
LEQBOOL 180 6
LESBOOL 181 6
GEQBOOL 176 6
GTRBOOL 177 6

Boolean =,
<>,
<=,
<,
>=,
and > comparisons.
Compare bit 0 of tos-1 to bit_0 of tos and push true or false.

(* V.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.

SQI 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; Z80/8080 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 <= tos-2 <= tos, leaving tos-2 on the stack. If conditions are not satisfied a run-time error occurs.

EQUI 195 Integer =,
NEQI 203 <>,
LEQI 200 <=,
LESI 201 <,
GEQI 196 >=,
GTRI 197 and >
comparisons. Compare tos-1 to tos and push true or false.

(* V.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.

SQR 153 Square real.

DVR 135 Divide reals. Divide tos-1 by tos.

POT 158 35 Power of ten. The integer tos is check for 0 <= tos <= 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.
EQUREAL	175	2	Real =, <>, <=, <, >=, and > comparisons. Push TRUE or FALSE.
NEQREAL	183	2	
LEQREAL	180	2	
LESREAL	181	2	
GEQREAL	176	2	
GTRREAL	177	2	

(* V.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 <= tos <= 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 tos-1 > 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.)
EQUPOWR	175	8	Set =, <>, <= (subset of), and >= (superset of) comparisons.
NEQPOWR	183	8	
LEGPOWR	180	8	
GEQPOWR	176	8	

(* V.B.5 Strings *)

EQUSTR	175	4	String =,
NEQSTR	183	4	<>,
LEQSTR	180	4	<=,
LESSTR	181	4	<,
GEQSTR	176	4	>=,
GTRSTR	177	4	and >

comparisons. The string pointed to by word pointer tos-1 is lexicographically compared to the string pointed at by tos.

(* V.B.6 Byte arrays *)

EQUBYT	175 10	Byte array =,
NEQBYT	183 10	<>,
LEQBYT	180 10	<=,
LESBYT	181 10	<,
GEQBYT	176 10	>=,
GTRBYT	177 10	and >

comparisons. <=, <, >=, and > are only emitted for packed arrays of char.

(* V.B.7 Array and record comparisons. *)

EQUWORD	175 12	Word or multiword structure =
NEQWORD	183 12	and <>

comparisons.

(* V.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^[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 <> 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,	<case table>

Case jump. W_1 is word-aligned, and is the minimum index of the table. W_2 is the maximum index. W_3 is an unconditional jump instruction past the table. The case table is W_2-W_1+1 words long, and contains self-relative locations.

If tos, the actual index, is not in the range W_1..W_2, then IPC is pointed at W_3. Otherwise, tos-W_1 is used as an index into the table, and IPC is set to byte_address(casetable[index-min_index])-casetable[index-min_index].

(* V.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 -1 or 0. 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 -1 or 0. 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 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 (0 for procedures, 1 for non-real functions, and 2 for real functions) value. 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.

(* V.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 ?? Scan(maxdisp, start, forpast, char, mask).

MVL 158 ?? Moveleft(src, dst, numbytes).

```
MVR        158 ??                 Moveright(src, dst, numbytes).

(* Compiler procedures (still undocumented)                        *)

TRS        158                    Treesearch.

IDS        158                    Idsearch.

(* Debugger                                                        *)

BPT         213                   Breakpoint (conditional HALT)

(* Miscellaneous                                                   *)

TIM        158                    Time.
XIT        214
```

```
************************************************* ***************
* INTRODUCTION TO THE PASCAL PSEUDO-MACHINE * * Section 3.5 *
************************************************* ***************
```

Version I.4b April 1978

This document is a medium level description of the UCSD implementation of Pascal. This implementation is interpreter based. That is, 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, as mentioned in the introduction and overview document, the entire system can be moved to a new host machine by rewriting the interpreter for the new host.

Figure 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 fleshed out to a complete Pascal system, it would consist of at least 6000 lines of Pascal and compile to more than 50,000 bytes of code--too big to fit all at once into the memory of a small machine (by our current definition of small). Therefore we have extended Pascal so that a programmer can explicitly partition a program into <u>segments</u>; only some of these need be resident in main memory at a time. The syntax of this extension is shown in figure 1. (Any syntactic objects not defined explicitly there retain their standard interpretation as defined by Jensen & Wirth: <u>Pascal User Manual and Report</u>.)

```
<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 1. SEGMENT DECLARATION SYNTAX.

Segment declaration syntax (figure 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 doesn't change its meaning in any fundamental sense. When a segment is called (e.g. the COMPILER segment in line A), the interpreter checks 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 2. PASCAL SYSTEM CODE FILE.

The code file resulting from compilation of The Program is diagrammed in figure 2*. The file is a sequence of code segments preceded by a segment dictionary. The size of each segment is noted in <u>blocks</u>, 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 bj the order that one encounters segment procedure bodies in passing through The Program.

* An overview of the relationship among figures 2 through 8 (to be discussed in the following pages) is given in figure 9 at the end of this document. It is helpful to study figure 9 at this point for a better understanding of the document.

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 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. THE SEGMENT DICTIONARY

A code segment contains the code for the body of each of its procedures, including the segment procedure,itself. Figure 4 below 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 it appears at the beginning of the segment. The outer block code is generated and appears last.

```
                   high addresses
              odd                   even
     !------------------------------------------!
     ! Number of procedures ! Segment Number    !
     !      in dictionary   !                   !
     !------------------------------------------!
     ! Procedure #1                PASCALSYSTEM !--!
     ! - - - - - - - - - - - - - - - - - - - -  !  !
!----! Procedure #2                 CLEARSCREEN !  !
!    ! - - - - - - - - rest of - - - - - - - -  !  !
! !--! - - - - procedure  dictionary - - - - -  !  !
! !  !------------------------------------------!  !
! !  !                                          !  !
! !  !                                          !  !
! !  !     PASCALSYSTEM's outer block code      !<-!
! !  !                                          !
! !  !                                          !
! !  !------------------------------------------!
! !  ! other procedures of the Pascal system    !
! !  !------------------------------------------!
! !->!  PROCEDURE #3                   code     !
!    !------------------------------------------!
!--->!  PROCEDURE #2  (clearscreen)  code       !
     !------------------------------------------!

                  low addresses
```

FIGURE 4. A CODE SEGMENT

A more detailed diagram of a single procedure code section is seen in figure 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 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.

```
                 high addresses
              odd              even
       !---------------------------------!
       !  Lex Level   !  Procedure #     !<-------!--------------------!
       !---------------------------------!        !  PASCALSYSTEM's    !
       !            Enter IC             !--!     !  Procedure         !
       !---------------------------------!  !     !  Dictionary        !
   !--!             Exit IC              !  !     !  Pointer           !
   !   !---------------------------------!  !     !--------------------!
   !   !        Parameter Size           !  !
   !   !---------------------------------!  !
   !   !     Data   Segment   Size       !  !
   !   !---------------------------------!  !
   !   !- - - - - Jump Table - - - - - - !  !
   !   !---------------------------------!  !
   !   !                                 !  !
   !-> !                                 !  !
       !---------------------------------!  !
       !                                 !  !
       !          CLEARSCREEN            !  !
       !             CODE                !  !
       !                                 !<-!
       !---------------------------------!
                low addresses
```

FIGURE 5. PROCEDURE CODE SECTION (OF CLEARSCREEN)

```
                 high addresses

          |----------------------------|
          |  System Resident Segment   |
          |----------------------------|
          |    System Data Segment     |
          |- - - - - - - - - - - - - - |
          |        mark  stack         |
          |----------------------------|
          |   Compiler Code Segment    |
          |----------------------------|
          |   Compiler Data Segment    |
          |- - - - - - - - - - - - - - |
          |        mark  stack         |
          |----------------------------|
          |   Compinit Code Segment    |
          |----------------------------|
          |   Compinit Data Segment    |
          |- - - - - - - - - - - - - - |
          |        mark  stack         |
          |----------------------------|
          | CLEARSCREEN Data Segment   |
          |- - - - - - - - - - - - - - |
          |        mark  stack         |
          |----------------------------|
          |        temporaries         |
          |- - - - - - - - - - - - - - |
          |                            |
          |                            |
          |                            |
          |                            |
          |                            |
          |----------------------------|
          |                            |
          |        H  E  A  P          |
          |                            |
          |----------------------------|
          |        Interpreter         |
          |-- - -- - -- - -- - -- - -- |
          |        S Y S C O M         |<- <segment dictionary>
          |-- - -- - -- - -- - -- - -- |
          |----------------------------|

                  low addresses
```

FIGURE 6. SYSTEM MEMORY DURING CLEARSCREEN EXECUTION

Figure 6 is a snapshot of system memory during the execution of a call to procedure CLEARSCREEN from line C in COMPINIT. The Pascal 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 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 prodedure COMPINIT is called in line B, its code segment (including all compiler initialization procedures) is loaded on the stack. Then the COMPINIT data segment is built on top of that. Figure 7 is a diagram of the data segment for COMPINIT.

```
                          high addresses

                  |-------------------------------|
                  | Other COMPINIT variables      |
                  |-------------------------------|
                  |             BOOL              |
                  |-------------------------------|
                  |              I                |
                  |-------------------------------|
                  |              J                |
                  |-------------------------------|<--|
                  |             MSSP              |   |
                  |-- - -- - -- -- -- - -- - --|   |
                  |             MSIPC             |   |
                  |-- - -- - -- -- -- - -- - --|   |
                  |             MSSEG             |   |
                  |-- - -- - -- -- -- - -- - --|   |--> markstack
                  |             MSJTAB            |   |
                  |-- - -- - -- -- -- - -- - --|   |
                  |             MSDYN             |   |
                  |-- - -- - -- -- -- - -- - --|   |
      |----|      |             MSSTAT            |   |
      | MP |-->|-- - -- - -- -- -- - -- - --|<--|
      |----|

                          low addresses
```

FIGURE 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 <u>called</u> 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. So 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 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.

```
                                   to PASCALSYSTEM resident code segment
                                                       ^
to PASCALSYSTEM resident data segment                  |
    ^                                                  | SEG  |<-    in
    |                      high addresses              | JTAB |<-PASCALSYSTEM
    |           |-----------------------------------|  | IPC  |<-code segment
    |           |-----------------------------------|
    |           |      COMPILER  code  segment      |
    |           |-----------------------------------|
    |           |      COMPILER  data  segment      |
    |           |- - - - - - - - - - - - - - - - - -|
    |           |            markstack              |
    |           |-----------------------------------|
    |   |------>|       20        |        4        |      ---------|
    |   |       |-----------------------------------|               |
    |   |       |     Pointer to COMPINIT code      |--|            |
    |   |       |-----------------------------------|  |      -----------
    |   |       |     Pointer to Procedure #2       |--+-|        code
    |   |       |-----------------------------------|  | |      segment
    | |-+------>|-----------------------------------|<-| |        of
    | | + |---->|     COMPINIT       code           |    |      COMPINIT
    | | | |     |-----------------------------------|    |    -----------
    | | | |     |     Procedures   of   COMPINIT    |<---|          |
    | | | |     |-----------------------------------|      ---------+
    | | | |     |     COMPINIT    variables         |               |
    | | | |     |-----------------------------------|               |
    | | | |     |              MSSP                 |               |
    | | | |     |-----------------------------------|               |
    | | | |     |             MSIPC                 |         -----------
    | | | |     |-----------------------------------|             data
    | | | |     |             MSSEG                 |           segment
    | | | |     |-----------------------------------|             of
    | | | |     |             MSJTAB                |           COMPINIT
    | | | |     |-----------------------------------|         -----------
    | | | |     |             MSDYN                 |               |
    | | | |     |-----------------------------------|               |
    | | | |     |             MSSTAT                |               |
    | | | |     |-----------------------------------|<--|  ---------|
    | | | |     |        evaluation   stack         |   |
    | | | | |-->|-----------------------------------|   |  ---------|
    | | | | |   |    CLEARSCREEN    variables       |   |           |
    | | | | |   |-----------------------------------|   |           |
    | | | | |---|             MSSP                  |   |           |
    | | | |     |-----------------------------------|   |           |
    | | | |-----|            MSIPC                  |   |     -----------
    | | |       |-----------------------------------|   |         data
    | | |-------|            MSSEG                  |   |       segment
    | |         |-----------------------------------|   |         of
    | |---------|            MSJTAB                 |   |     CLEARSCREEN
    |           |-----------------------------------|   |     -----------
    |           |            MSDYN                  |---|           |
    |           |-----------------------------------|               |
    |-----------|            MSSTAT                 |<----| MP |    |
                |-----------------------------------|      ---------|
                |        evaluation   stack         |
                |               .                   |
                |               v                   |
                |- - - - - - - - - - - - - - - - - -|<----| SP |
|                                  |
|                                  |
|- - - - - - - - - - - - - - - - - |<----| NP |
|            H  E  A  P            |
|----------------------------------|

             low addresses
```

FIGURE 8. The stack during clearscreen

The introduction promised a top-down description of the Pascal pseudo-machine. Figure 9 reflects that top-down process by showing the relationships among diagrams 2 through 7.

```
   code file
   figure  2
|--------------|
| PASCALSYSTEM |--->|  figure 4    |
|--------------|    |              |
|              |    | CLEARSCREEN  |--->|  figure 5     |
|              |    | code detail  |    | proc. code    |
|--------------|                        | detail        |
|   segment    |
| dictionary   |--->|         figure 3           |
|--------------|    | segment dictionary detail  |


  system memory

   figure 8
| code segment |--->|  figure 4    |
|--------------|
|              |
|--------------|
|   COMPINIT   |
| data segment |--->|  figure 7              |
|--------------|    | data segment detail    |
```

FIGURE 9. RELATIONSHIP OF DOCUMENT FIGURES

FIGURE 10. The Program

```
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.
```

N.G. ? 24-JAN-79

```
********************   ***************
* THE CALCULATOR   *   * Section 4.1 *
********************   ***************
```

Version I.4   January 1978

CALCULATOR is a program written by Dale Ander to utilize the computer as a calculator.

When the program is running the following prompt comes up '->' and you are expected to enter a one line expression in algebraic form.

You may have up to 25 different variables, each with different values assigned by you using the syntax of the given grammer. Only the first 8 letters are used to distinguish between variables. Once variables have a value they may be used like constants. There are two built-in variables: PI (3.141593) and E (2.718282), however these values may be changed by the user.

There is no distinction 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 make them 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 make it an integer and it must be between zero and thirty-three inclusive or else 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 a RETURN immediately following the prompt.

PAGES 169-178 DO NOT EXIST.

KAH (sd)

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 )
        3.0
-> A
        3.0
-> C
        3.0
-> 1 + 2
        3.0
-> 3 + 7 / 4
        4.75
-> SQRT(2*2+3*3)
        3.605551
```

```
************** ***************
* THE LINKER * * Section 4.2 *
************** ***************
```

Version I.4   January 1978

The linker is a program which "stitches" the separately compiled segments of a very large program together. When executing, it will ask for

Output Code File?

Reply with the name of the file you wish it to create. Next it will ask

Link Code File?

Reply with any one of the code files you want linked into the output file. It will ask about each of the segments within the code file individually. To link them into the output file reply with a 'Y'(es; any other character will make the LINKER go on to the next segment. Note that it will not ask about segments containing no code. After exhausting all segments within one code file, the LINKER will ask for another. Continue to reply in a like manner until all your files are linked together. Terminate execution with a carriage return. This will lock the output file onto your disk and exit the LINKER.

In many cases it will be beneficial for the output file to be named the same as one of the to-be linked files. This duplicate naming will cause the new output file to replace the old file. (It is advisable to re-bootstrap after such a procedure when working with the system.) If you ask the linker to link in the same segment procedure twice (i.e. two segment procedures with the same segment number) while running LINKER, it will ask you to reconfirm that link. It does so because linking that segment in twice will result in wasted blocks on disk within the code file since the segment, as was previously read in, is still there.

To replace a segment, re-execute LINKER and link the new segment in place of the old one. LINKER will inform you of any input/output errors that it incurs while running.

Creating a new system:

In order to create a new system, or to change part of it, you will need to compile that segment of the system. In order to do this so that all parts of the system will be talking to each other correctly, it is imperative that they all be compiled with the same globals. Dummy declarations of the segments that precede the segment you are working on are necessary in order to ensure that your segment gets the appropriate segment number. Having successfully compiled your segment, follow the procedure described above.

You may name the output file whatever you wish. The input "link" file(s) need to be those code files which the compilation(s) of your segment(s) has(have) generated and the system. An example of stitching three code files follows:

Example:

This is one file (A.TEXT) containing one segment procedure of a small system, which is to be linked later with other segment procedures.

```
PROGRAM LINKDEMO;
VAR I: INTEGER;

PROCEDURE XIT; FORWARD;

SEGMENT PROCEDURE A;
BEGIN
  WRITELN(' I HAVE ENTERED A');
  XIT;
END;

PROCEDURE XIT;
BEGIN
END; (* A DUMMY XIT *)

BEGIN
  (* DUMMY MAIN BLOCK *)
END.
```

This is a second file (B.TEXT) containing another segement of this sample system. Note the forward declarations of A and XIT.

```
PROGRAM LINKDEMO;
VAR I: INTEGER;

PROCEDURE XIT; FORWARD;

SEGMENT PROCEDURE A;
BEGIN
  (* DUMMY BLOCK *)
END;

SEGMENT PROCEDURE B;
BEGIN
  XIT;
  WRITELN('THIS IS IN B');
END;

PROCEDURE XIT; BEGIN END; (* ANOTHER DUMMY XIT *)

BEGIN
  (* DUMMY MAIN BLOCK *)
END.
```

This is the outer block of the system. It is the file which contains the actual declarations for the global routines as well as the outer block of the system. Note that all segment procedures are declared with dummy blocks.

```
PROGRAM LINKDEMO;
VAR I: INTEGER;

PROCEDURE XIT;
FORWARD;

SEGMENT PROCEDURE A;
BEGIN
  (* DUMMY BLOCK *)
END;

SEGMENT PROCEDURE B;
BEGIN
  (* DUMMY BLOCK *)
END;

PROCEDURE XIT;
BEGIN
  WRITELN(' THIS IS THE CODE FOR XIT');
END;

BEGIN
  WRITE(' THIS IS THE MAIN PROGRAM');
  B;
  A;
END.
```

If each of these files (A.TEXT, B.TEXT, and O.TEXT) is compiled separately and saved as A, B and O respectively, the session with the linker would look as follows:

```
Output code file? MYPROG.CODE
Link code file? A.CODE
Linking LINKDEMO.  Please Confirm(y/n) N
Linking A       .  Please Confirm(y/n) Y
A        Seg # 10, Block 1, 56 Bytes
Link code file? B.CODE
Linking LINKDEMO.  Please Confirm(y/n) N
Linking B       .  Please Co. 'irm(y/n) Y
B        SEG # 11, BLOCK 2, 50 BYTES
Link code file? O.CODE
Linking LINKDEMO.  Please Confirm(y/n) Y
LINKDEMO SEG # 1, BLOCK 3, 116 BYTES
Link code file?
String to be added to file? Copyright (c) 1978, Regents of Univ. Cal.
```

This session will create the file MYPROG. The string, which is added to the header page of the code file, is useful for making code files distinguishable from other files during disk recovery or for adding copyright notices to object files.

```
********************************* ***************
* SETUP -SYSTEM RECONFIGURATION * * Section 4.3 *
********************************* ***************
```

Version: I.4   January 1978

I. INTRODUCTION

The SETUP program enables the user to reconfigure the UCSD PASCAL Operating System to suit his equipment or taste. It accomplishes this remarkable feat by altering the contents of system global variables. Most modifications are related to adapting the system for use with different terminals, while others adapt the system to differences in machine configuration (eg. Whether or not it is equipped with a real time clock.). SETUP enables the user to make these changes at any time, quickly and easily, either permanently or temporarily.

The UCSD PASCAL Operating System has certain features making it especially well suited to high speed CRT devices. It is easily adaptable to other terminal units as well, including storage tube and slow impact printing devices. The PASCAL system must be adapted to differences between the hardware designs of these terminals as well as to the fact that control characters are not standardized at this time.

II. USING THE SETUP PROGRAM

A. DEFAULT CONFIGURATION

The UCSD PASCAL system assumes that all terminals respond to the standard ASCII <line feed> (decimal 10, octal 12) and <carriage return> (decimal 13, octal 15) characters. With the configuration in which it is supplied, the system uses these two control codes and no other cursor addressing schemes. The system also assumes that the NUL (0) character does nothing o. serious consequence to the terminal. The ASCII underline character (decimal 95, octal 137) is interpreted as the single character delete key (CHARDEL). Thus, all terminals "look like" Tele-types to the system in its default configuration. This has been done to make the system immediately compatible with as many different terminals as possible. By using SETUP to modify this Teletype-ish response, the user can take full advantage of the screen-oriented display features of UCSD PASCAL. As noted above, there are other relevant differences between installations, however the display options are the most "visible" to the user.

B. HOW TO DO IT, STEP BY STEP

SETUP is run like any other compiled PASCAL program, by entering the "Command:" level of the UCSD PASCAL system and typing 'X' (for eXecute), followed by the filename 'SETUP' (Don't use the quote marks) and a <carriage return>. If the system is working correctly, you will then see the prompt line for the command level of SETUP:

```
SETUP: N(ew, S(ingle, L(ist, R(adix, P(ermanent, Q(uit, <ESC>
```

First, type 'L' for 'List'. This should cause SETUP to list each of the variables with which it deals and their current values. Note that the output generated may exceed the display capacity of your screen and the first lines will be lost. To avoid this, type 'L' again and use CTRL S to freeze the display before the top lines get away. Typing CTRL S again will unfreeze the display so that it can continue -- no output will be lost. Next, type 'R' for 'Radix'. The prompt line informs you that you may now type 'O','D' or 'H' to change the "default radix" to Octal, Decimal or Hexadecimal, respectively. Try typing these characters in any order and note that the new default radix is displayed each time. See section F below for more on Radix Default Setting. Leave the default radix set the way you like it and type <carriage return> to return to the command level of SETUP. Now type 'N' for the 'New Configuration Mode'. You will see a prompt that looks like this:

```
New Configuration CHANGE:
[O(ct) D(ec) H(ex)] <New Value>, <Space>, <CR>, <ESC>
```

These commands are explained in greater depth under section D below. Note that the first variable is displayed beneath the prompt line, along with the current default radix. The variable display should look like this:

```
(STUPID) STUdent Program ID? [FALSE]:
```

The characters in parentheses are the identifier associated with this variable. This identifier is printed on the left when 'List' is called. The stuff between the square brackets is the current value of this variable, either Boolean (True, False), ASCII Character Code (Octal, Decimal or Hex) or Integer (Decimal only). STUPID is a Boolean variable, as noted in section III.A of this document. Press 'T' and STUPID will be set TRUE and the next variable will be displayed. Now press <spacebar>; the value of the current variable will be unchanged and the next variable will be displayed. Press <spacebar> several times and several variables will be displayed. Press <carriage return> to return to the command level of SETUP. Use 'L(ist)' (ie. type 'L') to display the variables once again and note that STUPID has been changed. Finally, type <ESCAPE> (<ALTMODE>) to exit from SETUP, returning to the command level of the PASCAL system. Sections II.I and II.H below explain the difference between the <ESCAPE> and 'Q(uit)' commands for exiting SETUP. The above is a simple walk-through of SETUP, showing only a few of its many wonderful features.

C. SETUP COMMAND SUMMARY

'N' --New configuration mode. Each of the variables
accessible with SETUP is displayed in turn. The user may change any or all of them and exit at any time.

'S' --Single change mode. The user must specify the variable(s) he wishes to change by name.

'L' -- List.
Displays a list of all the accessible variables and their current value.

'R' -- Radix default setter.
The user may change character variables in either octal, decimal or hexadecimal radices. This routine sets the default radix.

'P' -- Permanent.
Updates the disk file 'SYSTEM.INTERP' to incorporate the changes made using SETUP (Otherwise, the next time you bootstrap, the changes won't be there). This does not cause an exit from SETUP.

'Q' -- Quit.
Exit SETUP and incorporate the changes that have been made into the currently running PASCAL system.

<ESC> -- Escape.
Exit SETUP and cancel all changes that have been made (except Permanent changes).

D. NEW CONFIGURATION MODE

In New Configuration mode, each variable is displayed in turn for the user to change if desired. There are several options for the user for each variable displayed: (1) He may press <spacebar> to proceed to the next item. (2) He may press <carriage return> to exit from New Configuration mode, accepting the changes he has made. (3) He may press <Escape> to exit from New Configuration mode without accepting any of the changes, or (4) he may make a change by typing in the new value, followed by a <spacebar> to proceed, a <carriage return> accept and exit or an <escape> to abort and exit. Exiting from New Configuration mode will return program control to the outer level of SETUP.

Changing a Boolean variable is accomplished by entering a T or Y for True, or an F or N for False when prompted by the program. Other answers will generate an error message, leaving the current value of the variable unchanged.

Changing character variables may be done in either octal, decimal radices. The default radix is displayed with the prompt for the variable and a new value may be entered immediatly in that radix. To use one of the other radices, simply type 'O' for octal, 'D' for decimal, or 'H' for hexadecimal and the entry radix will be changed for this particular variable. The radix may be swapped around as much as you want until another character is typed after which the radix is fixed at its last setting. Note that the default radix remains unchanged until changed using the Radix default setting procedure.

Integer variables are most easily changed in decimal, thus the working radix for Integer variables is temporarily changed to decimal. The user may use the 'O','D' and 'H' commands as before, however.

E. SINGLE CHANGE MODE

Single Change mode is similar to New Configuration mode in that the prompts and responses are identical for each individual variable. See New Configuration mode for further information on how to change a variable once it is displayed. The two modes differ in that Single change mode requires the user to enter the name of the variable he wishes to change. The program then displays only that variable. If the user types <spacebar> before or after entering the new value, the program will ask him for another identifier. If the user types a <carriage return> instead, the new value will be accepted and he will be back in the outer SETUP level. If he types <Escape>, all the changes he has made since entering Single Change mode will be aborted and he will return to the command mode.

F. RADIX DEFAULT SETTING

The Radix Default Setting procedure displays the current default radix (initialized as decimal) and accepts changes as follows: 'O' changes it to octal, 'H' changes it to hexadecimal and 'D' changes it to decimal. A <carriage return> will return the user to the outer SETUP level. Unless otherwise specified when changing a variable, the prompt display and the changes to all character variables (ASCII) will be made in the default radix.

G. PERMANENT

The Permanent procedure allows the user to preserve his modifications as permanent changes in the disk file 'SYSTEM.INTERP'. These changes may be updated at any time through use of this procedure. The program requests the user to specify a volume name followed by a <carriage return>; if no volume is given, the system default volume is assumed, and entering a '*' implies the bootstrap disk (see document UD2, the File Handler). The program will respond 'Are you sure?' to request the user to confirm the change. Typing a 'Y' will cause the change to be carried out; any other response will leave 'SYSTEM.INTERP' unchanged. The user will not exit from SETUP but return to the outer SETUP level.

H. QUIT

Quit causes all changes that have been accepted from the New and Single modes to be incorporated into the currently running operating system. Quit terminates the SETUP program and returns the user to the PASCAL system command level. At that time, the user will discover that all changes he has made using SETUP are now in effect,except FILLCOUNT, which will not take effect until the system is reinitialized.

I. <ESCAPE>

Typing <Escape> (or <ALTMODE> on some terminals) causes the SETUP program to terminate without incorporating any changes into the currently running PASCAL system. Any changes that may have been made are now discarded and the user returns to the PASCAL system command level.

J. STRATEGIES FOR USING SETUP

We strongly recommend that no Permanent changes be made until the results have been thoroughly tested by simply using the 'Q'uit command. It should be noted that, if the changes are found to be satisfactory, they may be made permanent by simply executing SETUP and typing 'P'. Even though the modifications may have been made on a previous run of SETUP, the Permanent mode will incorporate those changes into the disk file, UNLESS the system has been rebootstrapped since they were made, in which case they are gone.

III. OPTIONS PROVIDED

A. Miscellaneous
B. Terminal Control ᴗodes
C. User Command Codes

A. MISCELLANEOUS

STUPID (True, False): STUdent Program ID. Not yet implemented.

Suggested setting: False. Do not set to True at this time.

SLOWTERM (True, False): Slow Terminal. When SLOWTERM is true, the system issues abbreviated promptlines and messages.

Suggested setting: 600 baud and under -- True, otherwise False. Default is False.

HASXYCRT (True, False): X-Y cursor addressing is available. When HASXYCRT is true, the system assumes that the cursor may be addressed using an X-Y addressing scheme. This variable is used only by the screen-oriented editor.

Suggested setting: Consult terminal manual to see if this feature is available on your terminal. Default is False.

HASLCCRT (Boolean): Lower case letters are available. HASLCCRT informs the system whether or not lower case letters may be input. This variable is used only by the screen-oriented editor.

Suggested setting: Set true if lower case letters are on terminal, otherwise set false. Default is False.

WIDTH (Integer): Screen Width. The WIDTH setting affects the screen-oriented editor.

Suggested setting: Set to number of characters per line. Default is 72.

HEIGHT (Integer): Screen Height. HEIGHT is used by several system routines to govern the page length of the display.

Suggested setting: Set to number of lines displayable at one time on your terminal. On non-paging devices (ie. those which scroll, eg. hardcopy or "glass teletypes"), set to zero. Default is zero.

HASCLOCK (True, False): A real time clock is available. A real time clock module, such as the DEC KW11, may be found on many processors. If available it is used by the PASCAL system to optimize disk directory updates. See the TIME intrinsic in the Intrinsics document.

Suggested setting: If a real time clock is available, set true, otherwise set false. Default is False.

BADCH (Character): Symbol for unprintable character. When a code is to be printed which does not denote a printable character in the ASCII code, the BADCH character is printed.

Suggested setting: Default is question mark (?), (Decimal 63, Octal 77).

B. TERMINAL CONTROL CODES

The Terminal Control Codes are ASCII character codes defined by the manufacturer of the individual terminal to control certain terminal functions such as erasing the screen or doing a line feed. Many control codes are now standardized, however there are also some that vary a great deal. To enable the PASCAL system to adapt to these variances, it is possible for the user to redefine the following values to conform to his terminal. Note that the Terminal Control Codes are not subject to the whim of the user, as are the User Command Codes which follow. Rather, they must conform to the terminal manufacturer's specifications. Note also that by setting all the Terminal Control Codes except HOME to zero, a terminal will look like a Teletype to the system. This is the configuration in which the system is supplied.

ESCAPE (Character): Escape-Mode character. Some terminal devices require an "escape sequence" for certain functions. In these circumstances, there is a distinguished character which sends the terminal into the "escape mode" (the ESCAPE character). It is then followed by another control character, such as ERASEEOS.

Suggested setting: Consult manufacturer's specs; if escape sequences are not used, set to zero, otherwise set to specs. Default is zero.

HOME (Character): Move cursor Home. Another screen terminal feature, the HOME command is used to move the cursor to the upper left ("home") corner of the screen.

Suggested setting: Set to manufacturer's specs; IMPORTANT -- if not available, set to <carriage return> character. Default is octal 15, decimal 13, hex OD.

ERASEEOS (Character): Erase to End of Screen. On screen terminals, usually there is available a command for erasing either the entire screen or from the cursor to the end of the screen.

Suggested setting: Set to manufacturer's specs; if not available, set to zero. Default is zero.

ERASEEOL (Character): Erase to End of Line. Screen terminals frequently offer the user the option of erasing from the cursor position to the end of the current line, without disturbing the rest of the screen. ERASEEOL is the control character which causes this function.

Suggested setting: Set to manufacturer's specs; if not available, set to zero. Default is zero.

BACKSPACE (Character): The BACKSPACE control character causes the terminal to move the cursor LEFT one position.

Suggested setting: Set to code emitted by backspace key on terminal, if such exists (usually decimal 8, octal 10). Default is zero.

NDFS (Character): Non-Destructive Forward Space. NDFS is the control character which causes the cursor to be moved RIGHT one position, without obliterating displayed data.

Suggested setting: Set of manufacturer's specs; if not available, set to zero. Default is zero.

RLF (Character): Reverse Line Feed. The RLF character is the control character causing the cursor to move UP one line.

Suggested setting: Set to manufacturer's specs; if not available, set to zero. Default is zero.

FILLCOUNT (Integer): Number of nulls to send after Y-axis 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. FILLCOUNT number of nulls will be sent after carriage returns, ERASEEOL, ERASEEOS and RLF.

Suggested setting: If a delay is required by your terminal, set FILLCOUNT to the longest delay needed for the baud rate at which you plan to run. If no delay is needed, set FILLCOUNT to zero. Default is 10.

C. USER COMMAND CODES

The following are user-selectable command codes. The PASCAL system responds to certain keyboard commands for special functions. The exact key which actuates these functions is sometimes subject to users' whims by changing the ASCII code to which the system will respond. Note that these are codes which are INPUT to the system, not the output control codes to which the terminal responds.

UP, DOWN, LEFT, RIGHT(Character): Vector keys. These keys cause the cursor to move according to their name, either right or left one space, or up or down one line at a t:me. The screen- oriented editor responds to these commands.

Suggested setting: If vector keys are available, set to the codes emitted by each key (consult terminal manual), otherwise, set to some convenient codes such as CTRL W,Z,A,S or CTRL E,S,X,D. Default is zero.

CHARDEL (Character): Single character delete key. The CHARDEL key will cause a single character to be removed from the end of the user's text buffer when entering an input string (eg. while editing).

Suggested setting: Set to code emitted by Backspace key (Decimal 8, Octal 10). Default is octal 137, decimal 95 (ASCII underline).

LINEDEL (Character): Line delete character. Depressing LINEDEL will cause the current line of input to be erased. LINEDEL is similar in function to CHARDEL differing in that CHARDEL does only single characters at a time. Successive actuations of LINEDEL will not erase successive lines of text, unlike CHARDEL which will erase successive characters.

Suggested setting: Rubout or Delete key (usually Decimal 127, Octal 177, which is the default).

STOP (Character): 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: CTRL S (Decimal 19, Octal 23) is the default.

FLUSH (Character): 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: CTRL F (6) is the default.

BREAK (Character): BREAK will cause the program currently executing to be terminated with a run-time error immediately.

Suggested setting: Code emitted by Break key on terminal, (usually null--0). You may wish to set BREAK to something difficult to type accidentally. Default is zero.

EOF (Character): 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 the discussion of the EOF intrinsic in another document.

Suggested setting: CTRL Z (Decimal 26, Octal 32) is the default.

ALTMODE (Character): Altmode or Escape key. The ALTMODE or ESCAPE key is used as a command in various levels of the PASCAL system. Consult specific documentation for its meaning in each context.

Suggested setting: Consult terminal manual and set to the code generated by the appropriate key on your terminal (Usually octal 33, decimal 27). Default is zero.

```
******************** ***************
* BOOTSTRAP COPIER * * Section 4.4 *
******************** ***************
    Version I.4       January 1978
```

The bootstrap copier BOOTER.CODE is a friendly handshaking program. It will ask you for the unitnumber of the volume you wish it to write the bootstrap on. Refer to Table 5 for a list of volume numbers. It will then ask you for a file name to write as the bootstrap. It writes the first two blocks of that file, so if you want to copy the bootstrap from an existing disk, just give it the diskname, and it will copy the bootstrap from the disk you have named to the unit you have numbered.

To execute the BOOTER program, type X BOOTER to C(ommand level (assuming that you indeed have a copy of BOOTER.CODE on your disk).

In some future release the bootstrap copier will be moved into the zero command of the F(iler, but, that is some future release.....

- Notes -

```
********* ***************
* PATCH * * Section 4.5 *
********* ***************
```

Version I.4 January 1978

The PATCH program is written for those programmers who simply must see it in HEX. It is screen oriented, and requires a correct version of the system procedure GOTOXY to be bound in. It uses the vector keys, like the screen oriented editor, to position the cursor over the hex digit in question. Typing the correct hex digit will update the in-memory version of the block you G(ot. Then you must type 'P' to the P(atch: level to write the block back to disk.

The commands PATCH understands are as follows:

F(ile: will ask for a filename (or <return for unit-number>).

Q(uit: leaves PATCH.

G(et: will ask for a block number (and expects <cr>).

M(ixed: Displays the block in mixed ascii and hex, ascii for those things that it can print, hex for the rest.

H(ex: Displays the block all in hex.

P(ut: Writes the block out.

While in display mode:

any hex character: goes into the block.

any vector key: moves one that direction.

U,Z,L,R: Up, Down, Left, Right. (Z because D is a hex char)

S(tuff: Stuffs the display for however many bytes you tell it.

Q(uit: Goes back to P(atch.

- Notes -

```
************************************ ***************
* SERIAL LINE INTERFACE DRIVER KIT * * Section 4.6 *
************************************ ***************
```

Version I.4   January 1978

This utility program designed for interactive use between the TERAK 8510A and other computer systems.
The program allows the user to interface the TERAK with other systems by emulating a terminal. File transfer is accomplished by having the program type the correct sequence of characters that would cause the host system to generate a listing stream to the terminal. It swallows these characters as they arrive, and ships them off to whatever file you have specified.

In order to make use of the interface between the two machines it is necessary to type <esc>, which then responds with a program prompt of the form:

```
Filexfer: G(et), S(end), P(ascal), H(ost)
```

Following is a description of each of the available commands.

G(et) : This command prompts the user for a host file title by asking for a 'Host input file?' and then prompts with 'Pascal output file?' . Assuming that a legal B6700 file title was entered the program will proceed to transfer the requested file to a TERAK file(assuming no I/O errors occur). Once the transfer is complete the user will be notified and the above prompt line will appear again.

S(end) : The program will prompt with 'Pascal input file?' and following the user's response will then ask for the 'Host output file?'. If the named Pascal file exists it will be transferred to a host file(and to a corresponding ARCHIVE file) under the title given in response to the second prompt.

P(ascal): This simply returns control to the Pascal Operating System. To return to the program host it is necessary to re-execute the program.

H(ost) : This command will return control to the host system. Another <esc> will respond with the 'File- xfer' prompt.

You will need to modify the source of this program in four places to make it behave properly with your host machine. The first two are at the very top of the program, there are two constants: HALFDUPLEX, and UPPERCASE. These need to be appropriately equated to either TRUE or FALSE as the case may be. The third is at the beginning of procedure SENDFILE. The line that reads:

```
        S :=  CONCAT(' - - - ',TITLE,' - - - ');
```

needs to be modified as to create the string that will cause your host machine to accept the following stream of characters as something that will be sent to some other destination. The fourth is at the beginning of procedure GETFTP. The first line of this procedure reads:

```
        S := CONCAT(' - - - ',TITLE,' - - - ');
```

you need to modify it so that when it is typed to your host computer, it will start sending the file specified in TITLE as a continuous stream of characters.

The author of this document suggests that you study the program to see what other little quirks it expects from your host system, and modify it appropriately. We have it talking to CANDE on our B6700, and to a number of UNIX installations. Good luck, and happy transferring.

```
**************************** ***************
* DEMONSTRATION PROGRAMS * * Section 4.7 *
**************************** ***************
```

Version I.4 January 1978

The demonstration programs available with the UCSD PASCAL SYSTEM are constantly changing, and, as documenting a moving target is a difficult proposition, an extra disclaimer that we are not responsible for any discrepancies between the documents and the actual programs is in order. All demonstration programs listed here are specific to the TERAK 8510a.

CYCLE :

A potpourri of graphics to keep the machine busy. Turtle 'squirals' and other 'pretty' pictures keep the screen moving. CYCLE demonstrates some of the capabilities of the turtle graphics package as well as displays generated with more 'primitive' low level graphics techniques such as DRAWLINE and recursively based algorithms. CYCLE also gives a demonstration of the panning capabilities of the screen graphics. The program may be terminated by typing a carriage return whereupon it will finish the current cycle and exit.

TURTLE :

This program is a useful introduction to the use of UCSD PASCAL turtle graphics procedures and functions. It emulates the turtle in an interactive mode allowing the user to type in turtle commands(e.g. MOVE(50)) which are activated (assuming that they are syntactically correct) by typing a carriage return following the input string. Commands not displayed on the prompt line include MOVETO(x,y) and TURNTO(angle).

SINEX :

Demonstrates the panning and double buffering capabilities of graphics mode of screen. Calculates and plots Y=SIN(X)/X. Program and panning may be terminated by typing return.

DERIVATIVE :

This demonstrates mathematical prowess by calculating successive derivatives of Y=SIN(X)/X via the difference method. It will continue until deverivative 'blows up'(i.e. exceeds screen boundaries.) Program waits for a carriage return.

CHEDIT :

The character set editor. (See document UD9)

CHDEMO :

An interactive demonstration of the character editing facilities of the CHEDIT program. Commands of the form <esc> followed by a digit will enable use of various character sets available on the disk. The user may type in the alloted workspace to see the format of any specific characters.(This program is self-prompting). End by typing <esc><esc>.

LIFE :

This is John Horton Conway's mathematical automata game. Patterns of dots may be created and then set into 'motion' after which they will 'reproduce' and 'terminate' until either a static state or death of all dots results. Dots maybe set with 'I'. Moving commands are: up='U', down=<linefeed>, right=<space>, left=<backspace>, homecursor='H', centercursor='C'. Typing 'S' will start the game.

LIFEDEMO:

An automatic demonstration of LIFE. The game will shoot a 'glider' pattern at a 'pulsar'.

RATMAZE:

This is an entertaining graphics demonstration that randomly generates a maze based upon a three digit input 'code'. (Initials are a good idea.) Upon completing the maze(which, incidentally, has one and only one solution path) it sends a 'rat' through the maze to solve it in a relatively unintelligent fashion. (The busy rat leaves 'droppings' over all paths that he runs.) Program awaits a carriage return once the maze has been solved.

DOODLE :

A 'creative' computer graphic artist program that creates aesthetically(?) pleasing works of art composed of squiggles, dots, circles, spiders and various other patterns.

CAI :

Demonstrates I(nstuctional S(upport package for a mathematics quiz. (Self-prompting)

DEVELOPER:

A utility program that formats FOTOFILES for the PRINTRONIX printer. (See document elsewhere.)

PATT1 :

Dynamic pattern generation using DRAWLINE intrinsic with XOR-mode demonstrating interference patterns.

PATT2 : Similar to PATT1.

SURFACE :

Demonstrates some of the 3-D graphics potential on the TERAK 8510a. First generates necessary data for plotting surface(line of periods written on screen). Then the program plots the surface by contour lines utilizing a simple hidden line algorithm(implemented with DRAWLINE).

U. C. S. D. 'PENNY ARCADE'

BLOCKADE :

A game of competition for two players. Put the keyboard in shift-lock mode(i.e. get the red light on the "lock" button lit). Player on the right has the vector pad 1,5,3,0 to turn his wall. Player on the left uses E,S,F,C to turn his wall. The object is to block off your opponents path so as to leave him as little space as possible. First player to 'crash' into a path or wall loses the point. Game is continuous and winning score is 4. Score will automatically reset to zero at end of match. (To stop the program re-boot the system.) No TEXTFILE is available on this program.

WWII :

A start on the development of an anti-aircraft game. A random number of planes are displayed and subsequently shot down by the computer. Game terminates itself upon wiping out all planes.

HANDBALL :

Handball is a fast-moving two player game using paddles and a ball that speeds up and heads off at unexpected angles. Instructions are provided. (Game plays to 11 points and <return> terminates it.)

BREAKOUT :

A challenging game for one player. The computer sets up a 'brick' wall and it is up to the player to break down as many bricks as he can by successfully aiming the ball at the wall so as to break through. Balls are served by typing carriage return. Learning to use the paddle will require some practice. The '<' and '>' keys move the paddle left or right(with a single press) and hitting the space bar (or any other key) will stop it. For faster paddle speed press the direction keys twice. Once the ball has broken through behind the wall paddle will shorten for an extra challenge. 'Q'<ret> will terminate game.

TANKGAME :

A game of skill and strategy for two players. Game offers a selection of terrain for tank battle. Players maneuver around playing field and fire shells at opposing player's tank. Game is self-prompting and includes a number of variations.

CHASE :

A stimulating cat-and-mouse pursuit game offering a number of playing variations. Players alternately chase and pursue for 30-second periods. Movement is provided in 8 directions. Player on the right uses vector pad keys(7,8,9,4,6,1,2,3) and player on the left uses keys Q,W,E,A,D,Z,X,C to move their respective fighter ships. Typing 'H' will end the game.

- Notes -

```
********************************* ***************
* RT11 to PASCAL CONVERSION KIT * * Section 4.8 *
********************************* ***************
```

Version I.4   January 1978

The utility file labeled RT11.LIST.CODE is intended for use with the RT-11 directory. It assumes the presence of an RT-11 directory spanning blocks 6-7. When the file is executed it inquires for the user to respond with either a 4 or 5 depending on the appropriate volume of which the user wants to view the directory. Once a legal on-line unit has been specified, RT11.LIST.CODE proceeds to read up 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 proceeds to list on the screen the entire contents of the directory. For each entry it specifies the file title with appended file kind, the size of the file in blocks, and the starting block location of the file (in base 10). All unused portions are appropriately identified as such.

The utility program called RT11TOEDIT is for transferring data from a RT11 disk to a PASCAL disk. The program requires a two-drive machine and is self-prompting.

- Notes -

```
******************************** ***************
* HEX-DECIMAL-OCTAL CONVERSION * * Section 4.9 *
******************************** ***************
```

Version I.4   January 1978

This text file contains the declarations necessary for the PASCAL programmer to do these common conversions. A quick look at the code involved in this should explain how to use them. It should also point out some of the neat, powerful things that can be done with UCSD PASCAL data structures.

The code file, when run, will prompt you for input in any one of the bases, and upon your request return it to you, converted to any other of the bases. The program is self-prompting.

- Notes -

```
********************************* ****************
* CHEDIT - THE CHARACTER EDITOR * * Section 4.10 *
********************************* ****************
```

Version I.4 January 1978

CHEDIT is a utility program intended only for use on the TERAK 8510a system. It converses directly with the character generator buffer and enables the user to edit the 192 characters in the 8510a's soft character set.

When executing, CHEDIT displays a 10 X 8 rectangular matrix which represents the workspace in which the user is editing a particular character. A blinking cursor is used to signify location within the matrix.

Following is a description of the available keyboard commands:

G : Get a character. (The character typed following the 'G' will be displayed.)

A : Add this bit. (i.e. turn on the bit at present cursor location)
Z : Zap this bit. (i.e. turn off the bit at current cursor location)
O : Oppose(complement) bit at current cursor position.

C : Clear entire character.
I : Invert(complement) entire character.

H : Home the cursor. (upper left corner)

X : eXchange or switch to other half of character set. (X is a toggle between STANDARD and OPTIONAL character sets.)

M : Move into. This is a multiple character command: Next character must be 'X' or 'G'('X' makes change only temporary). When the 'G' is finally typed the next character to be keyed in will be 'moved into' the character currently being displayed. (e.g. if 'A' is current character, then 'M G a' will duplicate 'a' in the space belonging to the ASCII character 'A'.)
S : Switches(copies) the alternate character set into the current set(STANDARD or OPTIONAL) and inverts(complements) the set. (This is useful for creating a 'reverse video' set with minimum effort.)

Q : Quit CHEDIT (no output results, but leaves character generator buffer as you have altered it.)
K : Keep updated character set (write set to disk as NEWSET.CHARSET and exit program.)

Vector keys : Move cursor in direction specified.
Numeric pad : Vector pad defined in standard format:
5: up; 1: left; 3: right; 0: down;
7: bit off; 8: bit swap; 9: bit on;
2: Alpha lock system wide

It is only possible to edit one half of the complete character set at a time(i.e. either the STANDARD or OPTIONAL set). The X command switches the character set indicator between the two sets.

In running CHEDIT it is important to have the file titled CHEDIT.PROMPT resident on the same disk. CHEDIT.PROMPT is a display file used by CHEDIT to clarify the operation of CHEDIT to the user (displays most commands, status of character set indicator, plus display of current character being edited in several locations for convenient view of the effect of changes being made.)

When the PASCAL SYSTEM bootstraps(or I(nits) it looks for a file named SYSTEM.CHARSET which is then read into the character generator buffer. Changes made to the character set in CHEDIT are onlyu saved permanently if 'K' command is typed. The effect of this is to write out a file called NEWSET.CHARSET to the disk(SYSTEM.CHARSET is retained). In order to use NEWSET.CHARSET it is necessary to change its name to SYSTEM.CHARSET.

```
************* ***************
* DEVELOPER * * Section 4.11 *
************* ***************
```

Version I.4   January 1978

The file entitled DEVELOPER.CODE is a utility program designed for converting FOTOFILES into a form compatible for printing on the PRINTRONIX PRINTER. In essence, DEVELOPER acts upon a data file treating it a seq- uence of bits only. It converts the input FOTOFILE and writes out a file called PRINTER.DATA which can then be transferred directly to the PRINTRONIX PRINTER. DEVELOPER assumes that the input file has been edited on the TERAK 8510a screen and will format any input file as being 320 bits across. Thus the printed file will be printed out as 320 printer bits wide regardless of the printer file. To further generalize the DEVELOPER program it is only necessary to change the 2 occurences of '319' in DEVELOPER.TEXT to whatever row size is wished according to the nature of the way in which the original FOTOFILE has been edited. For the most part, developer will produce an output file (i.e., PRINTER.DATA) which contains approximately twice as many blocks as the original SNAPSHOT file or FOTOFILE. The user should be careful with using larger files as DEVELOPER (if left unmodified) only checks to see if 64 blocks of free space exist on the disk and any output file exceeding this length will be incomplete if not enough space is available.

- Notes -

```
******************************************* ****************
* XREF: PASCAL TEXTFILE CROSS-REFERENCING * * Section 4.12 *
******************************************* ****************
```

Version I.4   January 1978

The utility program entitled XREF.CODE is useful for creating a listing of a Pascal TEXTFILE with an extensive cross-referenced index. XREF writes out a file called XREF.LISTING which is a TEXTFILE that may be transferred to the printer for easy reference. Basically, the cross-referencing catalogs any TEXTFILE by listing, in alphabetic order, all recognizable, non-reserved word identifiers. Alongside each reference there appears a list of the line numbers on which the pertinent identifier occurs.

In running XREF it is only necessary to specify a legal on-line TEXT- FILE followed by the specification of whether a line-numbered listing is wanted along with the cross-referencing(i.e., responding to the prompt: 'LISTING?' with a 'Y' will include a line-numbered listing and 'N' will not). While XREF is executing the only indication the user has of the program's operation is by sounds of the disk-drive and the occasional appear- ance of 'Running...' on the screen to signify that another page(55 lines) has been processed.

We thank SPERRY-UNIVAC for their contribution of XREF.

- Notes -

```
********************************** ***************
* COMP2: TEXTFILE SOURCE COMPARE * * Section 4.13 *
********************************** ***************
```

Version I.4   January 1978

The utility program entitled COMP2.CODE is useful for comparing any two textfiles that need to be checked for differences(that may be time- consuming to find). COMP2 does character-by-character compares and notes any and all differences between two (preferably similar) textfiles. If there exist no differences(i.e., the files are identical) then the user is notified with a message stating such. Otherwise, if a mismatch occurs then the user is prompted in one of two basic ways. Any small internal differences between lines in a file will be signalled with a 'MISMATCH' followed by a listing of the line(s) in each file that were found to dif- fer in any way. If extra text is found present in either file then the user is notified with 'EXTRA LINES OF TEXT' and is shown what excess text occurred where in the pertinent file. Also, empty files and premature end-of-files are signalled by the program.

(* NOTE: IT IS RECOMMENDED THAT THIS PROGRAM BE USED ONLY FOR *)
(* FILES THAT ARE KNOWN TO BE EXTREMELY SIMILAR IN CONTENT. *)

We thank SPERRY-UNIVAC for their contribution of COMP2.

- Notes -

```
***************** ****************
* GOTOXY BINDER * * Section 4.14 *
***************** ****************
```

Version I.4   January 1978

This program alters the SYSTEM.PASCAL on your default P(refix disk. It prompts you for 'local GOTOXY', a procedure which must be created and bound into the system (only once) in order to make your system communicate correctly with your screen.

Look at the file GOTOXY.TEXT on your release disk. This file contains a few procedures for doing GOTOXY cursor addressing on a few different CRT-type terminals. If the procedure you need 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 your terminal is not there, create one. Your procedure may <u>not</u> be named GOTOXY because this identifier is predeclared at the "$U-" level of compilation.

Possible error:	Fix:
Nil memory reference at compile time	Remove the program heading and try again
Value range error when executing BINDER	(*$U-*) should be the first thing in your GOTOXY file

Assumptions:

1.) You have a screen terminal

2.) You have a PASCAL system

3.) The upper left-hand corner of the screen is X=0, Y=0.

- Notes -

*********************************** ****************
* DISK DIRECTORY RECOVERY UTILITY * * Section 4.15 *
*********************************** ****************

February 1978

The utility program RECOVER optimally restores the contents of damaged disk directories, including most of those with "hard" errors such as CRC errors. At present, only textfiles and codefiles are restored, but not systems files or non-editable data files. Undamaged parts of the directory are identified and re-used when possible; the recovery operation only looks at damaged parts of the directory. Since normal damage is only to a few words within the directory, most of the directory is returned to the user in exactly its original condition. When the directory entry of a file is found to be damaged, the file is located on the disk, and its directory entry is restored with the program name serving as the filename. If no program name can be found, the filename used is DUMMYnnX, where nn is a two-digit integer.

In operation, RECOVER first reads the directory using a modified UNITREAD which ignores CRC and other disk errors. The modified UNITREAD temporarily alters the interpreter, so if RECOVER is interrupted during this period, the system will have to be re-bootstrapped. Errors found in the directory read are displayed on the screen.

Next, RECOVER checks the validity of each entry in the directory, by range-checking several different items in the entry. For this, the number of entries is found in the header entry; if invalid, the last valid entry in the directory is the last entry.

For each bad entry, RECOVER first looks at the (possibly still valid) disk address pointed to by the entry. If no file is present there, RECOVER gets the disk addresses from the next previous valid entry and the next following valid entry and uses these as limits in its search for the file. It searches backwards from the higher limit to the lower limit in order to find the most recent update of the file -- this is because, due to the Editor's updating procedure, several obsolete copies of a textfile may precede the most up-to-date copy on the disk.

If the file is found, the directory entry is restored, using the program name as a filename (if none is found, the name DUMMYnnX is used). If the name was previously encountered, then ".nn" is appended where nn is a unique number. If the file is not found, the entry is eliminated. All restored (or originally valid) entries cause the message "FILE filename INSERTED" to be displayed whereas eliminated entries display the message "FILE filename NOT FOUND" followed by "ENTRY n ZAPPED". RECOVER then goes back and processes the next invalid entry.

At the end of the above process, RECOVER will optionally locate additional files on the disk for which an (invalid or valid) directory entry did not exist. Normally the user will wish to bypass this processing, since the additional files in question are probably previously deleted files, which are not wanted back in the directory. The two occasions for which this option will be useful are (1) when the entire directory has been wiped out or zeroed,such that normal processing (described above) is not effective; or (2) when the end of the directory has been damaged and normal processing does not pick up important files which were located at the end of the directory. RECOVER displays the cue, "Are there still IMPORTANT files missing?" for which the response "Y" executes the option and the response "N" does not. If the option is executed, RECOVER finds the last file currently in the directory, goes to that file, and, beginning at the point after the file, scans to the end of the disk, putting each additional file found into the directory.

After the above processing, a reconstructed directory exists in memory. RECOVER displays the cue, "GO AHEAD AND UPDATE DIRECTORY?" and must receive the response "Y" before actually writing anything on the object disk. If a "Y" is received, the write takes place, and if an I/O error occurs, then it is assumed that a sector address has been wiped out on the object disk and that reformatting will be necessary; otherwise, the message "WRITE OK" is displayed.

The method of detecting a textfile is as follows. A null block signifies the possible beginning of a textfile. A block is null if at least the first 20 bytes of the block are nulls. The next non-null block is found. In this block either (1) the word PROGRAM must be found, or (2) 320 editable characters must be found, before 12 noneditable characters are found. (the numbers here can be changed and probably will be.) The beginning of the file is two blocks before the first non-null block.

The method of detecting a codefile is as follows. A codefile must begin with a block containing blanks in bytes 64 to 190, except containing a program name (possibly empty) starting at byte 72.

The end of a textfile is found by searching for a null in the file, either preceded within the previous 10 characters by the string "END." or in a block followed by a complete block of nulls. Textfiles are always rounded up to the next even number of blocks.

The end of a codefile is determined merely by looking for a null block or the beginning of the next file (as determined in the directory). Al Hoffman.

```
************************* ***************
                 * INSTRUCTIONAL SUPPORT * * Section 5.1 *
                 ************************* ***************
```

This document describes instructional support facilities for UCSD Pascal. The package is designed for automated testing or programmed instruction applications. There are two editions of the package: student self-test and formal, graded, quiz. The formal quiz version communicates with the student record-keeping system [described elsewhere] through a hardware dependent interface (for security). Until further notice, the formal quiz system will only be available for Terak 8510a configurations. The self-test edition relies on features of the I.5 release. Information regarding it is included for planning purposes only. For the curious, a preliminary self-test edition is available for the I.4 release. It is similar to the official version described, but a large number of annoying conversions will be necessary in any source written for the preliminary system.

Both editions require a CRT terminal with selective erase and X-Y cursor addressing. Optional supported features include vector or raster graphics and writable character set. Minor adjustments to the support system may be necessary, depending on the terminal characteristics; these will generally involve replacement of some constants. The two editions are compatable with each other, and a unit need only be compiled once to be used under both packages.

The instructional "courseware" applications are programmed in Pascal. The authoring environment is enriched by groups of procedures made available by the support package. These procedures format text, analyze input, produce graphics, make random selections, and track student performance. The graphic and text functions afford a large measure of display independence for courseware, as all coordinates are expressed in logical, author-defined units.

- Notes -

*********************** ***************

* STRUCTURE OF A UNIT * * Section 5.2 *

*********************** ***************

A single instructional program is known as a "unit." It is implemented as a SEGMENT PROCEDURE [see Section 3.3]. (For the self-test, this must be linked into a quiz-running program. The formal quiz system dynamically links the unit at run time). A compiler include file [see Section 1.6] supplies the applicable declarations for use by the author. This file must be included as the first line of the program. It includes program, constant, type, and procedure declarations. (The actual procedure bodies are compiled separately, and either linked or included from a library.) The first author-supplied line is a SEGMENT PROCEDURE declaration for the outer block of the unit quiz.

The unit consists of up to 25 "concept pools." Each pool is implemented as a major procedure within the unit. The outer block of the unit should contain only initialization and pool selection statements. The support package tracks student performance on each pool, including which pools may still be selected. It is also aware of conditions for terminating the unit with overall passing or failing grades. It is up to the author, however, to determine when a concept pool has been passed or failed, and to report this fact.

A concept pool may contain a number of repeatable and nonrepeatable variants. Again, the support package can select from these variants at random.

Common input and output requirements may be simplified by the supporting procedures. Reading and evaluation of numeric responses (optionally including algebraic computation) is provided, as is solicitation of a yes/no answer. The author may pause for a student-determined period (system waits for a space or carriage return to be typed). The usual Pascal output formatting is available, extended somewhat by pseudo-files called viewports. A viewport is a rectangular screen region, onto which text and graphics may be displayed. Viewports automatically "wrap around" to the next line for text (both input and output). If the hardware is equipped with a readable text display buffer, text lines may also be scrolled within the viewport. Graphics going outside the viewport will be clipped at its boundries. A current position is maintained for each open viewport, so that several different screen regions can be accessed concurrently.

- Notes -

************************************ ***************
* DESCRIPTION OF AUTHOR PROCEDURES * * Section 5.3 *
************************************ ***************

Conventions for this document

Variables named XLO, YLO, XHI, YHI, X, and Y may be either INTEGER or REAL. XTEXT, YTEXT, XSCREEN, and YSCREEN must be integers.

The INKCOLOR may be NONE (which will not affect the display), WHITE (make points visible "even on a green screen"), BLACK (make points invisible), or REVERSE (change the dot color). BLACK and REVERSE are available only on screens with selective erase capability.

PICTURE is a string identifying a graphic display segment in the GRAFFILE for the unit. Until the graphic editor is released, this information is for planning purposes only.

STREAM is in the subrange 1..25. Each value represents an independent random selector. Each stream may contain nonrepeating integers between 1 and 99.

V indicates a VIEWPORT name. All input or output for the procedure will then use that viewport, which must have been opened. The author may also do normal READs and WRITEs using any viewport.

Viewport text and graphics

OPENPORT parameters to be described in later release opens viewport V. This feature will be documented when incorporated with the next standard release.

LINETO (V, X, Y, INK) draws a line from the previous screen position to (X, Y). Lines which extend beyond the viewport boundary will be clipped.

POLARLINE (V, SIZE, ANGLE, INK) draws a line from the previous screen position using polar coordinates. The line is drawn at ANGLE degrees for SIZE units. Lines which extend beyond the viewport boundary will be clipped.

AXES (V, XLABEL, YLABEL, XSPACING, XHEIGHT, YSPACING, YHEIGHT, LIMITS) draws coordinate axes for the indicated viewport. The LABELs are strings to be displayed; for no labels specify null strings. The X and Y SPACING are the distance between tic (or grid) marks on each axis; the HEIGHTs are the length of the tics (units of the other coordinates!!). Zero (or negative) height supresses the tics; maximum coordinates will give a full grid. LIMITS is a boolean. If TRUE, the lower and upper limiting values will be displayed for each axis.

DISPLAY (PICTURE) locates the PICTURE and displays it.

> NOTE: A graphics-based editor is now in the design stage. It will create GRAFFILEs for this procedure.

CLEARSCREEN erases the screen. Both text and graphics are erased.

ERASEPORT (V) erases text and graphics from the specified viewport.

WRITEREAL (VALUE, NORMAL) is a specialized version of the real number output routine. It will not use scientific notation, and prints only two decimal places (rounded). NORMAL is a boolean, which, when true, will cause output in the standard character face. Otherwise, superscript characters will be used to display the number.

> NOTE: There are two purposes for this routine.The real number formatter in the system does not always print numbers in the expected standard notation, and real-valued exponents are sometimes useful. The first problem should be cleared up shortly, and this procedure may be eliminated.

Solicitation of Student Responses

YESNO (V, CONFIRM) is a boolean function which accepts a yes or no response. CONFIRM is a boolean; if TRUE, the response will not be accepted until a carriage return is typed. It returns TRUE if the response was YES.

CHOICE (V, CANDIDATE, CORRECT) is a boolean function for concealed multiple choice questions. The CANDIDATE (type string) is displayed in the current screen viewport, ano the student is given a yes/no choice. CORRECT is a boolean with the value TRUE if the candidate is a correct answer. CHOICE will be TRUE if the student response agrees with CORRECT.

EXPRESSION (V, OPTIONS, SOURCE) is a real function which evaluates the algebraic expression in the SOURCE string. It returns the value of the input expression. The student may use variables and functions which have been defined for the session. The permitted operations are

+	addition
-	subtraction
*	multiplication
/	division
^	raise to power
( )	group
\	mod (arguments rounded to nearest integer)

The built-in functions are

sin, cos, tan	standard trigonometric functions - RADIAN measurement
log	log (base 10)
ln	natural log
sqrt	square root
abs	absolute value

Predefined constants:

pi	3.14593
e	2.781828

OPTIONS is a set of (NOTRIG, NOPOWERS, NOCOMPLICATED, NOCONFIRM). NOTRIG disallows sin, cos, and tan. NOPOWERS prohibits use of log, ln, sqrt, and ^. NOCOMPLICATED requires a simple numerical response (no operations). If NOCONFIRM is specified, the student will not be asked to verify a complicated expression.

NUMERIC (V, OPTIONS, ANSWER, RANGE) is a boolean function for soliciting and evaluating numeric responses. It calls on EXPRESSION to evaluate an expression. If the student's answer is within RANGE of ANSWER, the function returns true. A negative RANGE indicates no numerical answer for the question. In this case, a correct answer must contain NO, N'T, NIL, NULL, or EMPTY. If the student does not respond correctly, the answer will be displayed, and NUMERIC returns false. OPTIONS is as defined for EXPRESSION.

PRESSRETURN (V) is a boolean function. It waits for the student to press the RETURN (or space or escape) key, then it clears the indicated viewport. PRESSRETURN is TRUE if the student pressed escape.

READSTRING (V, OPTIONS, DELIMITERS, DATA) allows a line of input to be captured under control of the quiz system. The input is returned in the string DATA.

NOTE: This procedure has been totally replaced by ported READs .... this documentation is obsolete and nearly useless.

OPTIONS is a set of (NOECHO, NOSHIFT, ONECHAR, NODELIMITER, NOTAHEAD, NOCOMMANDS, NOBLANKS). If NOECHO is set, the student's input will not be automatically displayed. NOSHIFT leaves input in its original case; otherwise all letters will be made upper-case. ONECHAR specifies that only a single character is to be read; otherwise a delimiter character will terminate input. NODELIMITER removes the terminating character (except ONECHAR mode). If NOTAHEAD is specified, any queued input characters will be discarded. NOCOMMANDS disables the system directive mode. If NOBLANKS is set, leading and trailing blanks will be removed from the input.

DELIMITERS is a set of characters used to signal the end of input (if not ONECHAR mode).

If NOCOMMANDS is not specified, a number of system directives may be invoked by the student. These are all armed by the '#' character. The command will be processed automatically, and only expected input will be returned to the calling program. These commands are available from YESNO, NUMERIC, and PRESSRETURN. The following are recognized:

STOP the student wishes to terminate the session. Quizzes are marked stopped (not passed).

COUNT a summary of right and wrong responses is presented.

NOTE the remainder of the line is captured as a comment for the instructor.

HELP the HELLO picture (from STARTQUIZ) will be displayed again to remind the student of special notation and restrictions.

MORE! (mostly a debugging function) allows the student to continue with the quiz until all the questions are exhausted. Once selected, this option may not be revoked.

Pseudo-random Selections

RANDOM (LO, HI) is an integer function returning a value LO <= RANDOM <= HI.

DEAL (STREAM) This integer function returns an unselected entry from the indicated STREAM. The value will be between 1 and the stream size (set in MAKEPOOL). Maximum value is 99.

SELECT (POOL) is an integer-valued function used to select question pools. POOL will be selected if it is non-zero, otherwise a random choice will be made from the available pools. This function also displays the question number and title on SCOREBOARD.

RESETLAST (STREAM) makes the last choice in STREAM eligible to be re-selected.

Reporting and Control Functions

LOGIT (DIFFICULTY, GOTIT) records the response to a given question. GOTIT is a BOOLEAN: TRUE indicates a correct response. If GOTIT, a message will be generated equivalent to "good." The DIFFICULTY is an integer between 0 and 3, which determines how strong the message should be (3 is the strongest). If DIFFICULTY=0, no message will be displayed. The message is written to PROMPTLINE.

GOOD (V, DIFFICULTY) displays "good" messages to the specified viewport. DIFFICULTY is an integer between 1 and 3, with 3 giving the strongest encouragement.

MAKEPOOL (POOL, SIZE, RETRIES, DESCRIPTION) associates the string DESCRIPTION with the indicated question POOL. The title may not be longer than 20 characters. SIZE is the number of entries to be made for DEALs from the associated random stream. Streams are limited to 99 entries. The pool selection mechanism permits a question to be missed and re-tried. RETRIES is the number of retries permitted before the question is eliminated. MAKEPOOL allows the question pool to be SELECTed. Should a random stream be required without a question pool, the DESCRIPTION must be an empty string. All MAKEPOOLs for actual question pools should appear before the STARTQUIZ call. This ensures that the student will be told the correct maximum number of questions to be presented.

STARTQUIZ (MAXWRONG, ALLWRONG, MAXMINUTES, DOALL) initiates the quiz session and performs the necessary student handshaking procedures. MAXWRONG is the number of incorrect responses to fail. MAXMINUTES is the quiz time limit. The quiz will automatically terminate after the specified period. DOALL selects the quiz ending strategy. With DOALL false, the quiz may be terminated after MAXWRONG incorrect answers. DOALL true continues the quiz until all questions have been eliminated. A future version will open the graphics file associated with the unit, and display the HELLO picture, if any.

QUIZDONE is a boolean function. It determines (from the STARTQUIZ parameters) when a quiz is concluded. It will either return FALSE (not done) or report the quiz result and stop the program. Before returning to the author program, QUIZDONE performs a PRESSRETURN using PROMPTLINE, and a CLEARSCREEN. The quiz will abort if the studsnt escapes at this time.

* DESIGNING (AND DOCUMENTING) A LESSON FRAME * * Section 5.4 *

(1) Educational Goal and Method

First, you should decide the general goal or topic of the frame. State this goal in the context of the unit. Next, find an approach to achieve the goal. This should indicate the type of question, kind of response required, and feedback for the student. Non-obvious scoring should be explained. Notice that so far you haven't said anything about automation.

(2) Programming Analysis

Now you want to describe a stategy for generating the questions and feedback. When necessary, this may be partially in terms of the implementation. Only the major mechanisms should be covered here.

(3) Animation

Lay out the screen format. Block out a number of viewports. Show their relative position, size, and contents. Spend some time on this; make it attractive and convenient to watch. Consider using lines and boxes to dramatize the spatial design.

(4) Coding

Now you have developed a functional spec. The coding will be a relatively direct matter in most cases. The specification will also serve as the main documentation for the frame. Add notes to explain coding tricks and funny variables.

In testing situations, think in terms of "question generators." These are procedures which can create many variations of the same basic problem. The question generator may have some options, set via parameter, to allow one generator to serve in more t..an one frame. The generator itself should rely on random numbers and pools. Question generators should return the correct answer(s) to their caller.

As usual, the frame main procedure should be short. Call on local procedures for lengthy calculations or screen formatting.

Sections 1 through 4 indicate successive job steps. Work may be delegated after any step.

SAMPLE DOCUMENT

<< any similarity to existing programs, living or dead, is purely incidental >

UNIT 1 - QUESTION 1

TITLE: <identifier>s

APPROACH:

The student states whether or not various character strings are <identifier>s. If the student answers incorrectly, the right answer will be given.

Remarks, such as "spelling is irrelevant," "lower case prohibited," "must start with a letter," "spaces not permitted," etc., should be displayed as appropriate.

SCORING:

Question pass = 10 in a row correct Question fail = total of 5 wrong. May be repeated once.

STRATEGY:

A selection of one from a group of 50 pre-determined strings will be made. A remark and correctness is associated with each string.

SCREEN FORMAT:

```
                    # right and wrong (SCOREBOARD)

                                               question/
                                                answer

                        instructions

                                               remarks
```

NOTES:

Two lists of 25 strings are kept. One contains only valid <identifier>s, the other, only invalid. COMMENT[I] corresponds to both VALID[I] and INVALID[I].

****
PROCTOR'S GUIDE TO BOOKKEEPER
****

This document is an introductory guide to BOOKKEEPER, which is a special version of the Pascal system used to perform the record keeping for the self- paced computer science courses taught at UCSD. This document is currently not a comprehensive description of this bookkeeping system, but is instead, as the above title suggests, an introductory user guide to proctors. In its current form, this document is organized as a collection of sections each of which covers a particular aspect of the bookkeeping system. These sections describe the basic aspects of the bookkeeping system which new proctors would need to understand as part of the process of becoming familiar with the role of a proctor. Therefore this document does not include any disscussion of the internal operation of the bookkeeping system, nor any information on maintenance of the system.

The software described by this document is not included in the I.4 release of the U.C.S.D. Pascal System, although it is currently being used at U.C.S.D. We hope to have this software in a releaseable condition by the time of the next release of the system.

>>>>>>> ATTENTION WINTER 1978 PROCTORS <<<<<<

YOU SHOULD TAKE THE TIME TO READ SECTIONS 7, 8, 9, 10, AND 11 OF THIS DOCUMENT. SUBSTANTIAL ADDITIONS AND REVISIONS HAVE BEEN MADE TO THESE SECTIONS.

****
TOPICS DISCUSSED IN THIS DOCUMENT
****

1. BOOTSTRAP PROCEDURE
2. CARE OF BOOKKEEPING DISKS
3. SECURITY
4. TRANSACTIONS UPON AND UDPDATING OF STUDENT RECORDS
5. MESSAGES FOR PROCTORS IN A STUDENT'S RECORD
6. SELECT COMMAND
7. ZAP COMMAND
8. ADMINISTRATION AND RECORDING OF FACTUAL (AUTOMATED) AND PROGRAMING QUIZZES
9. SETTING THE COURSE EVALUATION (CAPE) ENTRY
10. COMMENTS ON PAPER GRADE RECIEPTS
11. UTILITY PROGRAMS IN THE BOOKKEEPING SYSTEM

1. BOOTSTRAP PROCEDURE

The bootstrap procedure is identical for both of the bookkeeping machines, with the exception that different disks are used. On the main bookkeeping machine(the machine with the two disk drives) use the disk called BOOKER:. The disk should do its usual amount of clicking, and then the following lines should appear:

```
        U.C.S.D.   Bookeeping System Master     System I.3
```

```
Enter today's date:  <1..31>-<JAN..DEC>-<00..99>
--->
```

Once you have the machine to this state, you should enter the correct date and time in the manner indicated. Note that bookkeeper asks you to verify that the date and time are correct immediately after they are entered just in case you make a typographical error.

The date and the time are vital to the correct operation of the bookkeeping process, and therefore we ask that you be very concientious about entering them correctly.

After the date and the time have been successfully entered the screen should clear and the following prompt should appear:

```
Bookkeeper: T(transact, A(uto, M(aintain, O(ption, U(tility, Q(uit
```

You will notice that the disk will perform a substantial number of "clicking" operations as the master bookkeeping files are opened. Once you have the machine at this stage, you will have to "unlock" its "front panel" as described in the section called SECURITY below.

2. CARE OF BOOKKEEPING DISKS

The disks which will reside in the lab for bookkeeping purposes are not to be used for any other purpose, and should be used only in the machines which are designated as bookkeeping machines. You as a proctor are the only person who should be handling these disks.

Please keep these disks inside their floppy envelopes when not in use. Also don't leave these disks lying around on desk tops where someone can either sit on them or place something heavy on them. In short, keep these disks in the drawer where they belong.

THE FOLLOWING ARE TWO IMPORTANT INSTRUCTIONS CONCERNING THE MAIN BOOKKEEPING MACHINE:

1. DO NOT REMOVE THE DISK FROM THE BOTTOM DRIVE ON THIS MACHINE UNLESS IT IS A DIRE EMERGENCY THAT YOU DO SO !! WE DONT WANT TO RISK GETTING BAD BLOCKS ON THAT DISK.
2. WHEN SHUTTING THE SYSTEM DOWN FOR THE NIGHT, DO NOT JUST POWER OFF THE MACHINE. USE BOOKKEEPER'S Q(uit COMMAND FIRST. (This saves the transaction logfile which is lost if the Q(uit command is not used)

3. SECURITY

To prevent unauthorized persons from entering bookkeeping transactions, bookkeeper has a security system which requires that a person type in a password to "unlock" the front panel, before certain commands will work.

Just after being bootstrapped, bookkeeper's front panel is locked, When the panel is locked, records may be examined, but not altered. To unlock the front panel, a proctor must use a hidden command called S(ec. S(ec responds with the following promptline:

```
Sec:  N(ew, C(urrent, Q(uit
```

C(urrent displays the current security level number. N(ew is used to enter the password for a new security level. N(ew responds with the following promptline:

```
Enter new sec followed by [RET]
```

Bookkeeper is now waiting for you to type in your password. Note that the characters that you type will not be echoed back to the screen. When you have completely typed in your password press the RETURN key. You will know whether or not your password was accepted or not by the following line:

```
Sec changed to  1
```

You should see a "1" instead of a zero. If you don't succeede the first time you may try again as many times as you like.

Once you have accomplished your task of unlocking the front panel you may use the Q(uit command to exit the security changer and return to the main part of bookkeeper.

NOTE: Even after the front panel has been unlocked in the manner described above, there will be certain commands listed on the promptlines of bookkeeper which will not be allowed to you. These particular commands are reserved for persons who are maintaining the bookkeeping system, or who have a thorough understanding of how the program operates.

4. TRANSACTIONS UPON AND UPDATING OF STUDENT RECORDS

Changes to a students record can be made by entering T(ransact mode. You must then identify the student whose record you wish to update. To do so you may either type in the name or you may type in the student's bookkeeping number. If the name you type is not the name of a person in the file, then bookkeeper will display a short list of names which are closest to the name entered. The F(orward command will cause successive groups of names which are progressively higher in the alphabetic sequence. In a similiar manner, the B(ack command allows you to examine the student roster in the opposite direction.

Once you have successfully identified which student record you wish to update bookkeeper will display the contents of the record and then display the following promptline:

```
(unit #), C(hange, S(elect, A(uto, Z(ap, F(orfeit, E(xam, G(rd, M(essage, Q(uit
```

Most often you will want to record a homework taken, or the passing of a quiz. To do so, you first enter the unit number followed by the items that you wish to add to the student record. Once you enter the first digit of the unit number the following promptline will appear:

```
     Unit # & A-J(quiz) or W(hmwk) or P(ass),  [SP] to accept, [ESC] to abort
```

As the above promptline suggests, the letters A-J will be used to record quiz versions taken. We will adopt the convention that the letter A will be used to record a student as having passed an automated "factual" quiz, while the letters B-J will be used to record versions of the programming quiz taken by the student. The letter W will as in past quarters signify that the student has completed the homework for the unit. " P " will be used to indicate that the student has passed one of the programming quizzes for that unit. In the case where a particular unit does not have either a factual quiz or a programming quiz, the proctor should jus' fill in the "A" and "P" fields of that unit as necessary in order to convince bookkeeper that the unit has in fact been completed. (Note that the units requiring this special treatment tend to change from quarter to quarter).

As you will probably notice when you start to use this system, the bookkeeper software displays warning "flags" next to units that it believes have not been completed by the student prior to advancing to later units. (The "flag" is the appearance of "<---Warning !" to the right of the display for the unit) These warning flags are intended to catch the eye of the proctor so he/she can investigate the reasons for the "holes" in the students record.

Below are some example bookkeeping entries and explanations of what they instruct the bookkeeping system to do:

3AWP4ADWP instructs bookkeeper to record that the student has passed the automated quiz, the homework, and the written quiz for unit 3. The rest of the command string indicates that the student has passed the automated quiz, the homework, and written quiz D for unit 4.

Once you have entered a command string similiar to the one in the example above, you may cause the students record to be updated by typing either spacebar or RETURN. If you make an error you may backspace over your errors, and try again, or you may hit ESC to throw away the comnand sequence and start all over again.

5. MESSAGES FOR PROCTORS IN A STUDENT'S RECORD

Associated with each student record is an optional 50 character message which can be used by proctors to communicate to one another any unusual circumstances concerning any particular student. If a message is already present then the flag " *** MESSAGE *** " will appear in the upper right hand corner of the display of the student's record. You may view this message using the M(essage command. To change the message, or remove a message, you must enter the change mode by using the C(hange command. Once in the change mode, you request to change the contents of the message by using the M(essage command.

6. S(elect COMMAND

The S(elect command is a means whereby the proctor can ask the bookkeeping system to randomly pick a version of a quiz for a particular unit.

7. Z(ap COMMAND

Zap allows you to selectively erase mistakes that have been made in the process of making transactions upon a students record. Zap allows you to enter the items which you want erased in exactly the same manner as items are made part of the student's record.

To erase the "Automated Quiz Taken" and "Programming Quiz Taken" messages from a particular unit, first indicate the unitnumber and then type "A" or "P" respectively. One can think of this two messages as "covers" which cover-up portions of the display for that unit. When one of these messages is present, a zap command which includes the corresponding letter (i.e. "A" or "P") will cause the covering message to be removed, leaving the underlying information intact.

8. ADMINISTRATION AND RECORDING OF FACTUAL (AUTOMATED) AND PROGRAMMING QUIZZES

To give a student an automated quiz, first go into T(ransact and bring the student's record onto the screen. Then use the A(uto command, which will respond with the following prompts at the top of the screen:

```
Automated Quiz Administration:
Enter unitnumber for quiz --->
```

Once you are at this stage, enter the number of the unit desired followed by a carriage return. (If you just type RETURN then you will return to the normal transaction level.) Once you type in the unitnumber, the following prompt will appear:

Enter quiz version desired --->

Quiz version "A" has been designated to record the taking of a "factual" or Automated quiz, while quiz versions "B" thru "J" are used to indicate the taking of one of the programming quizes for that unit.

Once you type in the version letter, the quiz disk in the top drive will be "armed" or enabled, and the students record number is written onto the quiz disk, along with the unitnumber and quiz version to be administrated. If an I/O error occurs during the transfer of this information onto the quiz disk, then the following message will appear:

Error: No Quiz Disk In Drive !

In addition, the students record is updated to show that he/she has been given a quiz disk. On the display of a students record, this shows up in the form of one of the following messages:
"Automated Quiz Taken" or "Programming Quiz Taken"

Note that the above message appears on the display in the area normally used either for quizzes or other purposes . When the student brings the quiz disk back, this message will disappear. If the disk is not returned, then the message will remain in the display of th student's record until the zap command is used to erase this message. (See the section of this document describing the Zap command for details)

When the student returns from the quiz room with the disk, you should place the disk into the top drive, return the bookkeeper system to the Bookkeeper: level, and then use the A(uto command at that level. The A(uto command should give you the following prompts:
Automated Quiz Record Retreival: (type ESC to escape)
Place quiz disk into top drive and press RETURN

As before, ESC will cause you to return to the Bookkeeper: level, whereas RETURN instructs bookkeeper to attempt to read a quiz report from whatever disk in the top drive. If you type RETURN the bookkeeping system first checks to see that you have in fact placed a disk into the top drive. If it discovers that there is no disk in the top drive then the followino error message should appear on the screen:
Er.or: No Quiz Disk In Drive !

However, please note that whatever disk you put into the top drive will top drive will be interpreted as a quiz disk! If you place any other disk than a quiz disk into the top drive you will probably see the screen fill with garbage characters, which typically kills the system, thus requiring that you bootstrap the system again.

If you do in fact have a quiz disk in the top drive, then you will receive a quiz report on the screen which gives you information as to what occurred in the quiz room. (Further discussion of the contents of this report is beyond the scope of this document)

When you are finished looking at the quiz report, hit the spacebar. Bookkeeper will then enter the T(ransaction level and display the updated student record. This allows you to inspect the record and verify that the bookkeeping system has updated the record correctly. Please note that the recording of the passing of automated quizzes is done automatically by bookkeeper.

IMPORTANT: EVEN IF A STUDENT FAILS A QUIZ, YOU SHOULD STILL TAKE THE TIME TO GO THROUGH THE ABOVE PROCEDURE IN ORDER THAT THE BOOKKEEPING SYSTEM CAN RECORD THE FACT THAT THE QUIZ DISK WAS RETURNED BY THE STUDENT. OTHERWISE, THE STUDENTS RECORD WILL STILL CONTAIN EITHER THE "Automated Quiz Taken" OR THE "Programming Quiz Taken" MESSAGE, WHICH IS MEANT TO INDICATE THAT THE STUDENTHAS A QUIZ DISK IN HIS POSSESSION.

9. SETTING THE COURSE EVALUATION (CAPE) ENTRY

Every student record has one bit which indicates whether or not the student has filled out a CAPE (Course And Professor Evaluation) card. If this bit is not set then the message "CAPE CARD NOT COMPLETED" will appear as part of the display of the student record. Note that a student is not required to fill out one of these cards. This CAPE card indicator is primarily intended to promote a larger response by attracting the proctor's attention to the fact that the student has not yet filled one out. (The proctor can then ask the student if he/she would mind filling out one of these cards.)

To turn off the "CAPE CARD NOT COMPLETED" message you must enter the C(hange level by typing "C" from the T(ransaction level. Once inside of C(hange, you use the E(val (short for "evaluation") command, which simply asks the proctor whether or not the student has completed the CAPE survey. (One responds to this question by typing "Y" for yes, and "N" for no.

Since the CAPE survey is performed during the last few weeks of the quarter the "CAPE CARD NOT COMPLETED" prompt will not appear until the maintainer of the bookkeeping system sets an option which informs the bookkeeping system that CAPE is in season.

10. COMMENTS ON PAPER GRADE RECIEPTS

It is important that all of the proctor's realize that the paper reciepts which are filled out by the proctor when passing a student on the quiz or homework for a unit serve as the ultimate means of backing up the bookkeeping system. Also if any dispute arises over the legitimacy of the records maintained on the bookkeeping machine, these reciepts must be used to verify the records in the computer. Therefore, due to the important role assigned to these grade receipts, it is essential that they be filled out completely, neatly, and accuratly.

When filling out these reciepts, please sign the receipt using at least your last name, don't just use your initails as your signature since initials are easily forged. Also make sure that the information on the receipt is complete, especially important are the student's name and bookkeeping identification number.

11. UTILITY PROGRAMS IN THE BOOKKEEPING SYSTEM

Included within the bookkeeping system is a set of utility programs which allow proctors to do non-bookkeeping activities on the same machine on which the bookkeeping system is running.

The following is a list of utilities not provided in the current version of the system:

S(cheduler -(not enough space inside master bookkeeping machine)

W(aiting -(not yet fully developed)

P(rinter -(currently a utility program running on the Terak connected to the printer. See document "How To Get Listings" for details)

F(iler -a modified version of the standard system filer. Does not allow one to remove or create files on the bookkeeping disks. Also does not have a T(ransfer command, due to lack of space. Z(ero command is more intelligent, and won't allow you to zero the bookkeeping disks.

T(ransfer -program which accomplishes the same result as the T(ransfer command in the standard system filer. Note that this Transfer program is oriented for operation on a single drive machine, and therefore prompts you to place either the source or destination disk into the drive, depending on what disk it requires. The reason for the disk swapping is again due to a lack of memory space. If you accidentally place the wrong disk into the drive, an error message is written and the transfer is aborted. If there is insufficient room on the destination volume, then a warning message is written and the transfer is aborted.

Current version of T(ransfer will cause the entire bookkeeping system to go out to lunch if you attempt to use PRINTER: as the destination file. Also, T(ransfer insists that you provide the volume name when entering the filename.
You exit the T(ransfer program by just typing RETURN in response to the "Transfer what file ?" prompt.

C(alculator -Dale Ander's calculator program. This is generally a very useful program, but it is a potential hazard to the bookkeeping system. Particularly troublesome are the floating point math errors which occur when you attempt to use the exponentiation operator in a careless manner.

```
***************** *********
*EXECUTION ERRORS* *TABLE 1*
***************** *********
```

Version I.4   January 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 (XSYIOER)	FATAL
10	User I/O error (XUIOERR)	
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).

- Notes -

```
************ ***********
*IORESULTS * * TABLE 2 *
************ ***********
```

Version I.4   January 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

- Notes -

```
*************** ***********
* UNITNUMBERS * * TABLE 3 *
*************** ***********
```

Version I.4   January 1978

NUMBER	VOLUME NAME	
0	<empty>	
1	CONSOLE	
2	SYSTERM	
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).

- Notes -

```
************* ***********
* PENSTATES * * TABLE 4 *
************* ***********
```

Version I.4 January 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>

- Notes -

```
*********   **********************************************
*TABLE 5*   *SYNTAX ERRORS NOT FOUND IN JENSEN AND WIRTH*
*********   **********************************************
```

Version I.4   January 1978

ERROR #	MEANING
398,399	Implementation restriction.
400	Illegal character in the source text.
401	Unexpected end of input file.
402	Error in writing code file.
403	Error in reading an include file.
404	Error in opening info,list or code file

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.

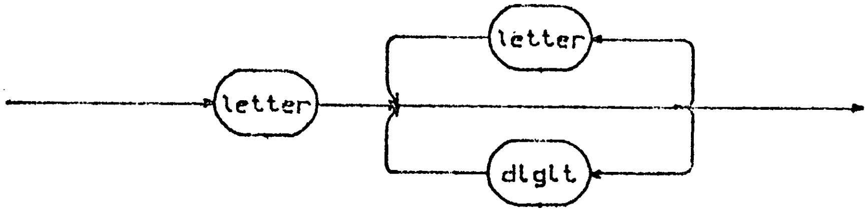
- Notes -

```
*****************************************************************
* TABLE 6 American Standard Code for Information Interchange *
*****************************************************************
```

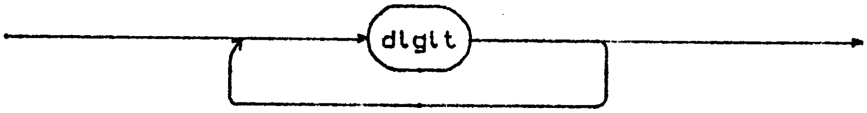
0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	`
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	68	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	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	/	89	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

Table 7 SYNTAX DIAGRAMS

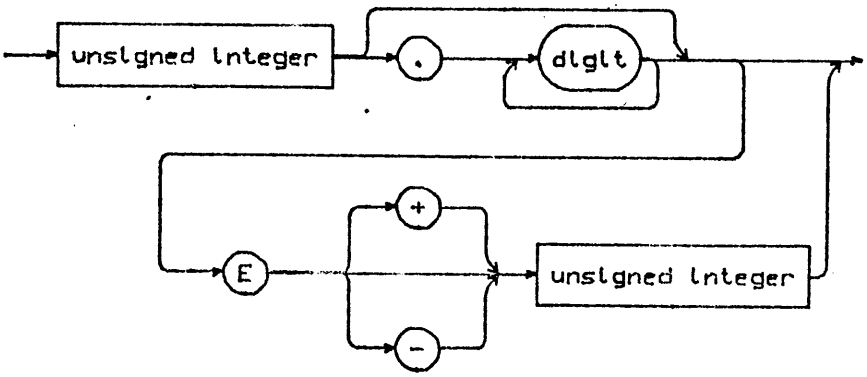
<identifier>



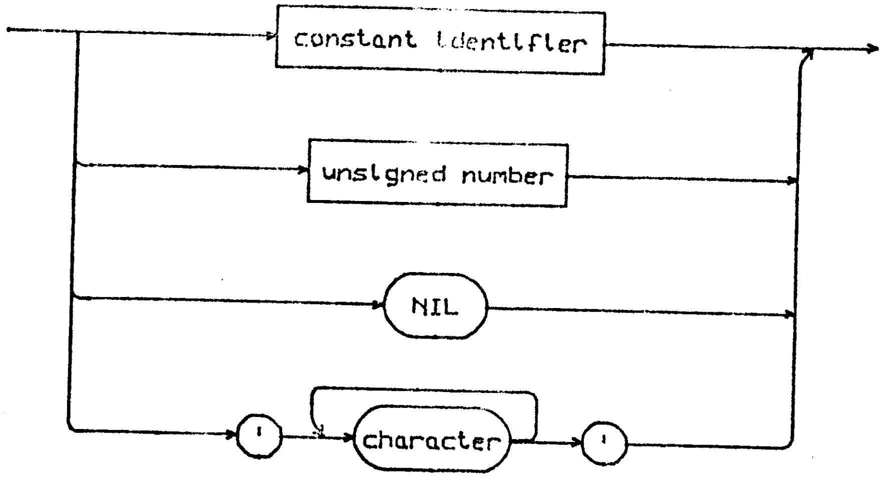
<unsigned integer>



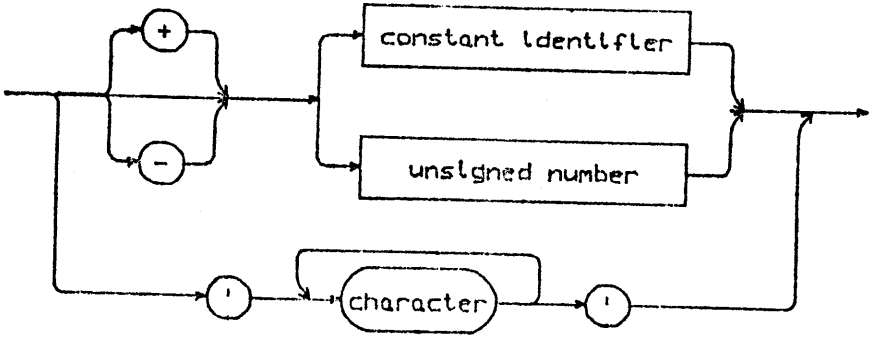
<unsigned number>



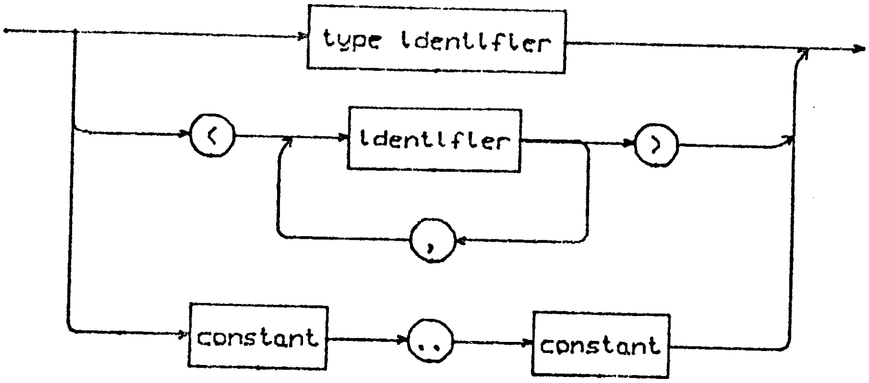
<unsigned constant>


constant identifier
unsigned number
NIL
'
character
'

<constant>

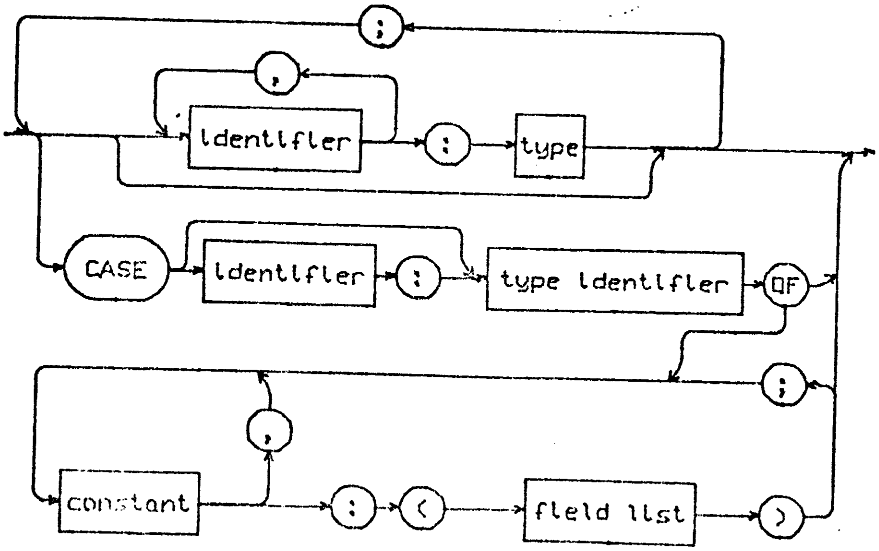

+
-
constant identifier
unsigned number
'
character
'

<simple type>



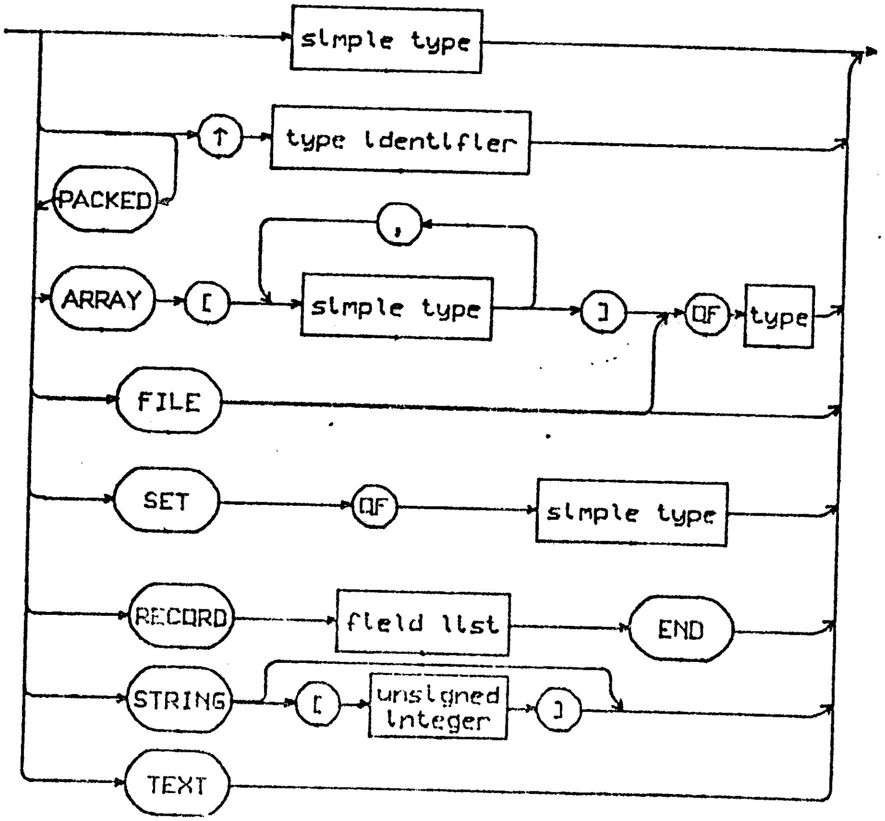
type identifier
(
identifier
)
,
constant
..
constant

<field list>

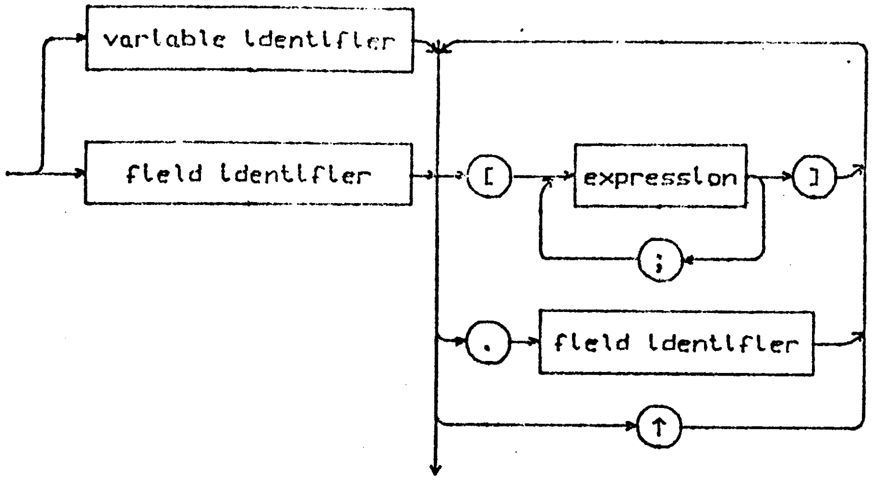


;
,
identifier
:
type
CASE
identifier
:
type identifier
OF
;
,
constant
:
(
field list
)

<type>

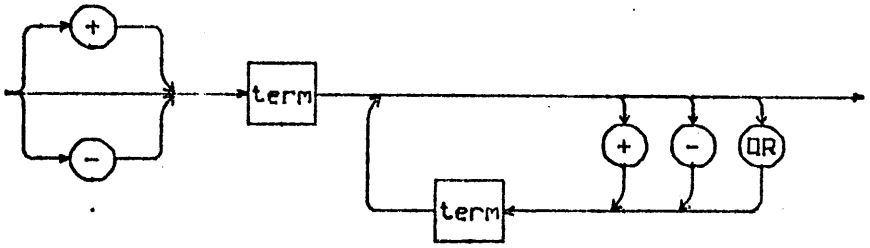

simple type
↑
type identifier
PACKED
,
ARRAY
[
simple type
]
OF
type
FILE
SET
OF
simple type
RECORD
field list
END
STRING
[
unsigned integer
]
TEXT

<variable>



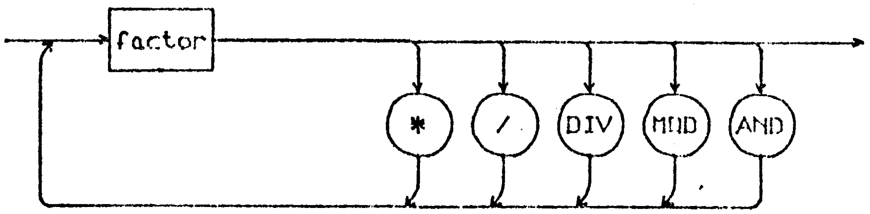
variable identifier
field identifier
[
expression
]
;
.
field identifier
↑

<simple expression>



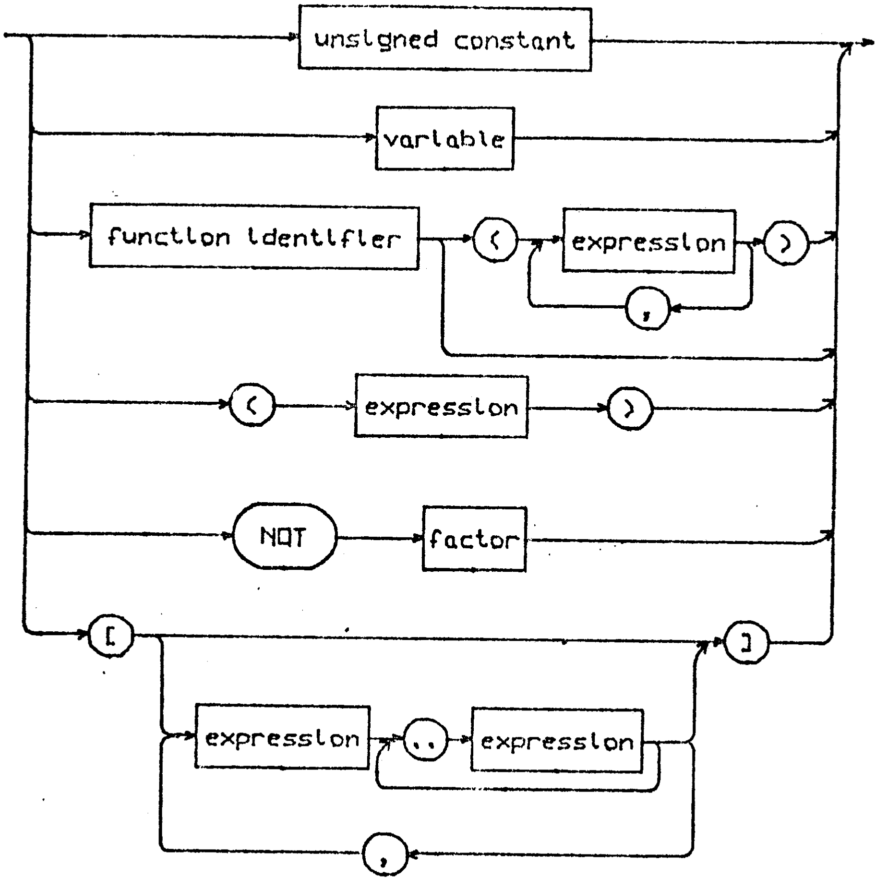
+
-
term
+
-
OR
term

<term>

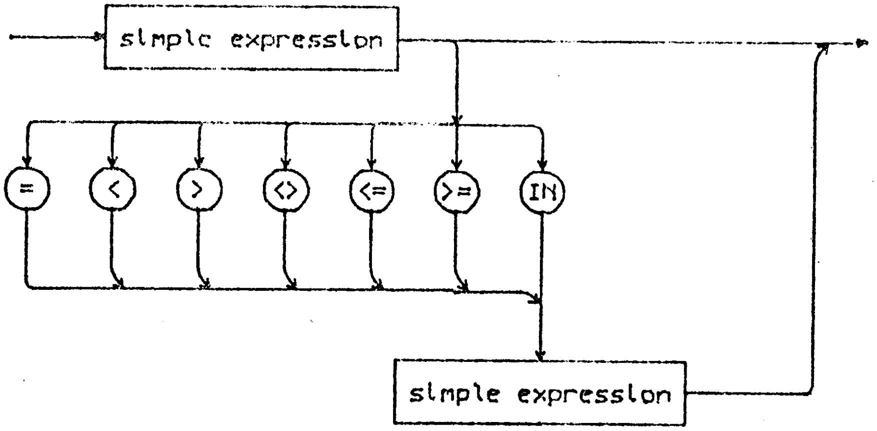


factor
*
/
DIV
MOD
AND

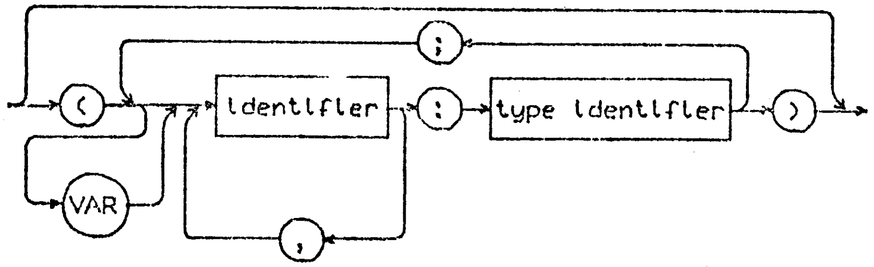
<factor>

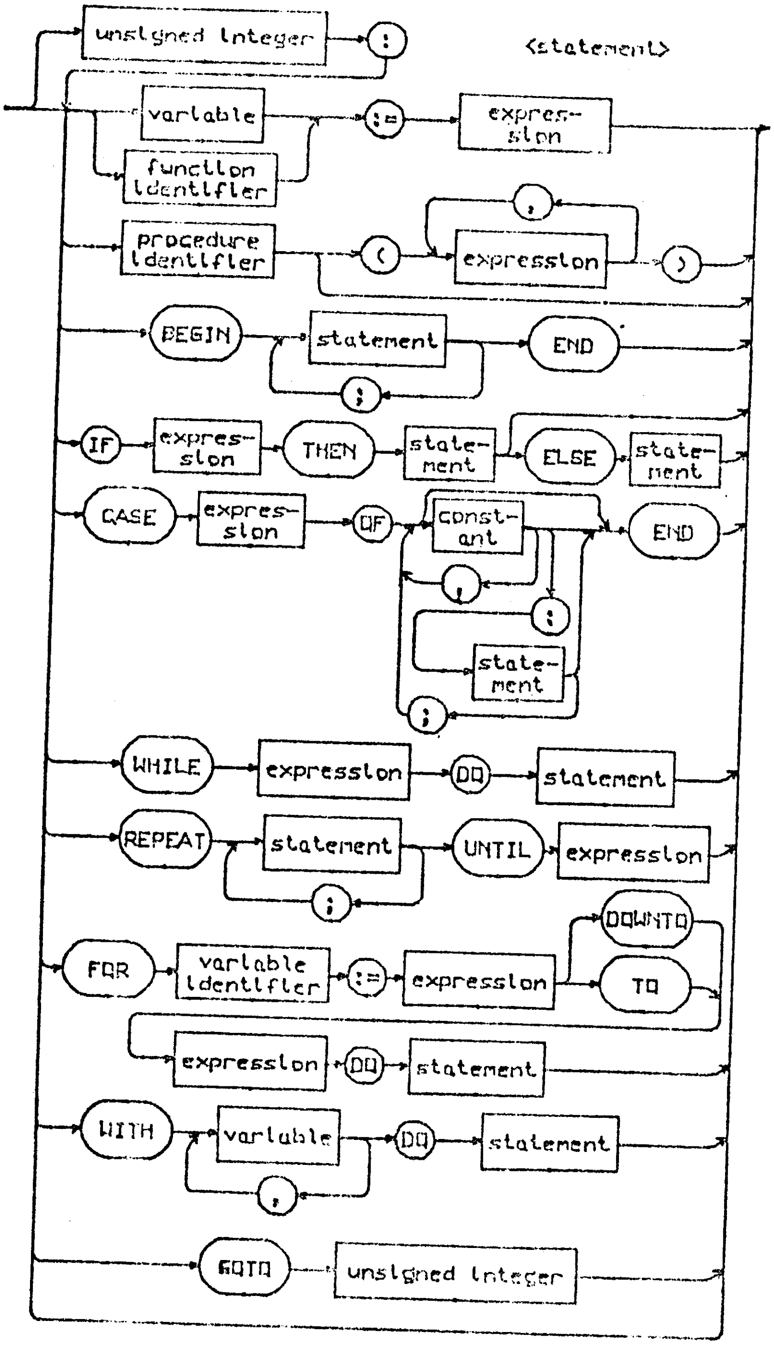

unsigned constant
variable
function identifier
(
expression
)
,
(
expression
)
NOT
factor
[
]
expression
..
expression
,

<expression>


simple expression
=
<
>
<>
<=
>=
IN
simple expression

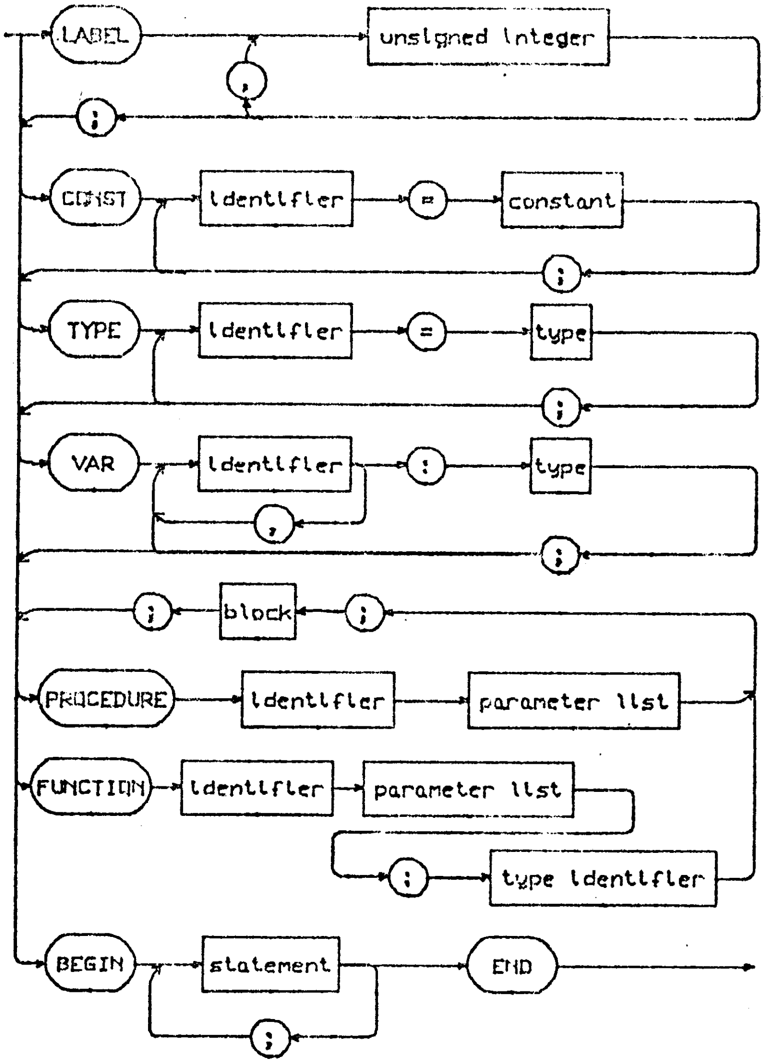
<parameter list>


(
VAR
identifier
,
:
type identifier
;
)

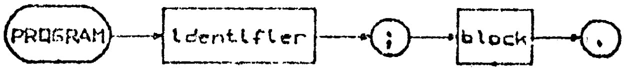


unsigned integer
:
<statement>
variable
function identifier
:=
expression
procedure identifier
(
expression
,
)
BEGIN
statement
;
END
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant
,
:
statement
;
END
WHILE
expression
DO
statement
REPEAT
statement
;
UNTIL
expression
FOR
variable identifier
:=
expression
DOWNTO
TO
expression
DO
statement
WITH
variable
,
DO
statement
GOTO
unsigned integer

<block>


LABEL
unsigned integer
,
;
CONST
identifier
=
constant
;
TYPE
identifier
=
type
;
VAR
identifier
,
:
type
;
;
block
;
PROCEDURE
identifier
parameter list
FUNCTION
identifier
parameter list
:
type identifier
BEGIN
statement
;
END

<program>


PROGRAM
identifier
;
block
.

```
1:Error in simple type
2:Identifier expected
3:'PROGRAM' expected
4:')' expected
5:':' expected
6:Illegal symbol (maybe missing ';' on the line above)
7:Error in parameter list
8:'OF' expected
9:'(' expected
10:Error in type
11:'[' expected
12:']' expected
13:'END' expected
14:':' expected
15:Integer expected
16:'=' expected
17:'BEGIN' expected
18:Error in declaration part
19:error in <field-list>
20:'.' expected
21:'*' expected
50:Error in constant
51:':=' expected
52:'THEN' expected
53:'UNTIL' expected
54:'DO' expected
55:'TO' or 'DOWNTO' 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:Unsatisified 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
```

```
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:Substition 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
173:Fortran procedure or function expected!
174:Pascal function or procedure expected
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 assinged 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
```