UCSD

PASCAL
System II.0
User's
Manual
March 1979

Institute for
Information
Systems
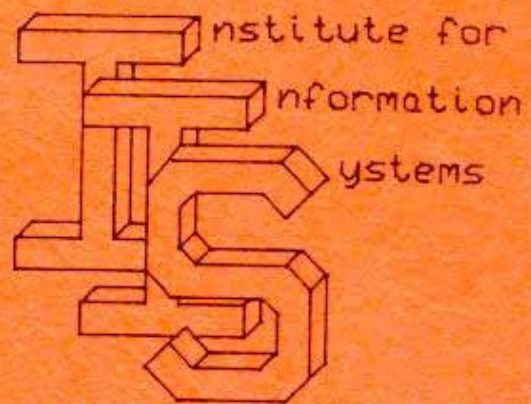
```
*************************************
* UCSD (MINI-MICRO COMPUTER)  PASCAL *
* VERSION  II.0          MARCH 1979 *
* Institute for Information Systems *
* UCSD Mailcode C-021               *
* La Jolla, CA  92093               *
*            (714) 452-4526         *
*************************************
```

Release serial number: _____ 4135 _____

```
********************
* TABLE OF CONTENTS *
********************
```

Version II.0   March 1979

SECTION   PAGE

```
*************************** **************
* MATERIALS AVAILABLE * * Section A.1 *
*************************** **************
```

As the UCSD Pascal system has grown, we have found that to
distribute all of the software which is useful to all users for all
systems, has become an unbearable task. To attempt to alleviate the
large number of diskettes the release software requires, and to
alleviate the number of pages of documentation sent to each subscriber,
we have started to split the system into a number of seperately
available sections.

The major section is the section which contains the operating
system and all the support routines that go with it. We include a
number of useful utilities which should enable the subscriber to do all
types of developmental work. The master release (as from herein it
shall be named) contains the interpreter for the initial system
ordered, the UCSD Pascal operating system, the Pascal compiler, two
text editors (one for screen devices, one for general purpose), a
BASIC compiler, the Linker, the Assembler for the appropriate machine
(at least). Other utilities include: a generalized file utility (the
File handler), a generalize patch and dump routine, a set of programs
to enable the subscriber to configure the system to run most
intelligently with any terminal, a desk calculator, and a librarian.

Software which is not included in the master release is
generally available from the IIS as a supplemental package at a nominal
handling charge (dependent on the amount of material involved with the
package). The sorts of software available are: interpreters for
machines other than the machine the master release was ordered for,
which will be accompanied by the assembler for that machine, in some
cases we have assemblers for machines for which we do not yet have
interpreters, program and data management systems, specifically a cross-
referencer, and a pretty-printer.

- Notes -

```
*************************** *****************
* THE FIRST TIME THROUGH * * Section A.2.1 *
*************************** *****************
```

Version II.0   March 1979

       Welcome to UCSD PASCAL. If you put the Advanced System I
diskette in your booting drive, went through your normal boot-
strapping procedure, and were greeted in a similar fashion, you do not
need to read this section.

       If this is not the case then here are a few of the problems we
encountered with I.4, and I.5 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 do not have enough memory. The minimum requirement for
    memory is 24K 16-bit words.

3.) 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.

4.) You have a system configured for RK and RX and the RX or RK is
    not present. Both must be present at the standard DEC UNIBUS
    or QBUS addresses for these devices.

5.) We haven't encountered your problem before. Call:

    The number listed on the front page of this document.

- Notes -

Version I.5   September 1978

THE CP/M IMPLEMENTATION OF UCSD PASCAL

BOOTING PASCAL

To get Pascal running under your version of CP/M, a two-disk
bootstrap is used. First, boot CP/M in the usual manner. On the CP/M
disk distributed with the Pascal system is a file called PASCAL.COM.
PIP this file over to the booted disk, then execute it.

When the program asks for a Pascal disk, put the disk labeled
PASCAL: in drive A and any disk in drive B. The system may not boot if
there is no disk in drive B, or if you have a 1-drive system and your
CP/M drivers wait on a request to drive B. Then hit [return]. In
about 15 seconds the Pascal welcoming message should appear. (Note: we
have discovered that some drives, possibly as a result of being double-
buffered, cannot keep up with a 2 to 1 interleaving and hence are
extremely slow. The bootstrap then may take about 30 or 40 seconds.
We intend to alleviate this problem in the next release, but persons
with such drives will have to bear with slow disk accesses for the
present.)

If all has gone well, Welcome to the Wonderful World of Pascal.
If not, please call to notify us of your problem.

MODIFICATIONS TO CP/M

The Pascal system will operate under an unmodified CP/M system,
but it is advisable to create a special CP/M for use with Pascal in
order to have Pascal running in the environment for which it was
designed.

1. If there is no disk in a drive and an access is made from
that disk, the driver should not wait to perform that access until a
disk is inserted, as the Pascal system often attempts to read from
empty drives when searching for a particular disk. Instead, simply
return a 1 to indicate a bad I/O operation.

2. If you have a keyboard interrupt handler, it should
recognize the character [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, and jump to that location.

You then need to edit PINIT.ASM, changing MSIZE to match your system. Assemble the file, creating PINIT.HEX.

The next step is to stitch together the one-sector boot, the Pascal interpreter loader, BIOS, and the program to write this information out to sector 0. The following is a session with DDT that performs all this. This session was used to create a 48K system. User input is in lowercase, and comments are off to the right.

```
A>ddt pgen.com          ; load PGEN.COM into memory.  PBOOT, PINIT,
                        ;   and BIOS will be overlayed into PGEN's
                        ;   data area, after which a memory image will
                        ;   be saved.
DDT VERS 1.3
NEXT   PC
0400   0100

-ipboot48.hex          ; set PBOOT48.HEX as input file
```

```
-h900 0                      ; PBOOT starts at location 0, and we want to
                             ;   read it in at location 900H
0900  0900
-r900                        ; read in PBOOT
NEXT  PC
0980  0000

-ipinit48.hex               ; set 'PINIT48.HEX' as input file
-h980 BA00                   ; PINIT starts at location BA00H in a 48K system
                             ;   (in general (MSIZE-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>
```

```
*************************************** **************
* DIFFERENCES AMONG IMPLEMENTATIONS * * Section A.3 *
*************************************** **************
```

Version II.0 March 1979

The following is a list of differences between PDP11 Pascal and
        8080/Z80 Pascal, the items describe the way it is on the
        8080/Z80, and how that differs from the documented system.

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 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 you have a keyboard interrupt
        handler, and this handler is modified as specified below in
        Modifications to CPM.

    This also means that UNITBUSY, UNITCLEAR, and UNITWAIT are
        meaningless. (In the future it may be possible to use the
        UNITBUSY and UNITCLEAR operations on the keyboard, but this is
        currently infeasible.)

3. The interpreter is called SYSTEM.MICRO instead of SYSTEM.PDP-11.

4. The CP/M implementations have bootstraps that are not accessible to
        Pascal, hence the program BOOTER will not work. See the
        appropriate section of this document for instructions on
        copying and/or creating a bootstrap.

5. There are no long integer functions available with the Z80/8080
        system. They will be available in a later release.

```
******************************** ***************
* CHANGES MADE IN RECENT RELEASES * * Section A.4 *
******************************** ***************
```

Version II.0   March 1979

SUMMARY OF DIFFERENCES BETWEEN UCSD PASCAL RELEASES I.5 AND II.0

The following additions, improvements and/or corrections apply
to Version II.0. Reference the (section #) preceding each entry for a
more detailed description. For information regarding differences be-
tween previous releases refer to the system documentation for those
releases.

(1.1)
OPERATING SYSTEM

> (1.1)   C(ompile will now prompt the user for the file to
>   compile, as well as the output file, if the workfile
>   is empty.

(1.2) FILE HANDLER

Substantial modifications have been made in the syntax of user
responses to filer prompts. The symbol "$" means 'same name'. This
symbol may be used on the right-hand side of a transfer command
expression. If the filer detects as some time that two volumes on
line have the same name, the warning message:
Warning: units N & M have the same name'
will appear beneath the prompt line. This message will remain until
some action is taken to convince the filer that this is no longer true.

> R(emove command prompts for verification always.
>
> K(runch command allows space to be opened anywhere on
> the disk.
>
> The bad block scan allows for scanning of any number of
> blocks.

EDITORS   (Sections 1.3 and 1.4)

Three different editors are currently provided with the UCSD
PASCAL system: YALOE, "EDITOR"(E.6), and the new L.2 EDITOR. EDITOR is a
substantially more powerful (and even easier to use) editor than YALOE,
but it makes some assumptions about the run-time environment.
The L.2 EDITOR (eventually to become the standard release editor) will
handle files of arbitrary size, however it is in its experimental form
and recommended for brave users only.

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

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

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

LF   - down one line (and if at the bottom of the screen
                scrolls up)

RLF   - reverse line feed (up one line; not required to
                reverse scroll)

Typing "E" at the main command level will execute the file
SYSTEM.EDITOR. Selection of either YALOE or EDITOR(E.6 or L.2) as
the system editor is made in the Filer by C(hanging the selected file's
name to SYSTEM.EDITOR.

Proper use of EDITOR requires that the system disk be left
on-line while editing.

When prompted with the no work file prompt, typing <escape +
return> will return you to the system command level.

The main purpose of release II.0 is to establish compatibility at the P-code level among all the interpreters maintained by the Pascal Project. This requires changing the P-machine supported on the PDP-11 and Z80/8080 processors, thereby invalidating all I.5 or earlier codefiles. No functional enhancements in the system software have been planned for II.0, although a number of evolutionary improvements have been made. Only two changes have been made which may affect I.5 level source programs:

1) the volume 'REMOTE:' has been split into two volumes -- 'REMIN:' and 'REMOUT:'. Programs using 'REMOTE:' will have to be modified to reflect this change.

2) User programs (rather sophisticated ones presumably) which called the system procedure FBLOCKIO must be changed to accomodate an additional parameter. Uses of BLOCKREAD and BLOCKWRITE are unaffected.

Evolutionary enhancements provided in II.0 are intended largely to improve the ability of the system to operate in in small memory (48K), and small disk (160 block mini-floppy) environments. These improvements include:

1) The Pascal compiler will run in swapping mode automatically (without the (*$S+*) directive) if it determines that no useful program could be compiled without swapping.

2) Codefiles for system programs are no longer required to reside on the system disk. The operating system will, at initialize time, (looking first on '*' volume) scan on-line volumes for the files: SYSTEM.EDITOR, SYSTEM.FILER, SYSTEM.COMPILER, SYSTEM.LINKER, and SYSTEM.ASSMBLER, and remember where they were found. Furthermore, if one of these files cannot be found when it is actually invoked by the user, the system will look again at that time.

The issue of byte-sex - (high order byte numbered 0 or 1?) has also been addressed. If the programmer wishes to have the compiler generate code for a machine of the opposite sex to the one he is running on, the pseudo-comment (*$F+*) (flip) will cause the codefile to be generated for a machine of the opposite sex. (See section 3.6)

The rest of this document concerns only those users who find themselves concerned with the P-Machine internals. A few instructions have been removed, a few have been replaced. The problems solved are those concerning word addressed machines, specifically: Addressing 128K bytes, word boundary troubles with strings and packed arrays, and byte-sex difficulties.

Byte addresses, which are specified by a 16 bit quantity in I.5 systems, are now specified with an address couple. A word base and a byte offset, each of which are 16 bits.

OPCODE Changes:

Decimal value	I.5	II.0	
157	S2P	unused	
16C	LCA	LSA	Load String Address
167	LDO	unused	
169	MVB	LDO	Load Global
208	S1P	LPA	Load Packed array Address
209	IXB	unused	
210	BYT	unused	

Operators with new or changed functional behavior:

1) LSA - puts address of length byte of string constant, which compiler has aligned to a word, on top-of-stack.

2) LPA - puts address of first data byte of string constant, which compiler has aligned to a word, on top-of-stack.

3) INC now adds its parameter (# of words) to the top of stack, and is now used only for addresses.

4) Other operators/standard procedures affected: LDB, STB, MVR, MVL, SCN, FLC, UNITREAD, UNITWRITE, BLOCKIO. These now use address couples, wherever one word byte addresses were used in I.5.

.

Version II.0 February 1979

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

The system is designed to be used primarily with a CRT terminal acting as the CONSOLE device; however, the system is flexible enough to be reconfigured for slower hard-copy terminals. The system does require some kind of fast mass storage such as a floppy disk system or better. For further information regarding compatability between various types of equipment and this system see the "SETUP" document in Section 4.3. This document is intended for programmers who are familiar with the Pascal programming language and have some experience in writing computer programs. Some additional reading suggestions follow: .

The following is a tutorial book on PASCAL:

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

We suggest the following book as a PASCAL reference guide:

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

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

1.1.1 THE UCSD PASCAL SYSTEM: AN OVERVIEW

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

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(omp, L(ink, X(ecute, A(ssem, D(ebug, ? [II.0]

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

Sometimes there are more commands than will fit on the screen. If this is true, a question mark (?) will appear at the end of the line, typing "?" will cause a different prompt to appear, such that more of the available commands will be displayed to the user.

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

1.1.2 OUTERMOST LEVEL COMMANDS: AN OVERVIEW

A. E(dit

Typing "E" while at the outermost command level of the system causes the editor program to be brought into memory from disk. The user may, while in the editor, alter text inside his workfile or any

Page 2

textfile. See Section 1.3 for details. The workfile text (if present) is read into the editor buffer, otherwise the Editor prompts for a file.

B. F(iler

"F" places the user in a level of the system called the Filer. This section of the system contains commands used primarily for maintenance of the disk directory. For more documentation on the Filer level including commands associated with the "getting", "saving", and "clearing" of the user's workfile see Section 1.2.

C. C(omp

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

D. R(un

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

E. X(ecute

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

F. A(ssem

Just like C(omp except the system assembler is invoked rather than the system compiler. See Section 1.9 for more information on the system assembler.

G. D(ebug

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

H. L(ink

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

1.1.3 UTILITY PROGRAMS

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

1.1.4 AN INTRODUCTION TO THE UCSD PASCAL SYSTEM

I.5 is the first release which contains the fully intergrated and implemented concept of separate compilation and assembly. I.4b was the first to support multiple types of processors. I.3 was the first releasable system.

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

Pascal software will also run on that new real machine.

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

Another class of differences stems from variations in the system I/O environments rather than in the host processor. Included here are difference in system console terminal types (i.e. hard-copy vs CRT vs storage tube) or command conventions and capabilities (eg. "intelligent" vs "dumb" CRT's). The system is intended to be able to cope with this sort of variation.

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

The situation in the 8080/Z80 world is much more chaotic. Since is would not be practical for the Project to write and support drivers for the vast multitude of 8080/Z80 I/O environments that exist, we have chosen to take advantage of the widespread implementation of Digital Research's CP/M operating system by structuring the pseudo-machine's I/O operations as calls on CP/M's Basic I/O Subsystem (BIOS) primitives. Therefore, any I/O configuration on which CP/M has been implemented should also be able to support the Pascal system. We do not guarantee this. For example, Intel MDS disk controllers cannot read disks generated here and some BIOS's we have encountered do not completely meet all the requirements specified for CP/M. UCSD plans to support some of the larger distribution 8080-based machines directly.

Our dominant mode of distribution is on 3740 compatible diskettes. One of the distribution diskettes for Z80/8080 systems will be CP/M oriented. This disk will be used, via a somewhat awkward two- step process, to bring up UCSD Pascal on a particular CP/M configuration. Look to section A for details on this process. It also describes the configuration of a modified BIOS, which will better support the needs of the Pascal system. Finally, directions are given for making it possible to boot directly to Pascal rather than indirectly through a CP/M program.

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

```
SYSTEM.PDP-11
SYSTEM.MICRO
SYSTEM.PASCAL
SYSTEM.SYNTAX
SYSTEM.ASSMBLER
```

```
SYSTEM.COMPILER
SYSTEM.EDITOR
SYSTEM.FILER
SYSTEM.LINKER
SYSTEM.STARTUP
SYSTEM.SWAPDISK
SYSTEM.CHARSET
SYSTEM.LIBRARY
SYSTEM.WRK.TEXT
SYSTEM.WRK.CODE
SYSTEM.WRK.INFO
SYSTEM.LST.TEXT
```

In most cases these files contain the system segment of the name they carry. That is to say that the EDITOR, FILER, LINKER, COMPILER, ASSEMBLER are the files that are invoked by the main level of the system when 'E', 'F', etc. is typed. Some of the files are machine specific. PDP-11 and MICRO are the files which contain the interpreters for the particular machine being used. CHARSET is a file which appears on disks meant for TERAK computers only and contains the definition for the soft character set, and the data for the Triton logo prompt. LIBRARY is a file containing separately assembled or compiled routines for use by the Linker in producing executable code files. PASCAL contains the operating system, and the Debugger. SWAPDISK is a file used by some of the system segments during open/close operations on files if a memory shortage exists. It is a 2048 byte file which gets a portion of memory swapped to it when a directory needs to be read into core. When the directory work is complete, the memory is restored to its original state. STARTUP is a file which can be created at the user's option. If it exists on a disk, the operating system considers it a runnable code-file, and executes it at initialize time. This allows the user to have a program that runs before the main command prompt comes up, and will run anytime the I(nitialize command is typed. WRK.TEXT and WRK.CODE are the current work-file after some action has occurred to the work-file. They appear after having done some text editing on a work-file (SYSTEM.WRK.TEXT) or compiling a work- file (SYSTEM.WRK.CODE). To change the editor which is invoked by "E", one simply names the codefile which is to respond to the 'E(dit' command SYSTEM.EDITOR. This is true for all system segments which have named files associated with their command.

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

1.2.1   FILES

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

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

Reserved type suffixes for filenames are:

.TEXT
.BACK   Human readable text.

.CODE   Machine executable code.

.DATA   Data.

.FOTO   A file containing one graphic screen-image.

.GRAF   Intended to be a file containing a
               compressed graphic image. Currently unused.

.BAD   An unmovable file covering a physically
               damaged area of a disk.

.INFO   Debugger information.

1.2.2   VOLUMES

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

Unit Number	Volume ID	Description
1	CONSOLE:	screen and keyboard with echo
2	SYSTERM:	screen and keyboard without echo
3	GRAPHIC:	the graphic 'side' of the screen
4	\<volume name\>:	the system disk
5	\<volume name\>:	the alternate disk
6	PRINTER:	the line printer
7	REMIN:	serial line input
8	REMOUT:	serial line output
9-12	\<volume name\>:	additional disk drives

FIGURE 1

1.2.3 THE 'WORKFILE'

The workfile is a scratch-pad copy of the file being worked
with. It is used by the Filer, in the Editor, and by the Compiler.
When the text part of a workfile is changed, the system stores it on
disk under the name '*SYSTEM.WRK.TEXT', and when a code version is
first created, it is named '*SYSTEM.WRK.CODE'. There may at times
exist other portions of the work-file, with appropriate names.

1.2.4 FILE SPECIFICATION

Many Filer commands require the user to respond with at least
one file specification. The diagram below illustrates the syntax of
file specification.
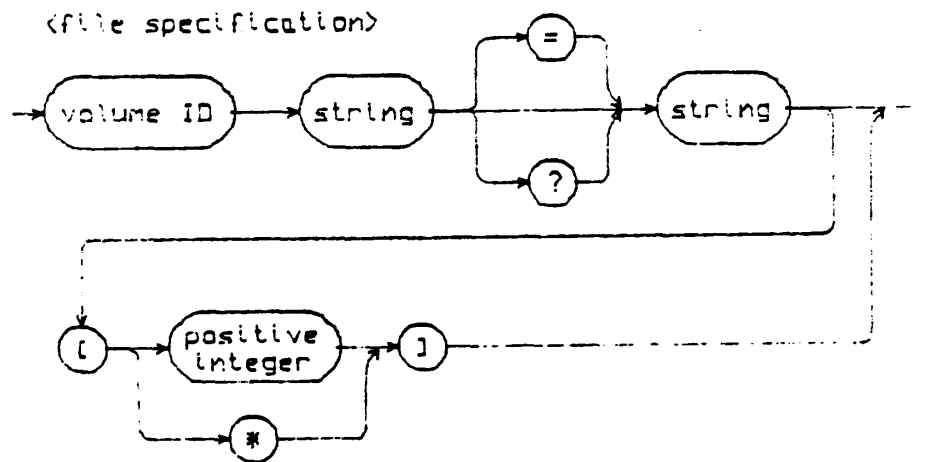
FIGURE 2

Volume i.d. syntax can be expanded thusly:

<volume ID>
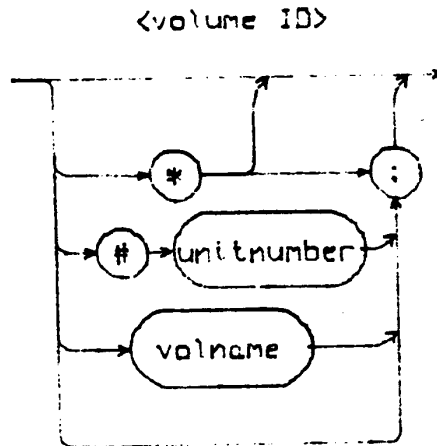
FIGURE 3

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

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

The II.0 Filer will not be able to access filenames containing
the characters '$', ':', '=', '?', and ','. If filenames contain
these characters, then they should be changed before attempting to use
those files with the II.0 System.

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

Given an example directory for volume MYDISK:

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

EXAMPLE:

Prompt: Remove what file?

Response: Typing 'N=' generates the message:

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

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

Typing 'N?' generates the message:

Remove NAUGHTYBITS: ?

After the user types a response, the Filer asks:

Remove NEVERMORE.TEXT: ?

EXAMPLE:

Prompt: Dir listing of what vol ?

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

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

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

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

1.2.5   COMMANDS AND USE

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

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

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

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

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

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

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

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

1.2.5.1 G(et

Loads the designated file into the workfile.

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

Given the example directory:

        FILERDOC2.TEXT
        ABSURD.CODE
        HYTYPER.CODE
        STASIS.TEXT
        LETTER1.TEXT
        FILER.DOC.TEXT
        STASIS.CODE

EXAMPLE:

Prompt: Get what file?

Response: STASIS

The Filer responds with the message

        'Text & Code file loaded'

since both text and code file exist. Had the user typed 'STASIS.TEXT' or 'STASIS.CODE', the result would have been the same - both text and code versions would have been loaded. In the event that only one of the versions exists, as in the case of A.OUT, then that version would be loaded, regardless of whether text or code was requested. Typing 'ABSURD.TEXT' in response to the prompt would generate the message: 'Code file loaded'. Working with the file may cause the files SYSTEM.WRK.xxxx to be created, as part of the workfile. These files will go away when the S(ave command is used. If the system is rebooted before the S(ave command is used, the name of the workfile will be forgotten.

1.2.5.2 S(ave

Saves the workfile under the filename specified by the user.

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

EXAMPLE:

Prompt:

Save as what file?

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

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

RED:EYE

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

MYDISK:SYSTEM.WRK.TEXT   —> RED:EYE.TEXT

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

1.2.5.3 N(ew

Clears the workfile. Creating a blank, unnamed workfile. It
will remain unnamed until it is saved.

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

Prompt:

Throw away current workfile?

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

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

Prompt:

Remove <workfile name>.BACK ?

1.2.5.4 Q(uit

Returns the user to the outermost command level.

1.2.5.5 W(hat

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

1.2.5.6 V(olumes

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

A typical display might be:

```
Volumes on-line:
   1   CONSOLE:
   2   SYSTERM:
   4 # MYDISK:
   6   PRINTER:
   8   REMOTE:
   9 # BIG:
Root vol is - MYDISK:
Prefix is   - MYDISK:
```

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

1.2.5.7 L(dir

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

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

Source file specification consists of a mandatory volume ID, and optional subset-specifying strings, which may be empty. Source file specifications are separated from destination file specifications by a comma (',').

Destination file specification consists of a volume ID, and, if the volume is a block-structured device, a filename.

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

Dir listing of what vol?

```
MYDISK:
FILERDOC2.TEXT    28   1-Sep-78
ABSURD.CODE       18   1-Sep-78
HYTYPER.CODE      12   1-Sep-78
STASIS.TEXT        8   1-Sep-78
LETTER1.TEXT      18   1-Sep-78
ASSEMDOC.TEXT     20   1-Sep-78
FILERDOC1.TEXT    24   1-Sep-78
STASIS.CODE        6   1-Sep-78
10/10 files <listed/in-dir>, 144 blocks used, 350 unused, 200 in largest
```

(The bottom line of the display informs the user that 10 files out of 10 files on the disk have been listed, that 130 disk blocks have been used, that 364 disk blocks remain unused, and that the largest area available is 200 blocks.)

EXAMPLE:

L(dir transaction involving wildcards:

Prompt: Dir listing of what vol ?

User response: #4:FIL=TEXT

generates the following display:

```
MYDISK:
FILERDOC2.TEXT    28   1-Sep-78
FILERDOC1.TEXT    24   1-Sep-78
2/10 files <listed/in-dir>, 62 blocks used, 432 unused, 200 in largest
```

EXAMPLE:

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

Prompt: Dir listing of what vol ?

User response: *FIL=TEXT,PRINTER: causes

```
MYDISK:
FILERDOC2.TEXT    28    1-Sep-78
FILERDOC1.TEXT    24    1-Sep-78
2/10 files <listed/in-dir>, 62 blocks used, 364 unused, 200 in largest
```

to be written to the Printer.

EXAMPLE:

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

Prompt: Dir listing of what vol ?

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

```
MYDISK:
FILERDOC2.TEXT    28    1-Sep-78
FILERDOC1.TEXT    24    1-Sep-78
2/10 files <listed/in-dir>, 62 blocks used, 364 unused, 200 in largest
```

1.2.5.8   E(xtended list

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

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

```
MYDISK:
FILERDOC2.TEXT    28    1-Sep-78     6      512     Textfile
ABSURD.CODE       18    1-Sep-78    34      512     Codefile
<UNUSED>          10                52
ABSURD             4    1-Sep-78    62      512     Datafile
HYTYPER.CODE      12    1-Sep-78    66      512     Codefile
STASIS.TEXT        8    1-Sep-78    78      512     Textfile
LETTER1.TEXT      18    1-Sep-78    86      512     Textfile
ASSEMDOC.TEXT     20    1-Sep-78   104      512     Textfile
FILERDOC1.TEXT    24    1-Sep-78   124      512     Textfile
```

```
<UNUSED>          200              148
STASIS.CODE         6   1-Sep-78   348       512      Codefile
<UNUSED>          154              354
10/10 files <listed/in-dir>, 138 blocks used, 364 unused, 200 in largest
```

1.2.5.9 C(hange

Changes file or volume name.

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

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

Prompt : Change what file?

User Response: #5:F5.TEXT,HOOHAH

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

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

```
POEMS.TEXT
MAUNDER.TEXT
MALPRACTICE
MAKELISTS.TEXT
```

EXAMPLE:

Prompt :   Change what file?

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

```
NOTSANE:MAUNDER.TEXT        —> XXUNDER.GAACK
NOTSANE:MAKELISTS.TEXT      —> XXKELISTS.GAACK
```

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

```
THIS.TEXT
THAT.TEXT
```

EXAMPLE:

Prompt :   Change what file?

User Response: T=T,=

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

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

EXAMPLE:

Prompt :   Change what file?

User Response: NOTSANE:,WRKDISK:

```
NOTSANE:                      --> WRKDISK
```

1.2.5.10 R(emove

Removes file entries from the directory.

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

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

EXAMPLE:

    Prompt: Remove what file?

    User Response: AMAZING.CODE

    removes the file AMAZING.CODE from the volume directory.

    Note: To remove SYSTEM.WRK.TEXT and/or SYSTEM.WRK.CODE the N(ew command should be used, or the system may get confused. Fortunately, before finalizing any wildcard removes, the Filer prompts the user with

    Prompt: Update directory?

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

    As noted before, wildcard removes are legal.

EXAMPLE:

    Prompt: Remove what file?

    User Response: A=CODE

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

1.2.5.11 T(ransfer

Copies the specified file to the given destination.

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

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

EXAMPLE:

Prompt: Transfer what file ?

User Response: MYDISK:FARKLE.TEXT

Prompt: To where?

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

User Response: BACKUP:NAME.TEXT

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

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

MYDISK:FARKLE.TEXT   —> BACKUP:NAME.TEXT

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

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

MYDISK:FARKLE.TEXT,BACKUP:$

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

MYDISK:FARKLE.TEXT,BACKUP:

generates the message :

Destroy BACKUP: ?

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

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

EXAMPLE:

Prompt: Transfer what file?

User Response: FARKLE.TEXT

Prompt: To where?

User Response: PRINTER:

causes FARKLE.TEXT to be written to the printer.

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

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

EXAMPLE:

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

Prompt: Transfer what file?

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

would cause the Filer to reply:

MYDISK:PAUCITY   --> ODDNAMZ:VAUCIS
MYDISK:PARITY   --> ODDNAMZ:VARIS
MYDISK:PENALTY   --> ODDNAMZ:VENALS

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

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

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

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

EXAMPLE:

Prompt: Transfer what file?

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

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

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

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

EXAMPLE:

Prompt: Transfer what file?

User Response: MYDISK:,EXTRA:

Prompt: Destroy EXTRA: ?

WARNING: If the user types 'Y', the directory of EXTRA: will be destroyed! An 'N' response will return the user to the outer level of the Filer, and a 'Y' will cause EXTRA to become an exact copy of MYDISK. Often this is desirable for backup purposes, since it is relatively easy to copy a disk this way, and the volume name can be changed (see C(hng) if desired.

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

1.2.5.12 D(ate

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

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

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

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

1.2.5.13 P(refix

        Changes the current default to the volume specified.

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

        To determine the current default volume, the user may respond to the prompt with ':'. To return the prefix to the booted or "Root" volume, user may respond with "*".

1.2.5.14 B(ad blocks

        Scans the disk and detects bad blocks.

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

Prompt: Bad block scan of what vol?

Response: <volume ID>

Prompt: Scan for 494 blocks ? <y/n>

Response may be "Y" for yes if you want to scan for the entire length of the disk. If you only wish to check a smaller portion of the disk, type "N" and you will then be prompted for the number of blocks you want the filer to scan for. The purpose of this part of the command is for disks where the filer has no idea of how 'long' the device is.

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

1.2.5.15   eX(amine

Attempts to physically recover suspected bad blocks.

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

EXAMPLE:

Prompt : Examine blocks on what volume?

Response : <volume ID> generates the

Prompt: Block-range ?

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

Prompt: File(s) endangered:
          <filename>
          Fix them?

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

Block <block-number> may be ok

in which case the bad block has probably been fixed, or

Block <block-number> is bad

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

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

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

1.2.5.16 K(runch

Moves the files on the specified volume so that unused blocks are combined.

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

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

EXAMPLE:

Prompt : Crunch what vol?

Response : <volume ID>

causes Filer to prompt with:

Prompt : From end of disk, block 493 ? (y/n)

Response: 'Y' initiates the K(runch. Typing an 'N' will cause
the prompt:

Prompt : Starting at block # ?

Response: The block number at which you wish the filer to
open a space on the disk.

1.2.5.17 M(ake

Creates a directory entry with the specified filename.

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

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

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

EXAMPLE:

Prompt : Make what file?

Response : MYDISK:FARKLE.TEXT[28]

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

1.2.5.18 Z(ero

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

EXAMPLE:

    Prompt: Zero dir of what vol ?

    Response: <volume ID>

    Prompt: Destroy <volume name> ?

    Response: A 'Y' response generates

    Prompt: Duplicate dir ?

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

    Prompt: Are there 494 blks on the disk ? (y/n)

    Response: 'N' generates

    Prompt: # of blocks on the disk ?

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

    'Y' generates

    Prompt: New vol name ?

    Response: User types any valid volume name.

    Prompt: <new volume name> correct ?

    Response: 'Y' causes the Filer, if it could indeed write the
new directory on the disk, to respond with the message:

    <new volume name> zeroed

MACHINE	DISK TYPE	# OF BLOCKS
Terak	Single-density soft-sectored 8" floppy	493
Northwest Micro	Double-density soft-sectored 8" floppy	1101
Zilog	Single-density hard-sectored 8" floppy	607
North Star	Single-density hard-sectored 5 1/4" floppy	167
DEC	RK05 / per volume	4871

These are the numbers that one types when the filer asks for a number of blocks, as the blocks are numbered from zero.   ed.

```
************* *** ************* ** *************
* SCREEN ORIENTED EDITOR * * Section 1.3.1 *
************************** *****************
```

Version II.0 February 1979

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

THE CONCEPT OF A 'WINDOW' INTO THE FILE

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

THE CURSOR

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

THE CONCEPT OF A PROMPT LINE

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

>Edit: A(djust C(py D(lete F(ind I(nsrt J(mp Rplace Q(uit X(chng Z(ap [E.6]

Page 31

NOTATION

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

ENVIRONMENT

        In order to establish the correct environment, depending on
whether text or a program is to be edited, see the options available
under Environment in the Miscellaneous commands section.

ENTERING THE WORKFILE AND GETTING A PROGRAM.

On entering the Editor :

No workfile is present. File? (<ret> for no file <esc-ret> to exit)
:

appears.

There are three ways to answer this question :

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

Figure 2.1
```
---------------------------------------------------------------
PROGRAM STRING1;
BEGIN
  WRITE('TOO WISE');
  WRITE('YOU ARE');
  WRITELN(',');
  WRITELN('TOO WISE');
  WRITELN('YOU BE')
END.
---------------------------------------------------------------
```

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

3) With <escape + return>. This causes the editor to drop you
back to the system command level. Useful when you didn't mean to type
'E'.

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

In order to edit, it is necessary to move the cursor. On the keyboard are four keys with arrows,(which may look like triangles), which move the cursor. The <up-arrow> moves the cursor up one line, the <right-arrow> moves the cursor right one space and so forth. On terminals which do not have cursor keys, the system will have to be set up with a set of control keys to act as vector keys. Refer to section 4.3 for more information on setting control keys.

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

Figure 2.2

BEGIN
  WRITE('TOO WISE ');

BEGIN
  wRITE('TOO WISE ');

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

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

Figure 2.3

----

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

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

The Edit level prompt line shows that to I(nsrt (insert) an item, type "I". The cursor must be in the correct position before typing "I". Earlier, the cursor was moved to the "W" in "TOO WISE"; now, on typing "I", an insertion will be made before the "W". The rest of the line from the point of insertion will be moved to the right hand side of the screen. In the event that the insertion is lengthy, that part of the line will be moved down to allow room on the screen. After typing "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 the screen it is NOT insert mode and a wrong key may have been typed.

If the cursor is at the "W" in "WISE", and on typing "I" the insert prompt line appeared, "SMART" may be inserted by typing those five letters. They will appear on the screen as they are typed.

There remains one more important step. The choice at the end of the prompt line indicates that pushing the <etx> key accepts the insertion, while pushing the <esc> key rejects the insertion and the text remains as it was before typing "I".

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 and causes the Editor to start a new line. Notice where carriage return places the cursor. This is intended as a programming aid.

USING DELETE

The DELETE mode works like the INSERT mode. Having inserted the 'SMART' into the STRING1 program and having pushed <etx>, 'WISE' must be deleted. Move the cursor to the first of the items to delete and type "D" to put the Editor into DELETE mode. The following prompt line should appear:

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

Each time <space> is typed a letter disappears. In this example typing 4 spaces will cause "WISE" to disappear. Now the same choice must be made as in insert. Type <etx> and the proposed deletion is made or type <esc> and the proposed deletion reappears and remains part of the text.

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

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

LEAVING THE EDITOR AND UPDATING THE WORKFILE

When all the changes and additions have been made, exit the Editor and "save" a copy of the modified program. This is done by typing "Q" which will cause the prompting display shown in Fig. 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 the modified file on disk is to type "U" for U(pdate which causes the workfile to be saved as SYSTEM.WRK.TEXT. With the workfile thus saved, it is possible to use the R(un command, provided of course the file is a program. It is also possible to use the S(ave option in the Filer to save the modified file before using the Editor to modify or create another file.

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

COMMAND AND MODE

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

REPEAT-FACTORS

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

THE CURSOR

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

DIRECTION

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

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

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

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

JUMP

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

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

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

Jump to what marker?

Miscellaneous commands.

PAGE

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

EQUALS

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

TEXT CHANGING COMMANDS

INSERT

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

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

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

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

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
first position in the next line. Note: if Filling is True, the first
position is the Left-margin. Unless the lne above is blank, in which
case the first position is that of Paragraph margin.

FILLING

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

space and a hyphen to be a word.

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

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

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

Figure 3.1
```
----------------------------------------------------------------
ONE           Original indentation
  TWO         Indentation changed by <space> <space>
    THREE     <return> causes auto-indentation to level of line above
  FOUR        <backspace> changes indentation from level of line above
----------------------------------------------------------------
```

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

"ONCE UPON A TIME THERE- WERE".

(Very narrow margins have been used for simplicity.)

Figure 3.2
```
----------------------------------------------------------------
ONCE UPON A     Auto-returned when next word would exceed margin
TIME THERE-     Auto-returned at hyphen
WERE
          ^
    Level of left margin
----------------------------------------------------------------
```

The cursor may be forced to the left margin of the screen by typing the ASCII control code DC1. (Generated by <CTRL-Q>)

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

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

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

DELETE

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

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

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

Example:

In Figure 3.3:
  1) Move the cursor to the "E" in END.
  2) Type"<" (This changes the direction to backward)
  3) Type "D" to enter DELETE mode.
  4) Type <ret> <ret>. After the first return the cursor moves to before the "W" in WRITELN and "WRITELN('TO BE.');"disappears. After the second return the cursor is before the "W" in WRITE and that line has disappeared.
  5) Now press <etx>. The program after deletion appears as is shown in Figure 3.4.

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

Figure 3.3

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

Figure 3.4

```
PROGRAM STRING2:
BEGIN

END.
```

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

ZAP

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

The position of the cursor where what was previously found, replaced, or inserted is called the "equals mark". Typing the "=" key will place the cursor exactly there.

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

Whatever was deleted by using the ZAP command is available for use with the COPY mode, unless the editor has stated otherwise.

COPY

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

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

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

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

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

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

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

On the completion of the copy command in either mode the cursor returns to immediately before the text which was copied.

The copy buffer is affected by the following commands:

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

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

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

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

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

EXCHANGE

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

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

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

Figure 3.5   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 the changes part of the file.

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

FIND AND REPLACE

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

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

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

The Editor considers any character which is not a letter or a number to be a delimiter.

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

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

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

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

NOTE: The S(et-E(nvironment mode displays the current target and substitution strings.

FIND

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

Figure 3.7
```
--------------------------------
>Find[1]: L(it <target>  =>

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

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

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

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

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

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

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

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

Figure 3.8

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

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

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

and the cursor will appear at the second WRITE.

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

Figure 3.9

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

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

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

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

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

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

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

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

Figure 3.10
```
--------------------------------------------------------------
    WRITE(',');                         WRITELN;
    WRITE( ',');                        WRITELN;
--------------------------------------------------------------
```

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

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

Typing an "R" at this point will cause a replacement while
typing a space will cause the REPLACE mode to search for the next
occurrence provided the <repeat-factor> has not been reached. The
<repeat-factor> counts the number of times an occurrence is found, not
the number of times you actually type "R". Use "/" as a <repeat-
factor> in order to examine every occurrence of the target string.
If the Editor can not find the target string the number of times
specified, the prompt:

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

appears.

FORMATTING COMMANDS

ADJUST

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

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

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

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

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

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

MARGIN

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

There are two 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 occurring between two
blank lines beginning or end of file, or a line which starts with the
command charager. To MARGIN a paragraph move the cursor to anywhere
in that paragraph and type "M". When doing an exceptionally long
paragraph it may take several seconds before the routine is ready to
redisplay the screen. Margin works with blanks and hyphens to do its
splitting. All other characters in sequence are considered words.
It does not know how to hypenate words itself.

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 Edit level.
The prompt for the SET command does not appear on the ">Edit:" prompt
line due to space limitations. On entering the SET mode the following
prompt line appears:

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

M(arker:

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

Name of marker?

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

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

Figure 3.15

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

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

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

E(nvironment:

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

Figure 3.16

`Environment: {options} <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'

    Date Created: 4-13-55   Last Used: 12-28-78

        By typing the appropriate letter, any or all of the options
may be changed. The options shown are the default options for the
Editor on most screens. 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 is set to True (turned on) by typing "AT" and to False (turned
off) by typing "AF".

F(illing:

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

L(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 and a <space>.
The positive integer typed replaces the old value for the L(eft margin
in the prompt shown in Figure 3.16. All positive integers with less
than four digits are valid margin values.

C(ommand ch:

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

T(oken def:

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

VERIFY

The VERIFY command is executed by typing "V" while at the Edit level. The status of the Editor is verified by redisplaying the screen. 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 Edit level. On entering QUIT mode the screen display is replaced by the following prompt:

Figure 3.17

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

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

U(pdate:

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

E(xit:

        This causes the Editor to leave without making any changes in
SYSTEM.WRK.TEXT. This means that any modifications made since entering
the Editor are not recorded in the permanent workfile. All editing
during the session is irrecoverably lost.

R(eturn:

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

W(rite:

        This option puts up a further prompt:

Figure 3.18

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

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

Figure 3.19

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

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

-- Notes --

```
<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
```

<repeat-factor> is any number typed before a command. Typing a / is the
        infinite number.

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 framed in insert/delete/zap 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.

                                                        Page 55

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 typed while in this
        mode. Each line must be done separately. <back-space> causes the
        original character to re-appear.

Z)ap: Treats the starting position of the last thing
        found/replaced/inserted as an anchor and deletes everything
        between the anchor and the current cursor position.

Version II.0 February 1979

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

Entering the Workfile and Getting a Program

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

ERROR: Not enough room for backup!

will be displayed. This disk must then be K(runched in order to
provide room if that is possible, a file removed or another disk must
be used. The backup file is always 'written' to disk with the
original file data in it.

The same prompt line is displayed; see section 1.3.2.

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

Figure 5.1

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

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

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

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

Leaving the Editor and Updating the workfile

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

Figure 5.2

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

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

U(pdate:

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

When a new file is created, the following appears:

Figure 5.3

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

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

Figure 5.4

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

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

E(xit:

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

R(eturn:

This is the same. See section 1.3.3.

MOVING COMMANDS

JUMP

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

Typing "M" causes the Editor to display:

Jump to what marker?

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

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

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

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

NEXT

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

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

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

Figure 5.5   SYMBOLIC FILE

```
---------------------------------------------------------------
      -----------------------------------------------------
      ! left stack    !             ! right stack        !
      ! Backwards      !   BUFFER    ! Forward            !
      ! Start          !             ! End                !
      -----------------------------------------------------


---------------------------------------------------------------
```

PAGE

        See section 1.3.3.

EQUALS

        See section 1.3.3.

TEXT CHANGING COMMANDS

INSERT

        See section 1.3.3.

DELETE

        See section 1.3.3.
ZAP

        See section 1.3.3.

COPY

        See section 1.3.3.

EXCHANGE

        See section 1.3.3.

FIND

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

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

On typing "Y", the Editor will move another section of the file into the buffer to continue searching. Find is still directional. If the pattern is not found, in a full-file search, the cursor is left in an arbitrary position in the file.

REPLACE

See section 1.3.3.

FORMATTING COMMANDS

ADJUST

See section 1.3.3.

MARGIN

See section 1.3.3.

MISCELLANEOUS COMMANDS

SET

See section 1.3.3. The same prompt line is displayed.

M(arker:

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

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

E(nvironment:

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

Figure 5.5

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

11582 bytes used. 2754 available.

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

Markers:

<P1   P2   >P3

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

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

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

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

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

VERIFY

        See section 1.3.3.

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

Version II.0 February 1979

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. Its name is
pronounced: Yaw-loo-ee.

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

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

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

1.4.1   SPECIAL KEY COMMANDS

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

> <esc>   Echoes a '$'. A single <esc> terminates a
>   text string. A double <esc> executes the
>   command string.

RUBOUT <linedel>	Deletes current line. On hard-copy terminals echoes '<ZAP' and a carriage return. On others, it clears the current line on the screen. In both cases the contents of that line are discarded by the editor.
CTRL H <chardel>	Deletes character from the current line. On hard-copy terminals it echoes a percent sign followed by the character deleted. Each succeeding CTRL H the 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 <linedel> 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.

1.4.2   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
useful for a large repeat factor.

= '=' is used only with the J, D and C commands
and represents -n, where n is equal to the length
of the last text argument used, for example
*GTHIS$=D$$ finds and removes THIS.

1.4.3 COMMAND STRINGS

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

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

*B   GTHE INSERTED$ -3CING$ 5K   GSTRING$$

The "B" sets the cursor position.
The "G" looks for the string "THE INSERTED" and places the cursor on
   the character which follows the "D".
The "-3CING" replaces the string "TED" with "ING".
The "5K" deletes text from the cursor to the 5th successive end-of-
   line.
The "GSTRING" finds the first occurance of "STRING" in the file and
   places the cursor just after the G.

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

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

1.4.4   THE TEXT BUFFER

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

The editor can only work on files that fit entirely within the
Text Buffer.

1.4.5   THE CURSOR

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

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

1.4.6   INPUT/OUTPUT COMMANDS

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

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$$	Prints all characters beginning at the cursor and terminating at the 4th <cr>.
*0L$$	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. Arguments are ignored. The V(erify command is equivalent to a 0LL (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. Refer to Figure 2 in section 1.2.4 for details on filename specifications.

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

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

OUTPUT ERROR. HELP!

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

The R(ead command is of the form

*R<file title>$

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

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

The Q(uit command has several forms

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.

1.4.7   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. Use /J to move to the end of the buffer.

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

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

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

1.4.8   TEXT MODIFICATION COMMANDS

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

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

type two successive <esc>s. To continue, type I to return to the Text mode.

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

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

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

     *B$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's position is the beginning of the line following the deleted text.

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

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

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

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

     *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 remains at the end of the changed text.

*-5XTEXT$$	Exchanges all characters beginning with the first character on the 5th line back and ending at the cursor with the string 'TEXT'.
*0XTEXT$$	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 or /DI).

1.4.9   OTHER COMMANDS

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

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

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

The Save Buffer may be cleared 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', 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. It also lists the numbers of the currently defined macros.

1.4.10 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 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.

 O: - invalid -

 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 -

-- Notes --

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

Version II.0   February 1979

        To facilitate the debugging of Pascal programs, an interactive
debugger was included in the system in earlier releases. In order to
use it, it required more memory than was available with any
meaningfully sized program. We removed the debugger from the system
as it was more of a thorn in the side of progress than a statement of
progress itself. We are working on a new debugger and hope to have it
in a useful state soon. The current changes in the P-machine may make
the task of writing the debugger somewhat easier, and therefor
quicker. Please do not inquire as to when the debugger will be ready
for release, as the answer you will get will be "soon".

        Thank-you for your patience and cooperation in this matter.

        Ed.

-- Notes --

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

Version I.5   September 1978

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

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

```
PASCAL compiler [I.5 unit compiler]
<   0>..................
P1 [7050]
<  19>.........................................
P2 [3040]
<  61>.................................................
< 111>.......
TEST [3003]
< 119>..............................................
```

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

If the compilation is successful, that is, no compilation
errors were detected, the compiler writes a codefile to the disk
called *SYSTEM.WRK.CODE. This is the codefile which is executed if
the user types the R(un command. See Section 1.1 INTRODUCTION AND
OVERVIEW for a global description of the system commands.

Should the compiler detect a syntax error, the text surrounding
the error and an error number together with the marker '<<<< ' will
point to the symbol in the source where the error was detected. In
the event that both the Q and L options are set, the compilation will
continue, with the syntax error going to the listing file, and the
console remaining undisturbed. Otherwise the compiler will the give
the user the option of typing a space, an <esc> or 'E'. Typing a .

space instructs the compiler to continue the compilation, while escape causes 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.

The syntax errors detected by the UCSD Pascal compiler are listed in Table 5. All error numbers will be accompanied by a textual message upon entry to the editor if the file *SYSTEM.SYNTAX is available.

1.6.1 COMPILE TIME OPTIONS

Compile time options in the UCSD Pascal compiler are set according to a convention described on pages 100-102 of Jensen and Wirth, where compile time options are set by means of special "dollar sign" comments inside the Pascal program text. The syntax used in UCSD's compiler control comments is essentially as described in Jensen and Wirth. The actual options and the letters associated with those options bear little resemblance to the options listed on pages 101 and 102 of Jensen and Wirth. Following is a description the various options currently available to the user of the UCSD Pascal compiler.

B:

Byte-flip. Causes the compiler to generate code for a machine which is byte-flipped from the one upon which it is running.

C:

Places the line following the C character for character somewhere in the codefile. The purpose of this is to have a copyright notice imbedded in codefiles.

D:

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

Default value: D -

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

D+: causes the compiler to emit breakpoint instructions.

G:

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

Default value: G-

G+: allows the use of the GOTO statement.

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

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

I:

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

IOCHECK OPTION

Default value: I+

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

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

The I-option is useful for programs which do many I/O operations and also check the IORESULT function after each I/O operation. The 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 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 a file name starts with '+' or '-' as the first character of the filename, a blank must be inserted between '(*$I' and 'FILENAME'. For example, the comment:

(*$ITURTLE.TEXT*)

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

(*$I +FARKLE.STUFF*)

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

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

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

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

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

L:

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

(*$L DEMO1.TEXT*)

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

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

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

P:

Page. Pages listing file.

Q:

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

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

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

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

R:

This option affects the value of the boolean variable
RANGECHECK in the compiler. If RANGECHECK is true, the compiler will
output 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 strongly advised to compile with the R+ option left on.

S:

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

Default value: S-

S+: puts compiler in swapping mode.

S-: puts compiler in non-swapping mode.

U:

USER PROGRAM OPTION:

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

Default value: U+

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

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

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

USE LIBRARY OPTION:

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

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

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

-- Notes --

```
*************************  ***************
* UCSD BASIC COMPILER *  * Section 1.7 *
*************************  ***************
```

Version I.5   September 1978

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

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

        The UCSD Basic Compiler has been written in the Pascal
language. Some of the intrinsics of the Pascal language, which are not
found in standard Basic, are found within the UCSD version of Basic.
Many of these are noted in the first section, all of them are noted or
recapped in the second.
¦IX SYSTEM.COMPILER
        The UCSD BASIC Compiler is invoked just like the Pascal
compiler, provided the compiler code is named *SYSTEM.COMPILER.
Originally it will be named BASIC.COMPILER. If you want a disk to be
BASIC oriented, you must change the name of, or remove, the Pascal
compiler, and change the name of BASIC.COMPILER to *SYSTEM.COMPILER.
That disk, and any copies of it, will now compile BASIC programs as a
result of the C(ompile or R(un command.

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 occurrence of the string s1 in the
          string s2.

OTHER FUNCTIONS

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

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

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

OLD(c,s)
NEW(c,s) c is a numeric constant without a fraction part, which becomes

associated with the disk file whose name is in s. OLD expects that
file to already exist, NEW creates a new one with the name s, removing
any previous file of that name. These functions must occur before
associated print or input statements. The numbers may not be
reassigned and must be in the range 1..16. For best results, use only
at the top of a program. In order that a file created by NEW be
editable with either of the system editors, '.text' must be appended to
the file title.

These functions return IORESULT as described in section 2.1.

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 quotation marks prints a single
quotation mark.

FOR var = exp1 TO exp2 STEP exp3

:
NEXT var

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

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

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

ON exp GOTO(ln1,ln2..)

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

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

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

LET var=exp
   or
var=exp

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

DIM var (n1,n2,...)

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

FEATURES TO BE ADDED

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.

RUNNING A BASIC PROGRAM

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

```
************** **************
* THE LINKER * * Section 1.8 *
************** **************
```

Version II.0   February 1979

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

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

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

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

1.8.1   USING THE LINKER

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

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

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

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

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

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

Host file?

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

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

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

For information on LIBRARIES and the LIBRARIAN see Section 4.2.

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

Map file? <file identifier> <ret>

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

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

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

1.8.2   LINKER CONVENTIONS AND IMPLEMENTATION

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

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

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

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

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

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

        LINKED: The codesegment is fully executable. Either all external
                references (UNITs or EXTERNALs) have been resolved, or
                none were present.

        HOSTSEG: the segkind assigned to the outer block of a PASCAL
                 program if the program has external references.

        SEGPROC: the segkind assigned to a PASCAL segment procedure.

        UNITSEG: the segkind assigned to a compiled SEGMENT. (see Section
                 3.3.1)

SEPRTSEG: This segkind is assigned to a separately compiled
procedure or function. Assembly language codefiles are
always of this type, as well as Pascal UNITs which are
not SEGMENT UNITs.

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

```
LITYPES = (EOFMARK, UNITREF, GLOBREF, PUBLREF, PRIVREF, CONSTREF,
           GLOBDEF, PUBLDEF, CONSTDEF, EXTPROC, EXTFUNC, SEPPROC,
           SEPFUNC, SEPPREF, SEPFREF);

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

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

```
********************** ***************
* ADAPTABLE ASSEMBLER * * Section 1.9 *
********************** ***************
```

Version II.0   February 1979

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

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

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

1.9.1   USAGE

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

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

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

        The next prompt line is:

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

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

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

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

       .INCLUDE   <FILE ID>

indicating which file has been included.

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

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

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

```
0000|                              .PROC    PRIMARYZ
Memory after initialization:    6068
0000|
0000|                    FLOPPY  .EQU  0BFDH         ;Rom-based floopy driver.
0000|                    SECMEM  .EQU  9000H         ;First location in memory
0000|                    SECENT  .EQU  9000H         ;Entry point of bootstrap
0000|                    SECDSK  .EQU  08H + 1700H   ;Sector start of 2nd bootstrap
0000|                    B1DSK   .EQU  10H + 1700H   ;Sector start of BIOS part 1
0000|                    B2DSK   .EQU  18H + 1700H   ;Sector start of BIOS part 2
0000|
0000|                            .ORG  1000H         ;Primary boot for ZILOG DOS
1000|
1000| FD 21 ****  PRIMARY LD    IY,SECREAD  ;Get block for second bootstrap
1004| CD FD0B             CALL  FLOPPY
1007| FD 21 ****         LD    IY,B1READ   ;Get block for part 1 of BIOS
100B| CD FD0B             CALL  FLOPPY
100E| FD 21 ****         LD    IY,B2READ   ;Get block for part 2 of BIOS
1012| CD FD0B             CALL  FLOPPY
1015| C3 0090             JP    SECENT      ;Jump into second bootstrap
1018|
1002* 1810
1018|                    SECREAD
1018| 00                  .BYTE $-$         ;Unused
1019| 0A                  .BYTE 0AH         ;Read command
101A| 0090                .WORD SECMEN      ;Memory loc. for second boot
101C| 0002                .WORD 200H        ;Number of bytes in boot
101E| 0000                .WORD $-$         ;Completion return address
1020| 0010                .WORD PRIMARY     ;Error in return address
1022| 00                  .BYTE $-$         ;Completion result code
1023| 0817                .WORD SECDSK      ;Disk block of second boot
1025|
1009* 2510
1025|                    B1READ
1025| 00                  .BYTE $-$         ;Unused
1026| 0A                  .BYTE 0AH         ;Read command
1027| 0093                .WORD SECMEN+300H ;Memory location or BIOS part 1
1029| 0002                .WORD 200H        ;Number of bytes in BIOS part 1
102B| 0000                .WORD $-$         ;Completion return address
102D| 0010                .WORD PRIMARY     ;Error return address
102F| 00                  .BYTE $-$         ;Completion result code
1030| 1017                .WORD B1DSK       ;Disk block of BIOS part 1
1032|
1010* 3210
1032|                    B2READ
1032| 00                  .BYTE $-$         ;Unused
1033| 0A                  .BYTE 0AH         ;Read command
1034| 0095                .WORD SECMEN+500H ;Memory location of BIOS part 2
1036| 0002                .WORD 200H        ;Number of bytes in BIOS part 2
1038| 0000                .WORD $-$         ;Completion return address
103A| 0010                .WORD PRIMARY     ;Error return address
103C| 00                  ..BYTE $-$        ;Completion result code
103D| 1817                .WORD B2DSK       ;Disk block of BIOS part 2
103F|
103F|                            .END
```

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

B1DSK   AB 1710¦ B1READ   LB 1025¦ B2DSK   AB 1718¦ B2READ   LB 1032 ¦
FLOPPY   AB 0BFD¦ PRIMARY LB 1000¦ PRIMARYZ PR ----¦ SECDSK   AB 1708¦
SECENT   AB 9000¦ SECMEM   AB 9000¦ SECREAD LB 1018¦

NOTES:

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

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

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

1.9.2   HIGH-LEVEL SYNTAX

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

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

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

A general railroad diagram for all assembly files looks like:
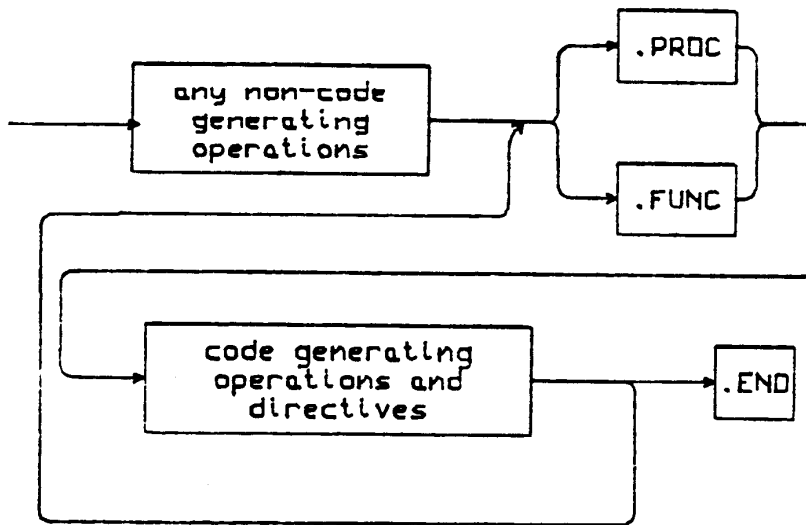
The non-code generating operations are:

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

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

1.9.3 EXPRESSIONS (one-pass restrictions)

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

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

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

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

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

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

For unary operations:
    '+'   plus
    '-'   minus
    '~'   ones complement

For binary operations:
    '+'   plus
    '-'   minus
    '^'   exclusive or
    '*'   multiplication
    '/' truncating division (DIV)
    '%' remainder division (MOD)
    '¦' bit wise OR
    '&' bit wise AND
    '=' equal (valid only in .IF)
    '<>' not equal (valid only in .IF)

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

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

1.9.4 ASSEMBLER DIRECTIVES: OVERVIEW

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

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

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

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

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

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

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

     expression = any legal expression as defined in Section 1.9.3.

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

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

1.9.4.1 DELIMITING DIRECTIVE FOR ROUTINES

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

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

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

the associated declaration for the .PROC would be

.PROC FARKLE,4

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

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

            FORM: .PROC <identifier>[,expression]

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

            EXAMPLE: .PROC   DLDRIVE,2

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

            FORM: .FUNC <identifier>[,expression]

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

            EXAMPLE: .FUNC RANDOM,4

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

1.9.4.2   LABEL DEFINITIONS AND SPACE ALLOCATION DIRECTIVES

.ASCII   Converts character values to ASCII equivalent byte

constants and places the equivalents into the code stream.

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

EXAMPLE: .ASCII   "HELLO"

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

           .ASCII   "AB"
           .BYTE   34   ; 42 octal
           .ASCII   "CD"

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

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

FORM: [label] .BYTE   [valuelist]

           the default for no stated value is 0.

EXAMPLE: TEMP .BYTE   4

           the associated output would be: 04

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

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

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

EXAMPLE: TEMP .BLOCK   4,6

the associated output would be:
```
06
06      ( four bytes with the value 06 )
06
06
```


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

FORM: [label] .WORD   \<valuelist\>

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

the associated output would be:
```
0000
0002
0004    (words with these values in them )
  .
  .
```

EXAMPLE: L1   .WORD L2
```
            .
            .
            .
```
L2   .EQU $ $ represents the LC on the Z80
      .WORD 5.
```
            .
```
if LC was 50 at the .EQU
the associated output would be:
```
0050    (* assignment due to the L2 value *)
  .
  .
0005    (* assignment due to the .WORD 5 *)
  .
```


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

```
FORM:        <label>    .EQU     <value>

EXAMPLE:    BASE       .EQU     R6
```

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

1.9.4.3 MACRO FACILITY DIRECTIVES

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

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

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

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

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

.ENDM   Indicates the end point of a MACRO.

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

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

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

```
          HELP   FIRST,SECOND
   #      STA    FIRST
   #      LDA    SECOND
```

The statement HELP, calls the macro and sends it two
parameters, FIRST and SECOND. These parameters are in turn
referenced inside the macro using the identifiers %1 for the
variable FIRST, and %2 for the variable SECOND.

1.9.4.4   CONDITIONAL ASSEMBLY DIRECTIVES

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

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

.IF   Identifies the beginning of the conditional.

.ENDC   Identifies the end of a conditional .IF

.ELSE   Identifies the alternate to the .IF. If the conditional
         expression is equal to 0 then the else is used.

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

where the expression is the conditional expression to be met.

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

1.9.4.5   PASCAL HOST COMMUNICATION DIRECTIVES

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

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

```
FORM:       .CONST    <idlist>

EXAMPLE:    (* see example after .PRIVATE *)
```

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

```
FORM:       .PUBLIC  <idlist>

EXAMPLE:    (* see example after .PRIVATE *)
```

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

```
FORM:       .PRIVATE  <identifier:integer list>

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

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

Given the following Pascal host program:

```
PROGRAM EXAMPLE;
CONST SETSIZE=50; LENGTH=30;

VAR I,J,F,HOLD,COUNTER,LDC:INTEGER;
    LST1:ARRAY[0..9] OF CHAR;

BEGIN
    ..;
    ...
    ...
END.
```

and the following section of an assembly routine:

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

This will allow the const LENGTH to be used in the assembly
routine almost as if the line LENGTH .EQU 80 had been
written. (Recall the limitation mentioned above for the use
.CONST identifiers.) The variables LDC,I,J to be used by both
the Pascal host and the assembly routine, and the variables
PRT, LST2 to be used only by the assembly routine. Further,

the LST2:9 causes the variable LST2 to correspond with the
beginning of a 9 word block of space in the global data
segment.

1.9.4.6   EXTERNAL REFERENCE DIRECTIVES

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

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

             FORM: .DEF <identifierlist>

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

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

             FORM: .REF <identifierlist>

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

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

1.9.4.7 LISTING CONTROL DIRECTIVES

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

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

```
FORM:        .LIST or .NOLIST
```

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

```
FORM:        .PAGE
```

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

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

```
           INTERP    SYMBOLTABLE DUMP
```

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

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

           EXAMPLE      .TITLE    QRC12 interpreter
```

1.9.4.8   FILE DIRECTIVES

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

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

```
           CORRECT EXAMPLE:     .INCLUDE  SHORTSTART.TEXT

           CORRECT EXAMPLE:     .INCLUDE  SHORTSTART.TEXT
                                         ; calls starter

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

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

Version II.0   February 1979

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.

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

ARRAY   : a PACKED ARRAY OF CHARacters
BLOCK   : one disk block, {512 bytes}
BLOCKS   : an INTEGER number of blocks
BLOCKNUMBER   : an absolute disk block address
BOOLEAN   : any BOOLEAN value
CHARACTER   : any expression which evaluates to a character
DESTINATION   : a PACKED ARRAY OF CHARacters to write into or
                      a STRING, context dependent
EXPRESSION   : part or all of an expression, to be specified
FILEID   : a file identifier, must be
                              VAR fileid: FILE OF <type>;
                                      or TEXT;
                                      or INTERACTIVE;
                                      or FILE;
INDEX   : an index into a STRING or PACKED ARRAY OF CHARacters,
                      context dependent or as specified.
NUMBER   : a literal or identifier whose type is either INTEGER
                      or REAL.
RELBLOCK   : a relative disk block address, relative to the start
                      of the file in context, the first block being
                      block zero.
SIMPLVARIABLE   : any declared PASCAL variable which is of one of the
                      following TYPEs:
                              BOOLEAN CHAR REAL STRING

Pages 113 and 114 -- numbering error.... ed.

```
                               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.  In the case of string intrinsics, it
                      must be STRING.
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 II.0   February 1979
FUNCTION LENGTH (STRING) : INTEGER

        Returns the integer value of the length of the STRING.

        Example:

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

        Will print:

                7 0

FUNCTION POS (STRING , SOURCE) : INTEGER

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

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

        Will print:

                26

FUNCTION CONCAT (SOURCEs) : STRING

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

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

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

                Will print:

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

FUNCTION COPY (SOURCE , INDEX , SIZE) : STRING

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

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

            Will print:

                SOMETHING

PROCEDURE DELETE (DESTINATION , INDEX , SIZE)

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

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

            Will print:

                THIS STRING HAS MANY CHARACTERS IN IT.

PROCEDURE INSERT (SOURCE , DESTINATION , INDEX)

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

Example:

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

            Will print:

                INSERT DEMONSTRATIONS

PROCEDURE STR (LONG , DESTINATION)

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

Example:

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

Will print:

$1020395.03

Note about using strings and string functions:

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

-- Notes --

```
****************************** ****************
* INPUT AND OUTPUT INTRINSICS * * Section 2.1.2 *
****************************** ****************
```

Version I.5   September 1978

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

These procedures open files for reading and writing and mark the
file as open. The FILEID may be any PASCAL structured file as
defined in Section 2.1, and the TITLE is a string containing any legal
file title as defined in Section 1.2 Figure 2.

The difference between them is that REWRITE creates a new file on
disk for output files; RESET 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 are 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.

Note that RESETting a file to an output only device, such as the
lineprinter may cause an non-zero IORESULT as a result of the implied
GET caused by the RESET.

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.
Unitnumber is the index into the physical device table, Table 3
describes these numbers. The ARRAY is any declared packed array,
which may be subscripted to indicate a starting position some machines
may be sensitive to having the starting position be on a word
boundary. 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 ', ,INTEGER' 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.

        Currently implemented only on DEC computers.

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};

        Currently implemented only on DEC computers.

PROCEDURE UNITCLEAR (UNITNUMBER);

        UNITCLEAR cancels all I/Os to the specified unit and resets the hardware to its power-up state. Sets IORESULT non-zero if unit is not present.

```
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. FILEID: FILE;). The length of ARRAY should be an integer multiple of 512. ARRAY may be indexed to indicate a starting position in the array, however care must be taken as some machines may be sensitive to having the I/O take place to a word boundary. 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. Specifying RELBLOCK for an 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 LOCKs the file to the point of last access. i.e. the position of the last GET or PUT to the file is where the file will end.

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. If, while doing

puts or writes at the end of a file, the file cannot be expanded to
accommodate the PUT or WRITE, EOF(FILEID) will return true.

FUNCTION IORESULT : INTEGER ;

    After any I/O operation, IORESULT contains an INTEGER value
corresponding to the values given in Table 2. If the compiler is
allowed (i.e. (*$I-*) has not been used), it will generate checks
after each I/O operation, causing the program to get a run-time
error on any bad I/O operation. These are not generated any time
after any UNITREAD or UNITWRITE.

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;'. 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.
The actual physical disk access may not occur until the next time the
physically associated block of the diskis no-longer considered the
current working block. The kinds of operation which tend to force the
block to be written are: a SEEK to elsewhere in the file, a RESET, and
CLOSE. Successive GETs or PUTs to the file will cause the physical
I/O to happen when the block boundaries are crossed.

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: INPUT, OUTPUT, and KEYBOARD. INPUT results in echoing of
characters typed to the console device. KEYBOARD does no echoing and
allows the programmer complete control of the response to user typing.
OUTPUT allows the user to halt or flush the output.

PROCEDURE PAGE (FILEID);

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

PROCEDURE SEEK (FILEID, INTEGER);

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

 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; as it is convenient not to have to figure out or remember
the number of bytes in a particular data structure. (Which may change
at the programmers whim.)

-- Notes --

```
*****************************  ******************
* MISCELLANEOUS ROUTINES *  * Section 2.1.3 *
*****************************  ******************
```

Version II.0   February 1979

FUNCTION SIZEOF (VARIABLE OR TYPE IDENTIFIER) : INTEGER;

        This function returns the number of bytes that the "item"
passed as a parameter occupies. 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);
                (* may not be implemented in all machines *)

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, therefore, if the Debugger is not
in core when this occurs, a fatal run-time error, #14, will occur.

PROCEDURE GOTOXY(XCOORD , YCOORD: INTEGER);

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 4.7.

FUNCTION MEMAVAIL: INTEGER;

This function returns the number of words currently between the top-of-stack and top-of-heap. This can be interpreted as the amount of memory available at that time. One must take into consideration the size of evaluation stacks, and error-procedure calls.

```
******************* *****************
*   O  O  P  S     * * Section 2.1.4 *
******************* *****************
```

Version I.5   September 1978

Out Of Place Section

— Notes —

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

Version I.5   September 1978

CAUTION

        These intrinsics are all byte oriented. Use them with care.
Read the descriptions carefully before trying them out as no range
checking of any sort is performed on the parameters passed to these
routines. The programmer should know exactly what he is doing before
he does it since the system does not protect itself from these
operations. There may lurk some machine dependencies in the
implementations of these, beware of byte/word and byte-sex problems.

FUNCTION SCAN (LENGTH, PARTIAL EXPRESSION, ARRAY) : INTEGER;

        This function returns the number of characters from the
starting position to where it terminated, i.e. the number of
characters scanned. It terminates on either matching the specified
LENGTH or satisfying the EXPRESSION. The ARRAY should be a PACKED
ARRAY OF CHARACTERS and may be subscripted to denote the starting
point. If the expression is satisfied on the character at which ARRAY
is pointed, the value returned will be zero. If the length passed was
negative, the number returned will also be negative, and the function
will have scanned backward. The PARTIAL EXPRESSION must be of the
form:

                "<>" or "=" followed by <character expression>

Examples:
        Using the array:
        DEM := '......THE TERAK IS A MEMBER OF THE PTERODACTYL FAMILY.';

        SCAN(-26,=':',DEM[30]);
                                will return -26
        SCAN(100,<>'.',DEM);
                                will return 5
        SCAN(15,=' ',DEM[0]);
                                will return 8

PROCEDURE MOVELEFT (SOURCE, DESTINATION, LENGTH);
PROCEDURE MOVERIGHT (SOURCE, DESTINATION, LENGTH);

These functions do mass moves of bytes for the length specified.
MOVELEFT starts from the left end of the specified source and moves
bytes to the left end of the destination. MOVERIGHT starts from the
right ends of both arrays and also moves byte by byte.

Some implementations of these intrinsics may do optimization of
such a move for the specific hardware involved.

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: |HE TEXT INE TEXT IN THIS ARRAY|
            MOVELEFT(ARAY[1],ARAY[3],10)
    ARAY: |HEHEHEHEHEHETEXT IN THIS ARRAY|
            MOVELEFT(ARAY[23],ARAY[2],8);
    ARAY: |HIS 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],LENGTH-1);

but FILLCHAR is twice as fast, as no memory reference is needed for a
source.

    See the note about move optimization in the section on MOVELEFT.
The notes about MOVELEFT also apply to FILLCHAR.
    The notes about MOVELEFT also apply to FILLCHAR.
    The intrinsic SIZEOF (Section 2.1.6) is meant for use with these
intrinsics; as it is convenient not to have to figure out or remember
the number of bytes in a particular data structure. (Which may change
at the programmers whim.)

```
*********************************************************** **************
* DIFFERENCES BETWEEN UCSD PASCAL AND STANDARD PASCAL* * Section 2.2 *
*********************************************************** **************
```

This section is a summary and quick referrence guide which
notes the areas in which UCSD Pascal differs from Standard Pascal,
and refers the user to the appropriate documents which explain various
aspects of UCSD Pascal. The Standard Pascal referred to by this
section is defined in PASCAL USER MANUAL AND REPORT (2nd edition) by
Kathleen Jensen and Niklaus Wirth (Springer-Verlag, 1975).

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

2.2.1   CASE STATEMENTS

Jensen and Wirth on page 31, state that if there is no label
equal to the value of the case statement selector, the result of the
case statement is undefined. UCSD 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.
```

2.2.2 COMMENTS

The UCSD Pascal compiler recognizes any text appearing between either the symbols "(*" and "*)" or the symbols "{" and "}" as a comment. Text appearing between these symbols is ignored by the compiler unless the first character of the comment is a dollarsign, in which case the comment is interpreted as a compiler control comment. See section 1.6 "Pascal Compiler" for details on compiler control comments.

If the beginning of the comment is delimited by the "(*" symbol, the end of the comment must be delimited by the matching "*)" symbol, rather than the "}" symbol. When the comment begins with the "{" symbol) the comment continues until the matching "}" symbol appears. This feature allows a user to "comment out" a section of a program which itself contains comments. For example:

```
{    XCP := XCP + 1;  (* ADJUST FOR SPECIAL CASE... *)   }
```

The compiler does not keep track of nested comments. When a comment symbol is encountered, the text is scanned for the matching comment symbol. The following text will result in a syntax error:

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

2.2.3 DYNAMIC MEMORY ALLOCATION

The standard procedure DISPOSE defined on page 158 of Jensen and Wirth is not implemented in UCSD Pascal. However, the function of DISPOSE can be approximated by a combined use of the UCSD intrinsics MARK and RELEASE. The process of recovering memory space described below is only an approximation to the function of DISPOSE as one cannot explicitly ask that the storage occupied by one particular variable be released by the system for other uses.

The current UCSD implementation allocates storage for variables created by use of the standard procedure NEW in a stack-like structure called the "heap". The following program is a simple demonstration of how MARK and RELEASE can be used to change in the size of the heap.

```
PROGRAM SMALLHEAP;

TYPE  PERSON=
        RECORD
           NAME: PACKED ARRAY[0..15] OF CHAR;
           ID: INTEGER
        END;
```

```
VAR P: ^PERSON;(* "^" means "pointer to" as defined in J&W *)
    HEAP: ^INTEGER;

BEGIN
  MARK(HEAP);
  NEW(P);
  P^.NAME:='FARKLE, HENRY J.';
  P^.ID:= 999;
  RELEASE(HEAP);
END.
```

The above program first calls MARK to place the address of the
current top of heap into the variable HEAP. HEAP must be declared to
be a pointer to an INTEGER. The parameter supplied to MARK must be a
pointer to an INTEGER. This is a UCSD restriction. This is a
particularly handy construct for deliberately accessing 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  |
                |                    |
```
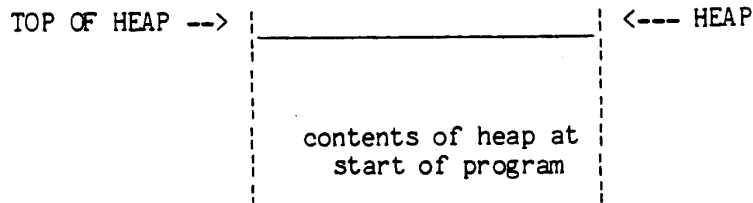

Next the program calls the standard procedure NEW and this
results in a new variable P^ which is located in the heap as shown in
the diagram below:


```
TOP OF HEAP --->  _____
                 |                    |
                 |         P^         |
                 |_____|   <--- HEAP
                 |                    |
                 |  contents of heap at |
                 |    start of program  |
                 |                    |
```
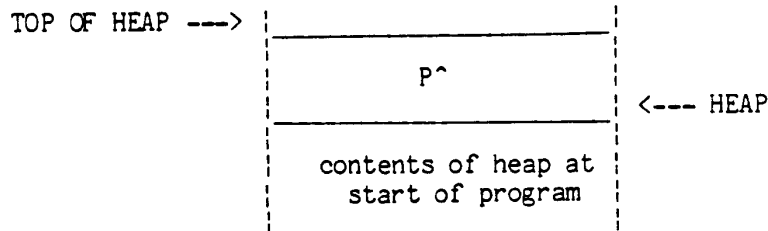
After the RELEASE the heap is as follows:

```
TOP OF HEAP --->  _____   <--- HEAP
                 |                    |
                 |  contents of heap at |
                 |    start of program  |
                 |                    |
```
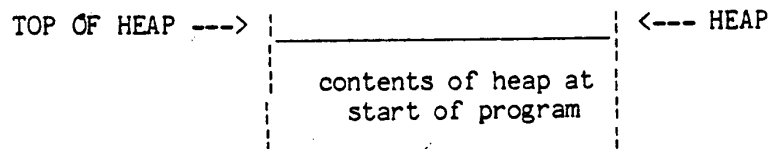
Once the program no longer needs the variable P^ and wishes to
"release" this memory space to the system for other uses, it calls
RELEASE which resets the top of heap to the address contained in the
variable HEAP.

If the above sample program had made a series of calls to the
standard procedure NEW between the calls to MARK and RELEASE, the
storage occupied by several variables would have been released at
once. Note that due to the stack nature of the heap it is not possible
to release the memory space used by a single item in the middle of the
heap. It is for this reason the use of MARK and RELEASE can only
approximate the function of DISPOSE as described in Jensen and Wirth.

Furthermore, it should be noted that careless use of the
intrinsics MARK and RELEASE can leave "dangling pointers", pointing to
areas of memory which are no longer part of the defined heap space.

2.2.4   EOF(F)

To set EOF to TRUE for a textfile F being used as an input file
from the CONSOLE device, the user must type the EOF character. The
EOF character can be altered by a suitable reconfiguration of the
system variable SYSCOM^.CRTINFO.EOF using SETUP. For further
information concerning system configuration and the SETUP program see
Section 4.3.

If F is closed, for any FILE F, EOF(F) will return the value
TRUE. If EOF(F) is TRUE , and F is a FILE of type TEXT, EOLN(F) is
also TRUE. After a RESET(F), EOF(F) is FALSE. If EOF(F) becomes TRUE
during a GET(F) or a READ(F,...) the data obtained thereby is not
valid.

When a user program starts execution, the system performs a
RESET on the predeclared files INPUT, OUTPUT, and KEYBOARD. See
section 2.2.11 READ for further details concerning the predeclared file
KEYBOARD.

As defined in Jensen and Wirth, EOF and EOLN by default will
refer to the file INPUT if no file identifier is specified.

2.2.5   EOLN(F)

EOLN(F) is defined only if the <type> of the window variable,
F^, is of type CHAR. EOLN becomes TRUE only after reading the end of
line character. The end of line character is a carriage return. In
the example program below, care must be taken as regards when the
carriage return is typed while inputing data:

```
PROGRAM ADDLINES;
VAR K,SUM:INTEGER;

BEGIN
  WHILE NOT EOF(INPUT) DO
    BEGIN
      SUM:=0;
      READ(INPUT,K);
      WHILE NOT EOLN(INPUT) DO
        BEGIN
          SUM:=SUM+K;
          READ(INPUT,K);
        END;
      WRITELN(OUTPUT);
      WRITELN(OUTPUT,'THE SUM FOR THIS LINE IS ', SUM);
    END;
END.
```

In order for EOLN(F) to be TRUE in the above program, the carriage return must be typed immediately after the last digit of the last integer on that line. If instead a space is typed followed by the carriage return, EOLN will remain FALSE and another READ will take place. See Section 2.2.11 for details on the behavior of READ(integer).

2.2.6   FILES

A. INTERACTIVE FILES

Files of <type> INTERACTIVE behave exactly as files of <type> TEXT. The standard predeclared files INPUT and OUTPUT will always be defined to be of <type> INTERACTIVE. All files of any <type> other than INTERACTIVE, are defined to operate exactly as described in Jensen and Wirth. For files which are not of <type> INTERACTIVE, the definitions of EOF(F), EOLN(F), and RESET(F) are exactly as presented in Jensen and Wirth. For more details concerning files of <type> INTERACTIVE see section 2.2.11 "READ AND READLN" and section 2.2.12 "RESET" and section 2.1.2..

B. UNTYPED FILES

UCSD Pascal has one type of file declaration which in not found in the syntax of Jensen and Wirth. This type and its use is demonstrated in the sample program below:

```
(*$I-*)
PROGRAM FILEDEMO;

VAR
   BLOCKNUMBER,BLOCKSTRANSFERRED:INTEGER;
   BADIO: BOOLEAN;
   G,F: FILE;
   BUFFER:   PACKED ARRAY[0..511] OF CHAR;

(* This program reads a diskfile called 'SOURCE.DATA' and
    copies the file into another diskfile called 'DESTINATION'
    using untyped files and the intrinsics BLOCKREAD and
    BLOCKWRITE *)

BEGIN
   BADIO:=FALSE;
   RESET(G,'SOURCE.DATA');
   REWRITE(F,'DESTINATION');
   BLOCKNUMBER:=0;
   BLOCKSTRANSFERRED:=BLOCKREAD(G,BUFFER,1,BLOCKNUMBER);
   WHILE (NOT EOF(G)) AND (IORESULT=0) AND (NOT BADIO) AND
         (BLOCKSTRANSFERRED=1) DO
     BEGIN
       BLOCKSTRANSFERRED:=BLOCKWRITE(F,BUFFER,1,BLOCKNUMBER);
       BADIO:=((BLOCKSTRANSFERRED<1) OR (IORESULT<>0));
       BLOCKNUMBER:=BLOCKNUMBER+1;
       BLOCKSTRANSFERRED:=BLOCKREAD(G,BUFFER,1,BLOCKNUMBER);
     END;
   CLOSE(F,LOCK);
END.
```

The two files which are declared and used in the above sample
program are both untyped files. An untyped file F can be thought of as
a file without a window variable F^ to which all I/O must be
accomplished by using the functions BLOCKREAD and BLOCKWRITE. Note
that any number of blocks can be transferred using either BLOCKREAD or
BLOCKWRITE. The functions return the actual number of blocks read. A
somewhat sneaky approach to doing a quick transfer would be:

       WHILE BLOCKWRITE(F,BUFFER,BLOCKREAD(G,BUFFER,BUFBLOCKS))>0 DO (*IT*);

       This is, however considered unclean. The program above has
been compiled using the (*$I-*) Compile Time Option, thereby requiring
that the function IORESULT and the number of blocks transferred be
checked after each BLOCKREAD or BLOCKWRITE in order to detect any I/O
errors that might have occurred.

C. RANDOM ACCESS OF FILES

The UCSD implementation of structured files supports the
ability to randomly access individual records within a file by means
of the intrinsic SEEK. SEEK expects two parameters, the first being
the file identifier, and the second, an integer specifying the record
number to which the window should be moved. The first record of a
structured file is numbered record 0. The following sample program
demonstrates the use of SEEK to randomly access and update records in
a file:

```
PROGRAM RANDOMACCESS;
VAR
  RECNUMBER: INTEGER;
  CH: CHAR;
  DISK: FILE OF RECORD
                NAME: STRING[20];
                DAY,MONTH,YEAR: INTEGER;
                ADDRESS: PACKED ARRAY[0..49] OF CHAR;
                ALIVE: BOOLEAN
              END;

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;

      (* Must point the window back to the record
         since GET(DISK) advances the window to
         the next record after loading DISK^       *)


    SEEK(DISK,RECNUMBER);
    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. See Section 2.1.2.

D. READ AND WRITE FROM ARBITRARILY TYPED FILES

It is not currently possible to READ or WRITE to files of type
other than TEXT or FILE OF CHAR.

2.2.7 GOTO AND EXIT STATEMENTS

UCSD has a more limited form of GOTO statement than is defined
as the standard in Jensen and Wirth. UCSD's GOTO statement prohibits
a GOTO statement to a label which is not within the same procedure
block as the GOTO statement itself. The examples presented on pages
31- 32 of Jensen and Wirth are not legal in UCSD Pascal.

EXIT is a UCSD extension which accepts as its single parameter
the identifier of a procedure to be exited. The use of an EXIT
statement to exit a FUNCTION can result in the FUNCTION returning
undefined values if no assignment to the FUNCTION identifier is made
prior to the execution of the EXIT statement. Below is an example of
the use of the EXIT statement:

```
PROGRAM EXITDEMO;
VAR T: STRING;
    CN: INTEGER;

  PROCEDURE 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
```

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, the most recent invocation of that procedure will be
exited. If, in the above example, one or both of the procedures P and
Q declared and opened some local files, an implicit CLOSE(F) is done
when the EXIT(Q) statement is executed, as if the procedures P and Q
terminated normally.

The EXIT statement may also be used to exit a Pascal program
by EXIT(PROGRAM) or EXIT(programname).

The creation of the EXIT statement at UCSD 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 UCSD Pascal compiler. The routine use of the
EXIT statement is, nevertheless, discouraged.

2.2.8 PACKED VARIABLES

A. PACKED ARRAYS

The UCSD compiler will perform packing of arrays and
records if the ARRAY or RECORD declaration is preceded by the word
PACKED. For example, consider the following declarations:

    A: ARRAY[0..9] OF CHAR;

    B: PACKED ARRAY[0..9] OF CHAR;

The array A will occupy ten 16 bit words of memory, with each
element of the array occupying 1 word. The PACKED ARRAY B on the other
hand will occupy a total of only 5 words, since each 16 bit word
contains two 8 bit characters. In this manner each element of the
PACKED ARRAY B is 8 bits long.

PACKED ARRAYs need not be restricted to arrays of type CHAR,
for example:

    C: PACKED ARRAY[0..1] OF 0..3;

    D: PACKED ARRAY[1..9] OF SET OF 0..15;

    D2: PACKED ARRAY[0..239, 0..319] OF BOOLEAN;

Each element of the PACKED ARRAY C is only 2 bits long, since
only 2 bits are needed to represent the values in the range 0..3.
Therefore C occupies only one 16 bit word of memory, and 12 of the bits
in that word are unused. The PACKED ARRAY D is a 9 word array, since
each element of D is a SET which can be represented in a minimum of 16
bits. Each element of a PACKED ARRAY OF BOOLEAN, as in the case of D2
in the above example, occupies only one bit.

The following 2 declarations are not equivalent due to the recursive nature of the compiler:

    E: PACKED ARRAY[0..9] OF ARRAY[0..3] OF CHAR;

    F: PACKED ARRAY[0..9,0..3] OF CHAR;

The second occurrence of the reserved word ARRAY in the declaration of E causes the packing option in the compiler to be turned off E becomes an unpacked array of 40 words. On the otherhand, the PACKED ARRAY F occupies 20 total words because the reserved word ARRAY occurs only once in the declaration. If E had been declared as

    E: PACKED ARRAY[0..9] OF PACKED ARRAY[0..3] OF CHAR;

            or as

    E: ARRAY[0..9] OF PACKED ARRAY[0..3] OF CHAR;

F and E would have had identical configurations.

The reserved word PACKED only has true significance before the last appearance of the reserved word ARRAY in a declaration of a PACKED ARRAY. When in doubt a good rule of thumb when declaring a multidimensional PACKED ARRAY is to place the reserved word PACKED before every appearance of the reserved word ARRAY to insure that the resultant array will be PACKED.

The resultant array will only be packed if the final type of the array is scalar, or subrange, or a set which can be represented in 8 bits or less. The final type can also be BOOLEAN or CHAR. The following declaration will result in no packing whatsoever because the final type of the array cannot be represented in a field of 8 bits:

    G: PACKED ARRAY[0..3] OF 0..1000;

G will be an array which occupies 4 16 bit words.

Packing never occurs across word boundaries. This means that if the type of the element to be packed requires a number of bits which does not divide evenly into 16, there will be some unused bits at the high order end of each of the words which comprise the array.

Note that a string constant may be assigned to a PACKED ARRAY OF CHAR but not to an unpacked ARRAY OF CHAR. Likewise, comparisons between an ARRAY OF CHAR and a string constant are illegal. (These are temporary implementation restrictions which will be removed in the next major release.) Because of their different sizes, PACKED ARRAYs cannot be compared to ordinary unpacked ARRAYs. For further information regarding PACKED ARRAYs OF CHARacters see section 2.2.16 "STRINGS".

A PACKED ARRAY OF CHAR may be output with a single write statement:

```
PROGRAM VERYSLICK;
VAR T: PACKED ARRAY[0..10] OF CHAR;
BEGIN
  T:='HELLO THERE';
  WRITELN(T);
END.
```

Initialization of a PACKED ARRAY OF CHAR can be accomplished very efficiently by using the UCSD 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 UCSD intrinsics see section 2.1.5 "CHARACTER ARRAY MANIPULATION INTRINSICS".

B.   PACKED RECORDS

The following RECORD declaration declares a RECORD with 4 fields. The entire RECORD occupies one 16 bit word as a result of declaring it to be a PACKED RECORD.

```
VAR R: PACKED RECORD
        I,J,K: 0..31;
        B: BOOLEAN
      END;
```

The variables I, J, K each take up 5 bits in the word. The boolean variable B is allocated to the 16'th bit of the same word.

In much the same manner that PACKED ARRAYs can be multidimensional PACKED ARRAYs, PACKED RECORDS may contain fields which themselves are PACKED RECORDS or PACKED ARRAYS. Again, slight differences in the way in which declarations are made will affect the degree of packing achieved. For example, note that the following two declarations are not equivalent:

```
VAR A:PACKED RECORD              VAR B:PACKED RECORD
        C:INTEGER;                       C:INTEGER;
        F:PACKED RECORD                  F:RECORD
                R: CHAR;                         R:CHAR;
                K: BOOLEAN                       K:BOOLEAN
              END;                             END;
        H:PACKED ARRAY[0..3] OF CHAR      H:PACKED ARRAY[0..3] OF CHAR
      END;                             END;
```

As with the reserved word ARRAY, the reserved word PACKED must appear with every occurrence of the reserved word RECORD in order for the PACKED RECORD to retain its packed qualities throughout all fields of the RECORD. In the above example, only RECORD A has all of its fields packed into one word. In B, the F field is not packed and therefore occupies two 16 bit words. It is important to note that a packed or unpacked ARRAY or RECORD which is a field of a PACKED RECORD will always start at the beginning of the next word boundary. This means that in the case of A, even though the F field does not completely fill one word, the H field starts at the beginning of the next word boundary.

A case variant may be used as the last field of a PACKED RECORD, and the amount of space allocated to it will be the size of the largest variant amoung the various cases. The actual nature of the packing is 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

UCSD Pascal does'not support the standard procedures PACK and UNPACK as defined in Jensen and Wirth on page 106. If a type or variable is declared as packed, the packing and unpacking in UCSDs Pascal system is implicit.

2.2.9 PARAMETRIC PROCEDURES AND FUNCTIONS

UCSD 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 5.9 for a revised syntax diagram of <parameter-
list>.

2.2.10 PROGRAM HEADINGS

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

2.2.11 READ AND READLN

Given the following declarations:

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

the statement READ(F,CH) is defined by Jensen and Wirth on page 85 to
be equivalent to the two statement sequence:

```
CH:=F^;      J & W
GET(F);      method
```

In other words, the standard definition of the standard
procedure READ requires that the process of opening a file load the
"window variable" F^ with the first character of the file. In an
interactive programming environment, it is not convenient to require a
user to type in the first character of the input file at the time when
the file is opened. If this were the case, every program would "hang"
until a character was typed, whether or not the program performed any
input operations at all. In order to overcome this problem, UCSD
Pascal defines an additional file <type> called INTERACTIVE. Declaring
a file F to be of <type> INTERACTIVE is equivalent to declaring F to be
of type TEXT, the difference being that the definition of the statement
READ(F,CH) is the reverse of the sequence specified by the standard

definition for files of <type> TEXT: i.e.

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

This difference affects the way in which EOLN must be used within a program when reading from a textfile of type INTERACTIVE. As in section 5, EOLN becomes true only after reading the end of line character, a carriage return. When this is read, EOLN is set to true and the character returned as a result of the READ will be a blank. In the following example, the left fragment is taken from Jensen and Wirth; only the RESET and REWRITE statements have been altered. The program on the left will correctly copy the textfile represented by the file X to the file Y. The program fragment on the right performs a 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;                      PROGRAM UCSDVERSION;
VAR X,Y:TEXT;                       VAR X,Y:INTERACTIVE;
    CH: CHAR;                           CH:CHAR;
BEGIN                               BEGIN
  RESET(X,'SOURCE.TEXT');             RESET(X,'CONSOLE:');
  REWRITE(Y,'SOMETHING.TEXT');        REWRITE(Y,'SOMETHING.TEXT');
                                      READ(X,CH);
  WHILE NOT EOF(X) DO                 WHILE NOT EOF(X) DO
    BEGIN                               BEGIN
      WHILE NOT EOLN(X) DO                 WHILE NOT EOLN(X) DO
        BEGIN                                 BEGIN
          READ(X,CH);                           WRITE(Y,CH);
          WRITE(Y,CH);                          READ(X,CH);
        END;                                  END;
      READLN(X);                           READLN(X);
      WRITELN(Y);                          WRITELN(Y);
    END;                                END;
  CLOSE(Y,LOCK);                      CLOSE(Y,LOCK);
END.                                END.
```

Note that the textfiles X and Y in the above two programs had to be opened by using the UCSD extended form of the standard procedures RESET and REWRITE.

The CLOSE intrinsic was applied to the file Y in both versions of the program in order to make it a permanent file in the disk directory called "SOMETHING.TEXT". Likewise, the textfile X could have been a diskfile instead of coming from the CONSOLE device in the right hand version of the program.

There are three predeclared textfiles which are automatically opened by the system for a user program. These files are INPUT, OUTPUT, and KEYBOARD. The file INPUT defaults to the CONSOLE device.

and is always defined to be of <type> INTERACTIVE. The statement
READ(INPUT,CH) where CH is a character variable, will echo the
character typed from the CONSOLE back to the CONSOLE device. WRITE
statements to the file OUTPUT will, by default, cause the output to
appear on the CONSOLE device. The file KEYBOARD is the non-echoing
equivalent to INPUT. For example, the two statements

                    READ(KEYBOARD,CH);
                    WRITE(OUTPUT,CH);

are equivalent to the single statement READ(INPUT,CH).

        Reading the type integer causes preceding blanks and end-of-
lines to be flushed until a non-blank character is observed. Reading
the type BOOLEAN is not implemented.

        For more documentation regarding the use of files see sections:
                2.2.6   "FILES"
                2.2.4   "EOF"
                2.2.5   "EOLN"
                2.2.17 "WRITE AND WRITELN"
                2.2.12 "RESET"

        See section 2.1.2 "INPUT/OUTPUT INTRINSICS" for more details
on the UCSD intrinsics.

2.2.12   RESET(F)

        The standard procedure RESET, as defined on page 9 of Jensen
and Wirth, resets the file window to the beginning of the file F. The
next GET(F) or PUT(F) will affect record number 0 of the file. In
addition, the standard definition of RESET(F) states that the window
variable F^ be loaded with the first record in the file. The UCSD
implementation of RESET(F) operates exactly as the standard definition,
unless the file F is declared to be of <type> INTERACTIVE in which case
the statement RESET(F) points the file window to the start of the file,
but does not load the window variable F^. Thus, for files of <type>
INTERACTIVE, the UCSD equivalent of the standard definition of
RESET(F) is the two statement sequence:

                    RESET(F);   makes INTERACTIVE
                    GET(F);   look like TEXT

        UCSD Pascal defines an alternative form of the standard
procedure RESET which is used to open a pre-existing file. In it,
RESET has two parameters, the first being the file identifier; the
second, either a STRING constant or variable which corresponds to the
directory filename of the file being opened. See section 2.1.2
"INPUT/OUTPUT INTRINSICS" for more information on this use of RESET.

2.2.13 REWRITE(F)

The standard procedure REWRITE is used to open and create a new file. REWRITE has two parameters, the first, being the file identifier, the second corresponds to the directory filename of the file being opened, and must be either a STRING constant or variable. For example, the statement REWRITE(F,'SOMEINFO.TEXT') causes the file F to be opened for output, and, if the file is locked onto the disk, the filename of the file in the directory will be "SOMEINFO.TEXT". See section 2.1.2 "INPUT/OUTPUT INTRINSICS" for further documentation regarding the use of REWRITE to open a file.

2.2.14 SEGMENT PROCEDURES

The concept of the SEGMENT PROCEDURE is a UCSD 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 UCSD Pascal system is necessarily partitioned in this manner because it is too large to fit into the memory of most small interactive computers at one time.

The following is an example of the use of SEGMENT PROCEDURES:

```
PROGRAM SEGMENTDEMO;

(*  GLOBAL DECLARATIONS GO HERE  *)

PROCEDURE PRINT(T:STRING); FORWARD;

SEGMENT PROCEDURE ONE;
  BEGIN
    PRINT('SEGMENT NUMBER ONE');
  END;

SEGMENT PROCEDURE TWO;
  SEGMENT PROCEDURE THREE;
    BEGIN
      ONE;
      PRINT('SEGMENT NUMBER THREE');
    END;
  BEGIN (* SEGMENT NUMBER TWO *)
    THREE;
    PRINT('SEGMENT NUMBER TWO');
  END;

  PROCEDURE PRINT;
  BEGIN
    WRITELN(OUTPT,T);
  END;

BEGIN
  TWO;
  WRITELN('I''M DONE');
END.
```

The above program will give the following output:

```
    SEGMENT NUMBER ONE
    SEGMENT NUMBER THREE
    SEGMENT NUMBER TWO
    I'M DONE
```

For further documentation on SEGMENT PROCEDURES, their use and syntax governing their declaration, see Section 3.3 - 'SEGMENT PROCEDURES'.

2.2.15   SETS

UCSD Pascal supports all of the constructs defined for sets on pages 50-51 of Jensen and Wirth. Sets (of enumeration values) are limited to positive integers only. Space is assigned, rounding up to word boundaries, in a bitwise fashion, starting at zero, up to 4079, inclusive. Therefor a set can be at most 255 words in size, and have at most 4080 elements.

Comparisons and operations on sets are allowed only between sets which are either of the same base type or subranges of the same underlying type. For example, in the sample program below, the base type of the set S is the subrange type 0..49, while the base type of the set R is the subrange type 1..100. The underlying type of both sets is the type INTEGER, which by the above definition of 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.
```

In the following example, the construct I = J   is not legal since the two sets are of two distinct underlying types.

```
PROGRAM ILLEGALSETS;
TYPE STUFF=(ZERO,ONE,TWO);
VAR I: SET OF STUFF;
    J: SET OF 0..2;

BEGIN
  I:= [ZERO];
  J:= [1,2];
  IF I = J THEN ...        <<<< error here
END.
```

2.2.16   STRINGS

UCSD 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
intrinsic LENGTH. The default maximum LENGTH of a STRING variable is
80 characters but can be overridden in the declaration of a STRING
variable by appending the desired LENGTH of the STRING variable within
[ ] after the reserved type identifier STRING. Examples of
declarations of STRING variables are:

        TITLE: STRING;   (* defaults to a maximum length of 80 characters *)

        NAME: STRING[20];   (* allows the STRING to be a maximum of 20
                              characters*)

Note that a STRING variable has an absolute maximum length of
255 characters. Assignments to string variables can be performed using
the assignment statement, the UCSD 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; beware of strings of length zero! The following
sequence will result in an invalid index run time error:

```
                  TITLE:= '1234';
                  TITLE[5]:= '5';
```

A variable of type STRING may be compared to any other variable
of type STRING or a string constant no matter what its current dynamic
LENGTH. Unlike comparisons involving variables of other types, STRING
variables may be compared to items of a different LENGTH. The
resulting comparison is lexicographical. The following program is a
demonstration of legal comparisons involving variables of type STRING:

```
            PROGRAM  COMPARESTRINGS;
            VAR S: STRING;
                T: STRING[40];

            BEGIN
              S:= 'SOMETHING';
              T:= 'SOMETHING BIGGER';
              IF S = T THEN
                WRITELN('Strings do not work very well')
              ELSE
                IF S > T THEN
                  WRITELN(S,' is greater than ',T)
                ELSE
                  IF S < T THEN
                    WRITELN(S,' is less than ',T);
              IF S = 'SOMETHING' THEN
                WRITELN(S,'  equals  ',S);
              IF S > '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 produces 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 UCSD Pascal system is reading file names from the CONSOLE device:

```
PROGRAM LISTER;
VAR BUFFER: PACKED ARRAY[0..511] OF CHAR;
    FILENAME: STRING;
    F: FILE;

BEGIN
  WRITE('Enter filename of the file to be listed --->');
  READLN(FILENAME);
  RESET(F,FILENAME);
  WHILE NOT EOF(F) DO
    BEGIN
      ...
      ...
      ...
    END;
END.
```

When a variable of type STRING is a parameter to the standard procedure READ and READLN, all characters up to the end of line character (a carriage return) in the source file will be assigned to the STRING variable. Note that care must be taken when reading STRING variables, for example, the single statement READLN(S1,S2) is equivalent to the two statement sequence READ(S1); READLN(S2). In both cases the STRING variable S2 will be assigned the empty string.

For further information concerning the predeclared type STRING see Section 2.1.1 "STRING INTRINSICS".

2.2.17 WRITE AND WRITELN

The standard procedures WRITE and WRITELN are compatible with Standard Pascal, except with respect to a WRITE or a WRITELN of a variable of type BOOLEAN. UCSD Pascal does not support the output of the words TRUE or FALSE when writing out the value of a BOOLEAN variable.

For a description of WRITE statements of variables of type
STRING see Section 2.1.1 "STRING INTRINSICS".

UCSD's WRITE and WRITELN do support the writing of entire
PACKED ARRAYs OF CHAR in a single WRITE statement:

```
VAR BUFFER: PACKED ARRAY[0..10] OF CHAR;
BEGIN
  BUFFER:= 'HELLO THERE'; (* contains exactly 11 characters *)
  WRITELN(OUTPUT, BUFFER);
END.
```

The above construct will work only if the ARRAY is a PACKED
ARRAY OF CHAR. See section 2.2.8 PACKED VARIABLES for further
information.

The following program demonstrates the effects of a field width
specification within a WRITE statement for a variable of type STRING:

```
PROGRAM WRITESTRINGS;
VAR S:STRING;

BEGIN
  S:='THE BIG BROWN FOX JUMPED...';
  WRITELN(S);
  WRITELN(S:30);
  WRITELN(S:10);
END.
```

The above program will produce the following output:

```
THE BIG BROWN FOX JUMPED...
    THE BIG BROWN FOX JUMPED...
THE BIG BR
```

Note that when a string variable is written without specifying
a field width, the actual number of characters written is equal to the
dynamic length of the string. If the field width specified is longer
than the dynamic length of the string, leading blanks are inserted and
written. If the field width is smaller than the dynamic length of the
string, the excess characters will be truncated on the right.

2.2.18   IMPLEMENTATION SIZE LIMITS

The following is a list of maximum size limitations imposed
upon the user by the current implementation of UCSD 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.

2.2.19   EXTENDED COMPARISONS.

UCSD Pascal allows = and <> comparisons of any array or record structure.

2.2.20   LONG INTEGERS.

UCSD Pascal allows integers of up to 36 digits. See section 3.3.3 for details regarding long integers.

2.2.21   UNITS.

UCSD Pascal now supports the modularity concept of UNITs. See section 3.3.2 for details regarding UNITs.

2.2.22   SUMMARY OF UCSD 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.
EXIT	2.2.3	Intrinsic used to exit PROCEDURES cleanly..

GOTOXY	2.1.3	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	2.1.3	Halts a user program which may result in a call to the interactive Debugger.
IDSEARCH	---	Routine used by the Pascal compiler, and the PDP-11 assembler.
INSERT	2.1.1	STRING intrinsic used to insert characters in STRING variables.
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.
MEMAVAIL	2.1.3	Returns number of words between Heap and Stack.
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	2.1.3	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	2.1.3	Function returning the number of bytes allocated to a variable.
STR	2.1.1	Procedure to convert long integer into string.

TIME	2.1.3	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. device.

— Notes —

```
***********************************  **************
*  IMPLEMENTATION NOTES - DRAWLINE *  * Section 3.1 *
***********************************  **************
```

Version II.0   February 1979

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;
ROWWIDTH, XSTART, YSTART, DELTAX, DELTAY, INK: INTEGER);

VAR X, Y, XINC, YINC, COUNT: INTEGER;

PROCEDURE DRAWDOT;

```
      PROCEDURE RADAR;
      VAR GOTIT:  BOOLEAN;
      BEGIN
        GOTIT := FALSE;
        COUNT := COUNT + 1;
        IF SCREEN [Y, X] THEN GOTIT := TRUE (*LANDED ON THE POINT*)
        ELSE (*WE MIGHT GO THROUGH A LINE*)
          IF SCREEN [Y+1, X] THEN
            GOTIT := SCREEN [Y, X+1];
        IF GOTIT THEN
          BEGIN
            RANGE := COUNT;
            EXIT(DRAWLINE)
          END;
      END (*RADAR*);

  BEGIN (*DRAWDOT*)
    CASE INK OF
        0 (*NONE*):      EXIT (DRAWLINE);   (*THEY HAD NO BUSINESS HERE*)
        1 (*WHITE*):     SCREEN [Y, X] := TRUE;
        2 (*BLACK*):     SCREEN [Y, X] := FALSE;
        3 (*REVERSE*):   SCREEN [Y, X] := NOT SCREEN [Y, X];
        4 (*RADAR*):     RADAR
      END (*CASE*)
  END (*DRAWDOT*);

  PROCEDURE DOFORX;        (*MORE HORIZONTAL*)
  VAR ERROR, I: INTEGER;
  BEGIN
    IF DELTAX = 0 THEN EXIT (DRAWLINE); (*THEY'RE GOING NOWHERE*)
    ERROR := DELTAX DIV 2;
    I := DELTAX;
    REPEAT
      ERROR := ERROR + DELTAY;
      IF ERROR >= DELTAX
        THEN BEGIN ERROR := ERROR - DELTAX;  Y := Y + YINC   END;
      X := X + XINC;
      DRAWDOT;
      I := I - 1;
    UNTIL I = 0;
  END (*DOFORX*);

  PROCEDURE DOFORY;        (*MORE VERTICAL*)
  VAR ERROR, I: INTEGER;
  BEGIN
    ERROR := DELTAY DIV 2;
    I := DELTAY;
    REPEAT
      ERROR := ERROR + DELTAX;
      IF ERROR >= DELTAY
        THEN BEGIN ERROR := ERROR - DELTAY;  X := X + XINC   END;
      Y := Y + YINC;
```

```
        DRAWDOT ;
        I := I - 1;        ,
    UNTIL I = 0;
END (*DOFORY*);

BEGIN (*DRAWLINE*)
    X := XSTART;
  . IF DELTAX < 0
      THEN BEGIN XINC := -1;   DELTAX := -DELTAX    END
      ELSE XINC := 1;
    Y := YSTART;
    IF DELTAY < 0
      THEN BEGIN YINC := -1;   DELTAY := -DELTAY    END
      ELSE YINC := 1;
    COUNT := 0;
    IF DELTAX >= DELTAY THEN DOFORX ELSE DOFORY;
    IF INK = 4 (*RADAR*) THEN RANGE := COUNT; (*HIT THE LIMIT GIVEN*)
END (*DRAWLINE*);
```

— Notes —

Version II.0   February 1979

Text files are of the format:

<1024 bytes> header page, information for editors. This space
is reserved for use by the text editors, and is respected by all
portions of the system. When a userprogram opens a TEXT file, and
REWRITEs or RESETs it with a title ending in '.TEXT', the I/O
subsystem will create and skip over the initial page. This is done to
facilitate users editing their input and/or output data. The file-
handler will transfer the header page only on a disk-disk transfer,
and will omit it on a transfer to a serial device. (i.e. transfers to
PRINTER:, and CONSOLE: will omit the header page)

<1024 byte pages> where a page is defined:
          <[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, and are a pad to the end of a page.
Because 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.

Code files have one block of information which describes the
code kept in the file. First is an array of 16 word pairs, the first
word in the pair describes the block which starts the code of the
segment which is numbered as the position in the array. The second
word is the number of bytes in that segment. For example if the third
word in the first block of a code file is an 8, and the fourth work is
1084, you now know that segment 1 of this code file starts on block 8
of the file, and has 1084 bytes of code. See Sections 3.4 and 3.5 for
notes on codefiles.

Following this array is an array of arrays of characters. The array is an array of 8 character arrays which describe the segments by name. These 8 characters are those which identify the segment at compile time. Here again, the position in this array corresponds to the segment number.

Following the array of names is an array, again 16 words long, of state descriptors. The values in this array indicate what kind of segment is at the described location. The values for this array, at present, are: LINKED,HOSTSEG,SEGPROC,UNITSEG,SEPRTSEG.

The remainder of the block, 144 words, is reserved for future use by later versions of the system. The format of the first block will most probably change completely for version II.0.

```
******************************  *****************
* SEGMENT PROCEDURE NOTES *  * Section 3.3.1 *
******************************  *****************
```

Version I.5   September 1978

Declarations of SEGMENT procedures and functions are identical
to standard Pascal procedures and functions except they are preceded by
the reserved word 'SEGMENT', for example:

```
    SEGMENT PROCEDURE INITIALIZE;
    BEGIN
      (* PASCAL code *)
    END;
```


Program behavior differs, however, as code and data for a
SEGMENT procedure (function) are in memory only while there is an
active invocation of that procedure.

Advantages and benefits:

The user may now put large pieces of one-time code, eg.
initialization code, into a SEGMENT procedure. After performing the
initialization, for example, the now-useless code is taken out of
memory thus increasing the available memory space.

Furthermore the user may now compile his/her program in chunks,
specifically in SEGMENTS. The LIBRARIAN program (described in Section
4.8) can be used to link together the separate segments to produce one
large code file.

Requirements and limitations:

The disk which holds the codefile for the program must be on-
line (and in the same drive as when the program was started) whenever
one of SEGMENT procedures is to be called. Otherwise the system will
attempt to retrieve and execute whatever information now occupies that
particular location on the disk, usually with very displeasing and
certainly unexpected results.

A maximum of seven (7) SEGMENT procedures are ordinarily available to the user, including the main body segment.

SEGMENT procedures must be the first procedure declarations containing code-generating statements.

For further details and examples see Section 3.5, INTRODUCTION TO THE PASCAL PSEUDO MACHINE.

Version I.5   September 1978

EXTERNAL COMPILATION UNITS

The UCSD Pascal I.5 system supports a facility for integrating externally compiled and assembled routines and data structures. Use of separately compiled structures allows the user to create files of frequently used routines. After a structure is compiled, the user adds it to a library, using the librarian. Files that reference that structure need not compile it directly into their code file, rather, the linker copies the existing code into the host code file. Separate compilation or assembly is supported in these areas: between portions of programs written in Pascal; between assembly language routines and Pascal hosts; and finally, between assembly language routines. Each of these areas is discussed in turn by the following sections.

3.3.2.1 PASCAL TO PASCAL LINKAGES — UNITS

A UNIT is a group of interdependent procedures, functions, and associated data structures which perform a specialized task. Whenever this task is needed within a program, the program indicates that it USES the UNIT. A UNIT consists of two parts, the INTERFACE part, which declares constants, types, variables, procedures and functions that are public and can be used by the host program, and the IMPLEMENTATION part, which declares constants, types, variables, procedures and functions that are private. These are not available to the host program and are used by the UNIT. The INTERFACE part declares how the program will communicate with the UNIT while the IMPLEMENTATION part defines how the UNIT will accomplish its task.

TURTLEGRAPHICS (example B) is a UNIT which enables the user to draw pictures using a graphics turtle. The INTERFACE consists of procedures like MOVE, TURN, and PENCOLOR, which allow the user to move the turtle and change colors. TURTLEGRAPHICS also employs DRAWLINE, an externally assembled procedure, to draw the lines and the turtle.

A program that uses TURTLEGRAPHICS has no need for DRAWLINE, and, consequently, DRAWLINE is private to that UNIT.

```
PROGRAM DRAWPOLYGON;
USES TURTLEGRAPHICS;
VAR I:INTEGER;
    SIZE,NUMSIDES:INTEGER;

BEGIN
  INITTURTLE;        (* Initialize the UNIT's variables *)
  WRITE('What size polygon?');
  READLN(SIZE);
  WRITE('How many sides?');
  READLN(NUMSIDES);
  FOR I:=1 TO NUMSIDES DO
    BEGIN
      MOVE(SIZE);
      TURN(360 DIV NUMSIDES);
    END;
END.
```

EXAMPLE A

A program must indicate the UNITs that it USES before the LABEL declaration part of the program. At the occurrence of a USES statement, the compiler references the INTERFACE part of the UNIT as though it were part of the host text itself. Therefore all public constants, types, variables, functions, and procedures are global. Name conflicts may arise if the user defines an identifier that has already been defined by the UNIT. Procedures and functions may not USE UNITs locally.

```
UNIT TURTLEGRAPHICS;
  INTERFACE
    TYPE
      TGCOLOR= ( NONE, WHITE, BLACK, REVERSE );

    PROCEDURE INITTURTLE;
    PROCEDURE TURN( RELANGLE: Integer );
    PROCEDURE MOVE( RELDISTANCE: Integer );
    PROCEDURE MOVETO( X, Y: Integer );
    PROCEDURE TURNTO( ANGLE: Integer );
    PROCEDURE PENCOLOR( PCOLOR: TGCOLOR );
```

```
IMPLEMENTATION

  CONST

    TERXSIZE = 319;
    TERYSIZE = 239;
    RADCONST = 57.29578;

  TYPE

    SCREEN = Packed
                Array [0..TERXSIZE, 0..TERYSIZE] of Boolean;

  VAR
    (* Private variables *)
    TGXPOS: Integer;
    TGYPOS: Integer;
    TGHEADING: Integer;
    TGPEN: TGCOLOR;

    I, J: Integer;
    S: SCREEN;

  (* Externally assembled procedure *)
  PROCEDURE DRAWLINE( Var RADAR: Integer; Var S: SCREEN;
                      ROW, X0, Y0, DX, DY, PEN: Integer );

    EXTERNAL; (* External declaration *)


  PROCEDURE INITTURTLE;
  BEGIN
    Fillchar( SCREEN, Sizeof(SCREEN), 0 );
    Unitwrite( 3, SCREEN, 63 );
    HEADING := 0;
    TGXPOS := 0;
    TGYPOS := 0;
  END;

  PROCEDURE MOVE;(* Public procedure, parameters declared above *)
  BEGIN
    MOVETO( Round(TURTLEX + DIST*Cos(TURTLEANGLE/RADCONST),
            Round(TURTLEY + DIST*Sin(TURTLEANGLE/RADCONST) );
  END ;
```

```
PROCEDURE MOVETO;
  VAR R: Integer;
BEGIN
  DRAWLINE( R, S, 20, 160+TURTLEX, 120-TURTLEY,
            X-TURTLEX, TURTLEY-Y, ORD(TURTLEPEN) );
END;

PROCEDURE TURN;(* Public procedure, parameters declared above *)
BEGIN
  HEADING := ( HEADING+RELANGLE ) mod 360;
END;

PROCEDURE TURNTO;
BEGIN
  HEADING := ANGLE;
END;

PROCEDURE PENCOLOR;
BEGIN
  TGPEN := PCOLOR;
END;

END. (* End of unit *)
```

EXAMPLE B

Example B is a skeleton for a TURTLEGRAPHICS UNIT. Note that the procedures MOVE, TURN, and INITTURTLE, and the TYPE TGCOLOR, are declared in the INTERFACE part and are available for use by the host program. Since the procedure DRAWLINE is not part of the INTERFACE, it is private, and may not be used by the host. The syntax for a UNIT definition is shown below. The declarations of routine headings in the INTERFACE part are similar to forward declarations; therefore, when the corresponding bodies are defined in the IMPLEMENTATION part, formal parameter specifications are not repeated.

A UNIT may also USE another UNIT, in which case the USES declaration must appear at the beginning of the INTERFACE part. In example C, PICTUREGRAPHICS indicates in the INTERFACE part that it USES TURTLEGRAPHICS. Note that the program USEGRAPHICS, which USES PICTUREGRAPHICS, indicates that it USES TURTLEGRAPHICS before using PICTUREGRAPHICS. It is important that the INTERFACE part of TURTLEGRAPHICS be defined before PICTUREGRAPHICS makes references to it, therefore this ordering is required.

NOTE: Variables of type FILE must be declared in the INTERFACE
part of a UNIT. A FILE declared in the IMPLEMENTATION part will cause
a syntax error upon compilation. This is due to the nature of
generation of initialization code for FILEs.

```
      PROGRAM USEGRAPHICS;

      UNIT PICTUREGRAPHICS;
      INTERFACE
        USES TURTLEGRAPHICS;       (* TURTLEGRAPHICS is defined in the        *)
        TYPE                       (* *system.library see section III below   *)
          PVECTOR=^VECTOR;
          VECTOR=RECORD
                    DELHEADING:INTEGER;
                    DELDISTANCE:INTEGER;
                    PENDOWN:BOOLEAN;
                    NEXTVEC:PVECTOR
                 END; (* record *)

      VAR
        START:PVECTOR;   (* Head of list of lines *)
        HEAP:^INTEGER;

     PROCEDURE MAKESUBPICTURE;

     PROCEDURE DRAWSUBPICTURE;

   IMPLEMENTATION

     PROCEDURE MAKESUBPICTURE;
     BEGIN
        (* Calculates next subpicture and stores on heap *)
     END;

     PROCEDURE DRAWSUBPICTURE;
     BEGIN
        LPVEC:=START;          (* Start at beginning of list  *)
        WHILE LPVEC<>NIL DO    (* and draw each that's there *)
          WITH LPVEC^ DO
            BEGIN
              TURN(DELHEADING);
              MOVE(DELDISTANCE);
              IF PENDOWN THEN TGPEN:=WHITE
                 ELSE TGPEN:=NONE;
              LPVEC:=NEXTVEC;
            END;
     END;  (*  drawsubpicture  *)
```

END;

USES TURTLEGRAPHICS,PICTUREGRAPHICS;   (* picturegraphics uses *)

BEGIN   (* turtlegraphics   *)
  INITTURTLE;
    REPEAT
      MARK(HEAP);
      MAKESUBPICTURE;
      DRAWSUBPICTURE;
      RELEASE(HEAP);
    UNTIL START=NIL;
  END.

                              EXAMPLE C

< Compilation unit >   ::=   < Program heading > ; { < Unit definition > ; }
                             < Uses part > < Block > |
                             < Unit definition > { ; < Unit definition > }.

< Unit definition >   ::=   < unit heading >;
                             < Interface part >
                             < Implementation part >
                             End

< Unit heading >   ::=   Unit < Unit identifier > |
                             Separate unit < Unit identifier >

< Unit identifier >   ::=   < Identifier >

< Interface part >   ::=   Interface
                             < Uses part >
                             < Constant definition part >
                             < Type definition part >
                             < Variable declaration part >
                             < Procedure heading > | < Function heading >

< Implementation part> ::=   Implementation
                             < Label declaration part >
                             < Constant definition part >
                             < Type definition part >
                             < Variable declaration part >
                             < Procedure and Function declaration part >

< Uses part >   ::=   Uses < Unit identifier >
                             { , < Unit identifier> } ; | < Empty >

    See Section 5.9 for Syntax diagrams.

                              DIAGRAM D

                                          Page 172

The user may define a UNIT in-line, after the heading of the host
program. In this case the user compiles both the UNIT, and the host
program together. Any subsequent changes in the UNIT or host program
require the user to recompile both. The user may also define and
compile a UNIT (or a group of UNITs) separately, and use the library
manager to store it (or them) in a library. After compiling a host
program that uses such a UNIT, the user must link that UNIT into the
code file by executing the LINKER. Trying to R(un an unlinked code
file will cause the LINKER to run automatically, trying to X(ecute an
unlinked file causes the system to remind you to link the file .

Changes in a host program require only that the user recompile
the program and link in the UNIT. Changes in the IMPLEMENTATION part
of a UNIT only require the user to compile the UNIT, and then to
relink all compilation units that use that UNIT. Changes in the
INTERFACE part of a UNIT require that the user recompile both the UNIT
and all compilation units that use that UNIT. In this case all these
compilation units must again be linked. For more information see
section 1.8 LINKER or section 4.2 LIBRARIAN.

The compiler generates LINKER information in the contiguous
blocks following the code for a program that uses UNITs. This
information contains locations of references to externally defined
identifiers. Section 1.8 explains the format of this information.

3.3.3.2 PASCAL TO ASSEMBLY LANGUAGE LINKAGES — EXTERNAL PROCEDURES

External procedures are separately assembled assembly language
procedures or separately compiled procedures, stored in a LIBRARY on
disk. Host programs that require external procedures must have them
linked into the compiled code file. Typically the user writes external
procedures in assembly language, to handle low-level operations that
Pascal is not designed to provide. External assembly language
procedures are also used for their comparative speed in 'real time'
applications.

A host program declares that a procedure is external in much the same way as a procedure is declared FORWARD. A standard heading is provided, followed by the keyword EXTERNAL. Calls to the external procedure use standard Pascal syntax, and the compiler checks that calls to the external agree in type and number of parameters with the external declaration. It is the user's responsibility to assure that the assembly language procedure respects the Pascal external declaration. The linker checks only that the number of words of parameters agree between the Pascal and assembly language declarations. For more information see section 1.8 Linker and 1.9 Assembler(s).

The conventions of the surrounding system concerning register use and calling sequences must be respected by writers of assembly language routines. These conventions for the PDP-11 and Z80/8080 implementations are given here.

First, for the PDP-11, registers R0 and R1 are available for use; any others affected by a routine must be saved on entry and restored on exit. The following call and return sequence is recommended for procedures. It has the advantage that calls can be made directly from assembly language as well as from Pascal.

```
        .PROC ENTRY, 2

PARAM1  .EQU    6       ;Offset for first parameter
PARAM2  .EQU    4       ;Offset for second paramter
RETADDR .EQU    2       ;Offset for return address
OLDR5   .EQU    0       ;Offset for original value of R5
LOCAL1  .EQU    -2      ;Offset for first local
LOCAL2  .EQU    -4      ;Offset for second local

        MOV     R5,-(SP)            ;Save contents of R5
        MOV     SP,R5               ;Use R5 to get at locals and parameters
        CLR     -(SP)               ;Reserve and Initialize
        CLR     -(SP)               ;Two local variables
        .
        .
        .
        ;Inside routine
        MOV     PARAM(R5),LOCAL1(R5)    ;Sample statement
        .
        .
        .
EXIT:   MOV     R5,SP               ;Cut back to entry SP
        MOV     (SP)+,R5            ;Restore previous R5
        MOV     (SP)+,R0            ;Get return address
        ADD     #NPARAMS,SP         ;Discard parameters, number of bytes
        JMP @R0                     ;Return to caller
```

In Z 80 assembly language routines, all registers are available for use, and the recommended interface sequence follows: (This code would work for both 8080's and Z80's. Optimizations are possible if the Z80 instructions are available.)

```
        .PROC   ENTRY, 2
        .PRIVATE        RETADDR,LOCAL1,LOCAL2,PARAM1,PARAM2
        ;Reserve static storage for this routine.  Much easier to
        ;reference objects like this rather than relative to
        ;register as on PDP-11
        POP     HL              ;Get return address
        LD      (RETADDR),HL    ;and save it
        POP     HL              ;Get and save PARAM2
        LD      (PARAM2),HL
        POP     HL              ;Get and save PARAM1
        LD      (PARAM1),HL
        .
        .
        .
        LD      HL,(PARAM2)     ;Move PARAM2
        LD      (LOCAL1),HL     ;to LOCAL1
        .
        .
        .
EXIT:   LD      HL,(RETADDR)    ;Get return address
        JP      (HL)
        .END
```

For assembly language functions (.FUNC's) the sequence is essentially the same, except that:

1) Two words of zeros are pushed by the compiler after any parameters are put on the stack.

2) After the stack has been completely cleaned up at the routine exit time, the .FUNC must push the function result on the stack.

Here is an example of an external assembly language procedure, and a program that uses it. This example takes a very primitive approach to interrupt handling (which might still be useful in some applications). There is no provision for handling interrupts from the device where a collected buffer is being written to disk. Support for continuous interupts would be more complex, involving multiple buffers and exclusion mechanisims to assure that buffer switching would occur reliably. The Project intends eventually to provide synchronization capabilities at the Pascal level, so that interrupt handling can be accomplished with greater convenience and safety.

```
.PROC                   DRCOLLECT,0     ; Name of routine for use by linker.
            .CONST      DRBUFLENG       ; Public constant.
            .PUBLIC     DRBUFFER        ; Public variable.

DRADDR      .EQU        167770
DRVECT      .EQU        140
            MOV         #HANDLR,@#DRVECT    ;Load address of interrupt
            MOV         #340,@#DRVECT+2     ;handler and set priority.
            MOV         #DRBUFLENG,R0       ;Load R0 with size of buffer.
            MOV         #DRBUFFER,R1        ;Load R1 with address of buffer.
            BIS         #100,@#DRADDR       ;Enable interrupts on DR interface.

LOOP:       TST         R0              ;Exit loop when buffer full.
            BNE         LOOP
            BIC         #100,@#DRADDR   ;Disable interrupts.
            RTS         PC              ;Return to PASCAL host program.

HANDLR:     MOV         @#DRADDR+2,(R1)+    ;Load buffer with next word,
            DEC         R0                 ;increment R1, decrement R0.
            RTI                            ;Return from interrupt.


PROGRAM COLLECTDATA;
  CONST
    DRBUFLENG = 256;

  TYPE
    DATABUFFER = Array [1..DRBUFLENG] of integer;

  VAR
    I: Integer;
    DRBUFFER: DATABUFFER;
    DATAFILE: File of DATABUFFER;

  PROCEDURE DRCOLLECT;
    External;

BEGIN (*of Collect Data*)
  Rewrite( DATAFILE, 'SAMPLE.DATA' );
   For I:=1 to 10 do
    BEGIN
     DRCOLLECT;
      DATAFILE^:=DRBUFFER;
       Put( DATAFILE );
    END;
   Close( DATAFILE, Lock );
END.
```

3.3.2.3 ASSEMBLY LANGUAGE TO ASSEMBLY LANGUAGE LINKAGES

The third way in which separate routines may share data structures and subroutines is by linkage from assembly language to assembly language. This is made possible through the use of the .DEF and .REF pseudo-ops provided in the UCSD assemblers. These generate link information that allows two separately assembled procedures to be L(inked together. One possible use for this will be the linking of separate routines and drivers in constructing new UCSD interpreters.

The following are very abbreviated versions of two assembly language routines which make separate references. They are used externally by the UNIT PSGRAPHICS:

The first routine declares three public variables and declares a .DEF for a label to be referenced by the second routine (Note that this is only a skeleton of the actual MOVETO routine):

```
.PROC   MOVETO,6   ; THE 3 REAL PARAMETERS OCCUPY 6 WORDS

; PROCEDURE MOVETO(X, Y, Z: REAL);
;
; COMPUTES A NEW PSXPOS & PSYPOS FROM PSMATP AND
; AN ASSUMED 1.0 AS THE INPUT VECTOR HOMOGENOUS
; COORDINATE...
;
; (X Y Z 1) dot PSMATP^ = (X' Y' Z' W')
; PSXPOS := X'/W';
; PSYPOS := Y'/W';

; THESE ARE GLOBALS IN THE PASCAL HOST
.PUBLIC PSXPOS
.PUBLIC PSYPOS
.PUBLIC PSMATP


; MOVETO ENTRY POINT

        MOV     R5,-(SP)        ; R5 USED AS FRAME POINTER
        MOV     SP,R5
        MOV     @#PSMATP,R0     ; R0 IS TOS MATRIX POINTER

; PARAMETER DISPLACEMENTS FROM R5 FRAME POINTER
X       .EQU    14
Y       .EQU    10
Z       .EQU    4
W       .EQU    -4
        ;
        ; COMPUTE W', HOMOGENEOUS COORD
        ; AND LEAVE IT ON STACK
        ;
        ;
```

```
        ; COMPUTE PSXPOS
        ;
        ; NOW COMPUTE PSYPOS
        ;
        ;
        ; CLEAN UP STACK AND RETURN
        ;

ROUND:  ; ROUND REAL ON STACK TO INTEGER
        ; IF < 0 THEN SUBTRACT 0.5 ELSE
        ; ADD 0.5, THEN TRUCATE.

   .END
```

The second routine references the first routine as well as the separately assembled DRAWLINE routine. MOVETO must be linked into LINETO before the routine can be linked in as an external procedure to a PASCAL UNIT or PROGRAM.

```
.PROC LINETO,6 ; PARAMETERS OCCUPY 6 WORDS

;   PROCEDURE LINETO(X, Y, Z: REAL);
;
;   DRAWS A LINE FROM THE LAST POINT CONTAINED IN
;   PSXPOS & PSYPOS TO THE NEW TRANSFORMED POINT
;   GIVEN BY X, Y, & Z...
;
;   SAVEX := PSXPOS; SAVEY := PSYPOS;
;   MOVETO(X, Y, Z);
;   DRAWLINE(JUNK, PSBUFP^, 20, 160+SAVEX, 120-SAVEY,
;                              PSXPOS-SAVEX, SAVEY-PSYPOS, 1);

.PUBLIC PSXPOS
.PUBLIC PSYPOS
.PUBLIC PSBUFP
.PRIVATE RANGE

.REF MOVETO
.REF DRAWLINE

; LINETO ENTRY POINT
```

```
        MOV     R5,-(SP)
        MOV     SP,R5               ; USE R5 AS STACK FRAME POINTER
SAVEX   .EQU    -2
SAVEY   .EQU    -4
X       .EQU    14
Y       .EQU    10
Z       .EQU    4
        ;
        ;   SAVEX := PSXPOS; SAVEY := PSYPOS;
        ;
        ;   MOVETO(X, Y, Z);
        ;
        JSR     PC,@#MOVETO
        ;
        ;   DRAWLINE(...);
        ;
        JSR     PC,@#DRAWLINE
        ;
        ; ALL DONE... RETURN
        ;
        JMP     @R0
.END
```

For examples and more information see section 1.9 ASSEM

-- Notes --

```
***************** *****************
* LONG INTEGERS * * SECTION 3.3.3 *
***************** *****************
```

Version I.5   September 1978

With the predeclared type INTEGER the optional use of a length
attribute constitutes a new type and will, in the remainder of this
document, be referred to as LONG INTEGER. The LONG INTEGER is
suitable for business, scientific or other applications which
need extended number lengths with complete accuracy. This
extension supports the four basic standard INTEGER arithmetic
operations (addition, subtraction, division and multiplication) as
well as routines facilitating conversion to strings and standard
INTEGERs. Strong type checking is enforced throughout in the Pascal
spirit. Input/Output, in line declaration of constants and inclusion
in structured types are all fully supported and are analogous to the
usage of standard INTEGERs.

LONG INTEGERs are declared using the standard identifier
INTEGER followed by a length attribute in square brackets. This
length is an unsigned number, not larger than 36, denoting the minimum
number of decimal digits representable by the LONG INTEGER. For
example, a variable called 'X' capable of storing at least an eight
decimal digit signed number would be created by:

VAR X: INTEGER[8];

Constants are defined in the normal manner:

CONST RYDBERG = 10973731;

In the above example RYDBERG would be by default a LONG INTEGER
and could be used anywhere a LONG INTEGER could be used.

In general LONG INTEGERs may be used anywhere it is
syntactically correct to use REALs, however care must be taken to
ensure that sufficient words have been allocated by the declared
length attribute for storage of the result of assignment or arithmetic
expression statements (see note in next subsection for complete
details). INTEGER expessions are implicitly converted as required
upon assignment to, or arithmetic operations with, a LONG INTEGER.
The reverse is not true.

Examples:

```
              VAR I: INTEGER;
                  L: INTEGERN; {where N is an integer constant
                                    <= 36 }
                  S: REAL;

              I:= L; {syntax error, see TRUNC(L) below}
              L:=-L; {correct, with the usual exception}
              L:= I; {always correct}
              L:= S; {never accepted}
              S:= L; {will be implemented with II.0}
```

Arithmetic operations which may be used in conjunction with LONG INTEGERs are any or all from the set {+,-,*,DIV,unary plus/minus}. On assignment the length of the LONG INTEGER is adjusted (during execution) to the declared length attribute of the variable, therefore overflow may result. Overflow occurs only when the intermediate result exceeds the number of words required to store (as a minimum) thirty-seven decimal digits, or when the final result is assigned to a variable with insufficient length attribute. A length attribute of 5 thru 9 may store up to and including 2147483647, length attributes of 10 thru 14 may store thru 140737488355327, 15 thru 18 .. 9223372036854775807. It is left to the interested reader to compute any larger length attribute storage capacities. This range of length attributes all having the same upper bound is a result of the allocation of a full word as the least amount of additional storage, i.e. 5 thru 9 represent a two word INTEGER.) All of the standard relational operators may be used with mixed LONG INTEGER and INTEGER.

The function TRUNC(L), where 'L' is a LONG INTEGER, will convert 'L' to an INTEGER (i.e. TRUNC will accept a LONG INTEGER as well as a REAL as an argument). Overflow will result if L is greater than MAXINT.

The procedure STR(L,S) converts the INTEGER or LONG INTEGER 'L', into a string (complete with minus sign if needed) and places it in the STRING 'S'. The following program segment will provide a suitable dollar and cent routine:

```
    STR(L,S); INSERT('.',S,LENGTH(S)-1); WRITELN(S);
```

Where 'L' and 'S' are appropriately declared. TRUNC and STR are the only two routines which currently will accept LONG INTEGERs as parameters. An attempt to declare a LONG INTEGER in a parameter list will result in a syntax error, which may be circumvented by creating a type which is a LONG INTEGER. For example:

```
TYPE LONG = INTEGER18;
PROCEDURE BIGNUMBER(BANKACCT: LONG);
```

The LONG INTEGER is stored as a multi-word, twos complement
binary number. System and interpreter routines do the I/O conversions
as required. Maximum storage efficiency is achieved by dynamic
expansion and contraction of word allocation as required. During
LONG INTEGER operations the length is placed on the stack above the
number itself, the declared length attribute need not be the same and
can be less than this length.

— Notes —

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

Version I.5   September 1978

The UCSD Pascal P-machine, designed specifically for the
execution of Pascal programs on small machines, is an extensively
modified descendant of the P-2 pseudo-machine from Zurich. It
supports variable addressing, including strings, byte arrays, packed
fields, and dynamic variables; logical, integer, real, and set top-of-
stack arithmetic and comparisons; multi-element structure comparisons;
several types of branches; procedure/function calls and returns,
including overlayable procedures; miscellaneous procedures used by
systems programs; and basic I/O subsystem.

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.

3.4.1 HARDWARE (emulation)

The P-machine uses 16-bit words, with two 8-bit bytes per
word. It has several registers and a user memory, in which are kept a
stack and a heap. All registers are pointers to word-aligned
structures, except IPC, which is a pointer to byte-aligned
instructions. The registers are:

SP: Stack Pointer is a pointer to the top of the execution stack. The
      stack starts in high memory and grows toward low memory. It
      contains code segments and activation records, and is used to pass
      parameters, return function values, and as an operand source for
      many instructions. The stack is extended by loads and procedure
      calls, and is cut back by stores, procedure returns, and arithmetic
      operations.

NP: New Pointer is a pointer to the top of the dynamic heap. The heap
      starts in low memory and grows upward toward the stack. It
      contains all dynamic variables (see Jensen and Wirth, Chapter 10).
      It is extended by the standard procedure 'new', and is cut back by
      the standard procedure 'release'.

JTAB: Jump TABle pointer is a pointer to the procedure attribute table
      of the currently executing procedure. (See Section 3.5, figure 5.)

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.

3.4.2   OPERATING SYSTEM/P-MACHINE COMMUNICATION - SYSCOM

        It is sometimes necessary for the operating system and the P-
machine to exchange information. Hence there exists a variable SYSCOM
in the outer block of the operating system, and a corresponding area in
memory known to the hardware. The fields in SYSCOM actually relevant
to this communication are:

IORSLT: contains the error code returned by the last activated or
      terminated I/O operations. (See I/O section below, and operating
      system read and write procedures.)

XEQERR: contains the error code of the last run-time error. (See
      exception handling below.)

SYSUNIT: contains the unit number of the device the operating system
      was booted from (usually 4 or 5).

BUGSTATE: contains the current bugstate. (See BPT instruction below.)

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.

3.4.3   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^.BCMBP 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^.BCMIPC.
4) points IPC to a CXP 0,2 (call operating system procedure EXECERROR) instruction.
5) resumes execution of interpreter code, starting with the CXP.

3.4.4   OPERAND FORMATS

Although an element of a structure may occupy as little as one bit, as in a PACKED ARRAY OF boolean, variables in the P-machine are always aligned on word boundaries. All top-of-stack operations expect their operands to occupy at least one word, even if not all the information in a word is valid. The least significant bit of a word is bit 0, the most significant is bit 15.

BOOLEAN: One word. Bit 0 indicates the value (false=0, true=1), and this is the only information used by boolean comparisons. However, the boolean operators LAND, LOR, and LNOT operate on all 16 bits.

INTEGER: One word, two's complement, capable of representing values in the range -32768..32767.

SCALAR (user-defined):   One word, in range 0..32767.

CHAR: One word, with low byte containing character. The internal character set is "extended" ASCII, with 0..127 representing the standard ASCII set, and 128..255 as a user-defined character set.

REAL: Two words, with format implementation dependent. The system
is arranged so that only the interpreter needs to know the detailed
internal format of REALs (beyond the fact that they occupy two
words) Following are the two detailed formats for the CPUs we now
(as of I.4) support.

PDP11:

```
        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           !
        ----------------------------------------------------
```
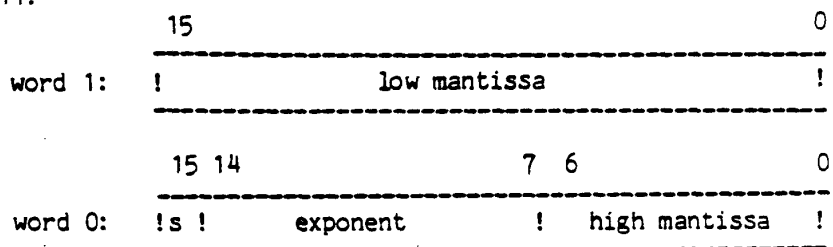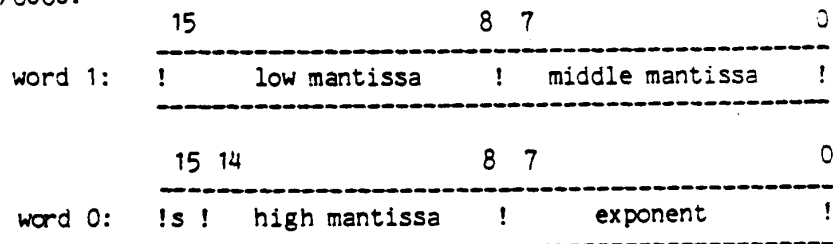
Both representations have an excess-128 exponent, a fractional
mantissa that is always normalized, exponent base 2, an implicit
24th mantissa bit, and zero represented by a zero exponent. (See
PDP11 processor manual or Z80/8080 interpreter listing for greater
detail.)

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 number of words, all of which
contain valid information. All elements past the last word of a
set are assumed not to be elements of the set. Before being stored
back in the data segment, a set must be forced back to the size
allocated to it, and so an ADJ instruction must be issued.

RECORDS and ARRAYS: any number of words (up to 16384 words in one
    dimension). Arrays are stored in row-major order, and always have
    a lower index of zero. Only fields or elements are loaded onto the
    stack - never the structure itself. Packed arrays must have an
    integral number of elements in each word, as there is no packing
    across word boundaries (it is acceptable to have unused bits in
  . each word). The first element in each word has bit 0 as its low-
    order bit.

STRINGS: 1..128 words. Strings are a flexible version of packed
    arrays of char. A string[n] occupies (n div 2)+1 words. Byte 0
    of a string is the current length of the string, and bytes
    1..length(string) contain valid characters.

CONSTANTS: constant scalars, sets, and strings may be imbedded in
    the instruction stream, in which case they have special formats.
All scalars (excluding reals) not in the range 0..127: two bytes,
    low byte first.
Strings: all string literals take length(literal)+1 bytes, and
    are byte aligned. The first byte is the length, the rest are the
    actual characters. This format applies even if the literal should
    be interpreted as a packed array of char (see S1P and S2P
    below).
Reals and sets: word aligned, and in reverse word order.

3.4.5   INSTRUCTION SET FORMAT

    Instructions on the P-machine are one or two bytes long, followed
by zero to four parameters. Most parameters specify one word of
information, and are one of five basic types.

UB unsigned byte: high order byte of parameter is implicitly zero.
SB signed byte: high order byte is sign extension of bit 7.
DB don't care byte: can be treated as SB or UB, as value is always in
        the range 0..127.
B   big: this parameter is one byte long when used to represent values in
        the range 0..127, and is two bytes long when representing
        values in the range 128..32767. If the first byte is in
        0..127, the high byte of the parameter is implicitly zero.
        Otherwise, bit 7 of the first byte is cleared and it is used as the
        high order byte of the parameter. The second byte is used
        as the low order byte.
W   word: the next two bytes, low byte first, is the parameter value.

Any exceptions to these formats are noted in the instructions where they occur.

3.4.6 ENGLISH INSTRUCTION SET DESCRIPTION

In the following section, references to an element on the stack are context-dependent, and can mean anywhere from one word to 256 words. Also, unless specifically noted to the contrary, operands are popped off the stack - they are not left around.

Abbreviations are used widely, but use fairly simple conventions. Parameters are written as X or X_n, where X is UB, SB, DB, B, or W, and n is an integer indicating the parameter position in the instruction. Tos means the operand on the top of stack, tos-1 the next operand, etc. Mark Stack Control Word is abbreviated to MSCW.

Many instructions refer to the activation record of a procedure, and this document assumes the reader has a general knowledge of procedure calling in stack machines, and the concept of stack frames. An activation record as defined in this document specifically consists of:
  1) the local data segment of the procedure, and
  2) the MSCW, containing addressing information (static links), and information on the calling procedures environment when the procedure was called.
  (See Section 3.5, figure 7.)

The dynamic chain refers to the calling chain, traversed using the MSCW.MSDYN links. The static chain refers to the lexical or ancestor chain, traversed using the MSCW.MSSTAT links.

MNEMONIC   OP-CODE   PARAMETERS   FULL NAME AND OPERATION

5.A VARIABLE FETCHING, INDEXING, STORING, AND TRANSFERING
5.A.1 ONE WORD LOADS AND STORES

5.A.1.a CONSTANT ONE WORD LOADS

SLDC   0..127   Short load word constant. Pushes the
                                    opcode, with high byte zero, onto stack.

LDCN   159   Load constant nil. Pushes the
                            implementation-dependent value of nil.

LDCI   199   W   Load constant word. Pushes W.

5.A.1.b LOCAL ONE WORD LOADS AND STORE

SLDL1   216   Short load local word. SLDLx fetches
..   ..   the word with offset x in MP activation
SLDL16   231   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.

5.A.1.c GLOBAL ONE WORD LOADS AND STORE

SLDO1   232   Short load global word. SLDOx fetches
..   ..   the word with offset x in BASE activation
SLDO16 247   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.

5.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.

5.A.1.e INDIRECT ONE-WORD LOADS AND STORE

STO   154   Store indirect. Tos is stored into the
word pointed to by tos-1.

SINDO   248   Load indirect.

5.A.2 MULTIPLE WORD LOADS AND STORES (SETS AND REALS)

LDC   179   UB,<block> Load multiple word constant. UB is the
                             number of words to load, and <block> is a
                             word aligned block of UB words, in reverse
                             word order. Load the block onto the stack.

LDM   168   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.

5.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.

5.A.4 STRINGS

LCA   166   UB,<chars> Load constant string address. Push a
                             byte pointer to the location UB is contained
                             in, and skip IPC past <chars>.
                                  in, and skip IPC past <chars>

SAS	170	UB	String assign. Tos is either a source byte pointer or a character. (Characters always have a high byte of zero, while pointer never do.) Tos-1 is a destination byte pointer. UB is the declared size of the destination string. If the declared size is less than the current size of the source string, a run-time error occurs; otherwise all bytes of source containing valid information are transferred to the destination string.
S1P	208		String to packed conversion on tos. Tos is a byte pointer to a string, and is incremented by one byte in order to point to the first character of the string.
S2P	157		String to packed conversion on tos-1. Tos and tos-1 are byte pointers, and tos-1 is incremented by one byte.
IXS	155		Index string array. Performs the same operation as IXB, except before indexing the index is checked to see if it is in the range 1..current length. If not, a run-time error occurs.

5.A.5 RECORD AND ARRAY INDEXING AND ASSIGNMENT

MOV	168	B	Move words. Tos is a source pointer to a block of B words, tos-1 is a destination pointer to a similiar block. Transfer the block from the source to the destination.
SINDO .. 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.

5.A.6 DYNAMIC VARIABLE ALLOCATION AND DE-ALLOCATION

NEW   158 1   New variable allocation. Tos is the size (in words) to allocate the variable, and tos-2 is a word pointer to a dynamic variable. If GDIRP is non-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.
                                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.

5.B TOP OF STACK ARITHMETIC AND COMPARISONS

5.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	Boolean =,
NEQBOOL	183	6	<>,
LEQBOOL	180	6	<=,
LESBOOL	181	6	<,
GEQBOOL	176	6	>=,
GTRBOOL	177	6	and > comparisons.

Compare bit 0 of tos-1 to bit_0 of tos and push true or false.

5.B.2 INTEGER

ABI	128	Absolute value of integer. Take absolute value of integer tos. Result is undefined if tos is initially -32768.
ADI	130	Add integers. Add tos and tos-1.
NGI	145	Negate integer. Take the two's complement of tos.
SBI	149	Subtract integers. Subtract tos from tos-1.
MPI	143	Multiply integers. Multiply tos and tos-1. This instruction may cause overflow if result is larger than 16 bits.
SQI	152	Square integer. Square tos. May cause overflow.
DVI	134	Divide integers. Divide tos-1 by tos and push quotient. (PDP 11 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.

5.B.3 REALS

All over/underflows cause a run-time error.

FLT	138	Float top-of-stack. The integer tos is converted to a floating point number.
FLO	137	Float next to top-of-stack. Tos is a real, tos-1 is an integer. Convert tos-1 to a real number.
TNC	158 22	Truncate real. The real tos is truncated (as defined in Jensen and Wirth) and converted to an integer.
RND	158 23	Round real. The real tos is rounded (as defined in Jensen and Wirth), then truncated and converted to an integer.
ABR	129	Add reals. Take the absolute value of the real tos.
ADR	131	Add reals. Add tos and tos-1.
NGR	146	Negate real. Negate the real tos.
SBR	150	Subtract reals. Subtract tos from tos-1.
MPR	144	Multiply reals. Multiply tos and tos-1.
SQR	153	Square real.
DVR	135	Divide reals. Divide tos-1 by tos.
POT	158 35	Power of ten. The integer tos is checked for 0 <= 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 =,
NEQREAL	183 2	<>,
LEQREAL	180 2	<=,

```
LESREAL    181  2                                    <,
GEQREAL    176  2                                        >=,
GTRREAL    177  2                                                and > comparisons.
                                    Push TRUE or FALSE.
```

5.B.4 SETS

```
ADJ       160      UB          Adjust set.  The set tos is forced to
                          occupy UB words, either by expansion (putting
                          zeroes "between" tos and tos-1) or
                          compression (chopping of high words of set),
                          and its length word is discarded.


SGS       151                  Build a singleton set.  The integer tos
                          is checked to insure that 0 <= 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 =,
NEQPOWR   183  8                   <>,
LEQPOWR   180  8                       <= (subset of),
GEQPOWR   176  8                                    and >=
                          (superset of) comparisons.
```

5.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.

5.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.

5.B.7 ARRAY AND RECORD COMPARISONS

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

5.C   JUMPS

        Simple (non-case statement) jumps are all two bytes long. The
first byte is the op-code, the second is a SB jump offset. If this
offset is non-negative, it is simply added to IPC. (A value of zero
for the jump offset will make any jump a two-byte nop.) If SB is
negative, then SB div 2 is used as a word offset into JTAB, and IPC
is set to the byte address(JTAB^[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].

5.D PROCEDURE AND FUNCTION CALLS AND RETURNS

The general scheme used in procedure/function invocation is

1) Calculate the data_size and parameter_size of the called
procedure by using the information in the current procedure
dictionary (pointed to by SEG).
2) Extend stack by data_size bytes.
3) Copy parameter_size bytes from the old top-of-stack to the
beginning of the space just allocated.
4) Build a MSCW, saving SP, IPC, SEG, JTAB, MP, and a pointer
to the most recent activation record of the called procedure's
immediate parent.
5) Calculate new values for SP, IPC, JTAB, MP, and if necessary,
SEG. Check for stack overflow.
6) If the called procedure has a lex level of -1 or 0 save BASE
and calculate a new BASE.

CLP	206	UB	Call local procedure. Call procedure UB, which is an immediate child of the currently executing procedure and in the same segment. Static link of MSCW is set to old MP.
CGP	207	UB	Call global procedure. Call procedure UB, which is at lex level 1 and in same segment. The static link of the MSCW is set to BASE.
CIP	174	UB	Call intermediate procedure. Call procedure UB in same segment as the currently executing procedure. The static link of the MSCW is set by looking up the call chain until an activation record is found whose caller had a lex level one l 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 any 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.

CSP   158

Scan this document for op of 158.

RNP   173   DB

Return from non-base procedure. DB is the number of words that should be returned as a function value (0 for procedures, 1 for non-real functions, and 2 for real functions). DB words are copied from the bottom of the data segment and "pushed" onto the caller's top-of-stack. The information in the MSCW is then used to restore the caller's correct environment.

RBP   193   DB

Return from base procedure. The saved base is moved into BASE, after which things proceed as in the RNP instruction.

EXIT   158   4

Exit from procedure. Tos is the procedure number, tos-1 is the segment number. This operator sets IPC to point to the exit code of the currently executing procedure, then sees if the current procedure is the one to exit from. If it

is, control returns to the instruction
fetch loop.
    Otherwise, each MSCW has its saved IPC
changed to point to the exit code of the
procedure that invoked it, until the
desired procedure is found.
    If at any time the saved IPC of main body
of the operating system is about to be
changed, a run-time error occurs.

5.E   SYSTEMS PROGRAMS SUPPORT PROCEDURES

    See Section 2.1 for description of these procedures.

    BYTE ARRAY PROCEDURES

FLC   158 10   Fillchar(dst, len, char).

SCN   158 11   Scan(maxdisp, start, forpast, char, mask).

MVL   158 02   Moveleft(src, dst, numbytes).

MVR   158 03   Moveright(src, dst, numbytes).

    COMPILER PROCEDURES (still undocumented)

TRS   158 08   Treesearch.

IDS   158 07   Idsearch.

    DEBUGGER

BPT   213   Breakpoint (conditional HALT)

    MISCELLANEOUS

TIM   158 09   Time.

XIT   214   Exit.

NOP   215   No operation.

-- Notes --

Version II.0   February 1979

UCSD uses an interpreter based implementation of Pascal. This
means that the compiler emits code for a pseudo-machine which is
emulated at run time by a program written in the machine language of
the host. The compiler, program editor, stand-alone operating system,
and various utilities are themselves written in Pascal and run on the
same interpreter. Thus the entire system can be moved to a new host
machine by rewriting the interpreter for the new host. This document
describes the Pseudo-machine codefiles as they were in version I.3.
Many of the segments mentioned are no longer resident in the codefile
used as an example. This does not affect the functionality of the
description of the mechanisims put forth by this document.

Figure 3.5.10 (the last page of this document) is a skeleton
version of a large Pascal program, here-in-after referred to as "The
Program". This document is a top-down description of the realization
of that program on the UCSD Pascal system. We will make occasional
use of a helpful coincidence: The Program is the framework of the
portion of the UCSD Pascal environment that's written in Pascal.

If The Program were expanded to a complete Pascal system, it
would consist of at least 6000 lines of Pascal and compile to more
than 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). We have
therefore extended Pascal so that a programmer can explicitly
partition a program into segments; only some of which need be
resident in main memory at a time. The syntax of this extension is
shown in figure 3.5.1. (Any syntactic objects not defined explicitly
there retain their standard interpretation as defined by Jensen &
Wirth: Pascal User Manual and Report.) See Section 5.9 for revised
syntax diagrams.

<program> ::= <program heading> .

::= <label declaration part>
     <constant declaration part> <type definition part>
     <variable declaration part>

::= SEGMENT <procedure heading>
     ; \ SEGMENT <function heading>
     ;

::= <procedure and function declaration part>
     <statement part>

FIGURE 3.5.1. SEGMENT DECLARATION SYNTAX.

Segment declaration syntax (figure 3.5.1) requires that all nested segments be declared before the ordinary procedures or functions of the segment body. Thus, a code segment can be completely generated before processing of code for the next segment starts. This is not a functional limitation, since forward declarations can be used to allow nested segments (COMPILER in The Program) to reference procedures in an outer segment body (CLEARSCREEN). Similarly, segment procedures and functions can themselves be declared forward.

Segmenting a program does not change its meaning in any fundamental sense. When a segment is called (e.g. the COMPILER segment in line A), the interpreter checks to see if it is present in memory due to a previous invocation. If it is, control is transferred and execution proceeds: if not, the appropriate code segment must be loaded from disk before the transfer of control takes place. When no more active invocations of the segment exist, its code is removed from memory. For instance, in The Program, the code for the COMPINIT segment is not present in memory either before or after the execution of line A. Clearly, a program should be segmented in such a way that (non-recursive) segment calls are infrequent; otherwise, much time could be lost in unproductive thrashing (particularly on a system with low performance disk).

high address

DEBUGGER	10	
FILER	17	
EDITOR	12	
COMPINIT	7	
COMPILER	41	
INITIALIZE	3	
USER PROGRAM	1	
PASCALSYSTEM	17	
SEGMENT DICITONARY	1	

not shown in the program

low address

FIGURE 3.5.2. PASCAL SYSTEM CODE FILE..

The code file resulting from compilation of The Program is diagrammed in figure 3.5.2*. The file is a sequence of code segments preceded by a segment dictionary. The size of each segment is noted in blocks, the 512-byte disk allocation quantum used on most PDP-11 operating systems. The sizes indicated are representative of a full Pascal system. Each code segment begins on a block boundary. The ordering (from low address to high address) is determined by the order that one encounters segment procedure bodies in passing through The Program.

* An overview of the relationship between figures 3.5.2 through 3.5.8 (to be discussed in the following pages) is given in figure 3.5.9 at the end of this section. It is helpful to study figure 3.5.9 at this point for a better understanding of the section.

The segment dictionary in the first block of a code file contains an entry for each code segment in the file. The entry includes the disk location and size(in bytes) for the segment. The disk location is given as relative to the beginning of the segment dictionary (which is also the beginning of the code file) and is given in number of blocks. This information is kept in the system communications area (also called SYSCOM) during the execution of the code file, and is used in the loading of non-present segments when they are needed. Figure 3.5.3 details the layout of the table and shows representative contents for the Pascal system code file.

```
location  |          1          |
          |----------------------|  PASCALSYSTEM
size      |         8500         |
          |----------------------|
          |          18         |
          |----------------------|  USERPROGRAM
          |       variable       |
          |----------------------|
          |          22         |
          |----------------------|  COMPILER
          |        20932         |
          |----------------------|
          |          63         |
          |----------------------|  COMPINIT
          |         3480         |
          |----------------------|
          |          70         |
          |----------------------|  DEBUGGER
          |         5880         |
          |----------------------|
          |                      |
```

FIGURE 3.5.3. THE SEGMENT DICTIONARY

A code segment contains the code for the body of each of its procedures, including the segment procedure,itself. Figure 3.5.4 is a detailed diagram of the code segment of The Program (Pascalsystem). Each of a code segment's procedures are assigned a procedure number, starting at 1 for the segment procedure, and ranging as high as 255 (current temporary limit of 127). All references to a procedure are made via its number. Translation from procedure number to location in the code segment is accomplished with the procedure dictionary at the end of the segment. This dictionary is an array indexed by the procedure number. Each array element is a self-relative pointer to the code for the corresponding procedure. Since zero is not a valid procedure number, the zero'th entry of the dictionary is used to store the segment number (even byte) and number of procedures (odd byte). Observe that CLEARSCREEN is the first procedure for which code is generated and that it appears at the beginning of the segment. The outer block code is generated and appears last.
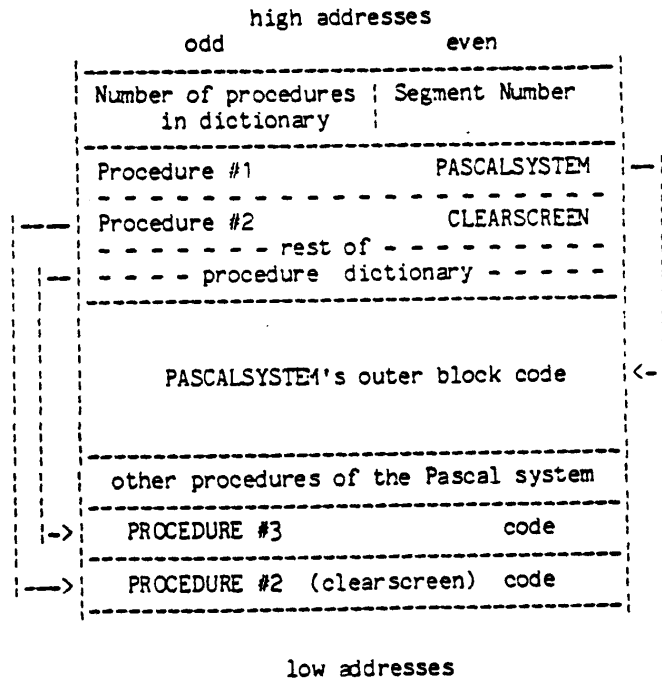
FIGURE 3.5.4. A CODE SEGMENT

A more detailed diagram of a single procedure code section is seen in figure 3.5.5. It consists of two parts: the procedure code itself in the lower portion of the section) and a table of attributes of the procedure. These attributes are:

LEX LEVEL:This odd byte is the depth of absolute lexical nesting for the procedure. (i.e. Lex Level (LL) Pascalsystem=-1, LL COMPILER or CLEARSCREEN=0, LL COMPINIT=1, etc.).

PROCEDURE NUMBER:This even byte refers to the number given in the procedure dictionary of the parent segment procedure. For example, the Procnum of CLEARSCREEN is 2. (see figure 3.5.4).

ENTER IC:This is a self-relative pointer to the first instruction to be executed for this procedure.

EXIT IC:This is a self-relative pointer to the beginning of the block of procedure instructions which must be executed to terminate procedure properly.

PARAMETER SIZE:The param size is the number of bytes of parameters passed to a procedure from its caller.

and DATA SEGMENT SIZE:The data size is the size of the data segment (See below) in bytes, excluding the markstack and PARAM SIZE.

Between these attributes and the procedure code there may be an optional section of memory called the "jump table". Its entries are addresses within the procedure code. JTAB is a term commonly applied to the six attributes just discussed and the jump table itself.

```
                    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
```
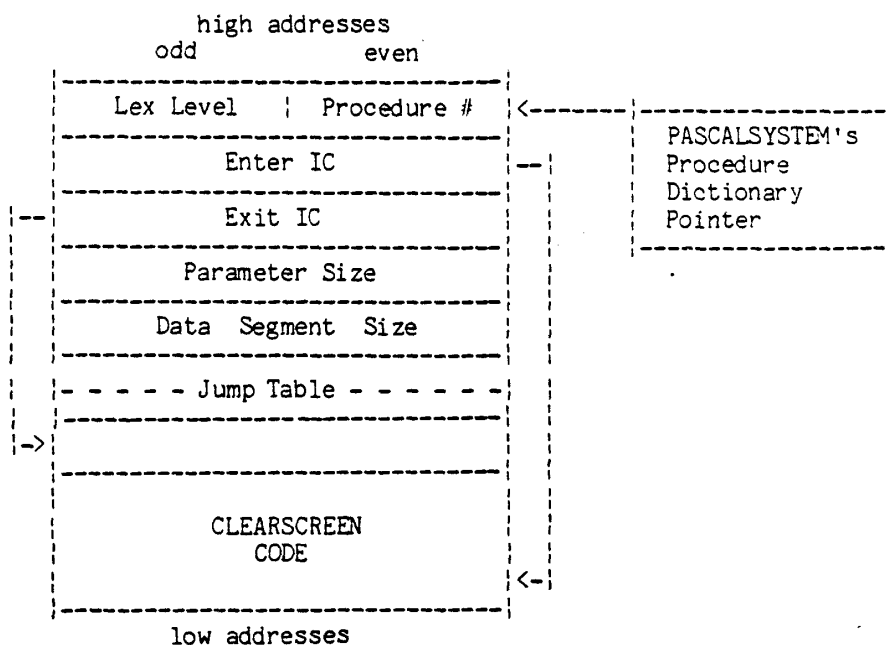
FIGURE 3.5.5. PROCEDURE CODE SECTION (OF CLEARSCREEN)

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
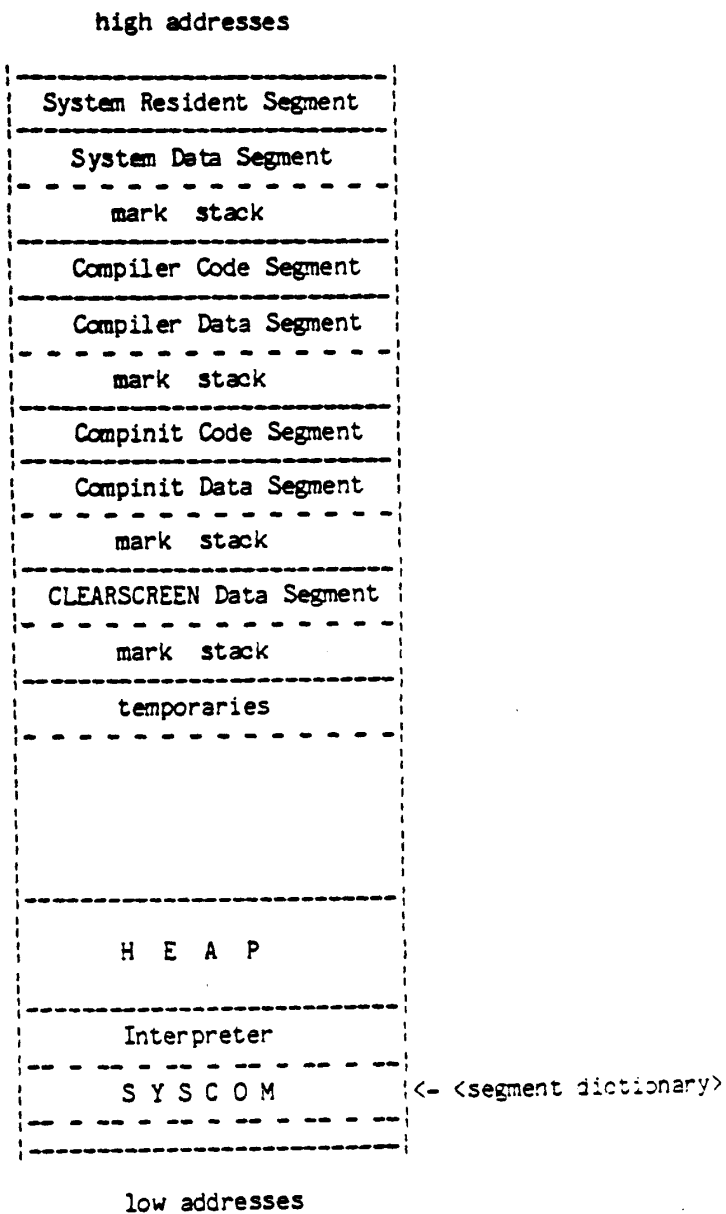
FIGURE 3.5.6. SYSTEM MEMORY DURING CLEARSCREEN EXECUTION

Figure 3.5.6 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 3.5.6)

Consider the status of operations just before COMPINIT is called in line B. Conceptually, there are six pseudo-variables which point to locations in memory:

a STACK POINTER(SP):which points to the current top of the stack,

a MARK STACK POINTER(MP):which points to the "topmost" markstack in the stack,(remember that the the stack grows down!),

a SEGMENT(SEG) variable:which points to the base of the procedure dictionary for the currently active segment procedure. For example, just before COMPINIT is called, SEG points to the COMPILER segment's procedure dictionary,

an INTERPRETER PROGRAM COUNTER(IPC):which contains the address of the next instruction to be executed in the code segment of the current procedure,

a JTAB pointer:which points to the collection of procedure attributes and jump table entries in the body of the current procedure code section,

and a NEW POINTER(NP):which points to the current top of the heap.

When segment procedure COMPINIT is called in line B, its code segment (including all compiler initialization procedures) is loaded on the stack. The COMPINIT data segment is built on top of the stack. Figure 3.5.7 is a diagram of the data segment for COMPINIT.

high addresses

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

low addresses
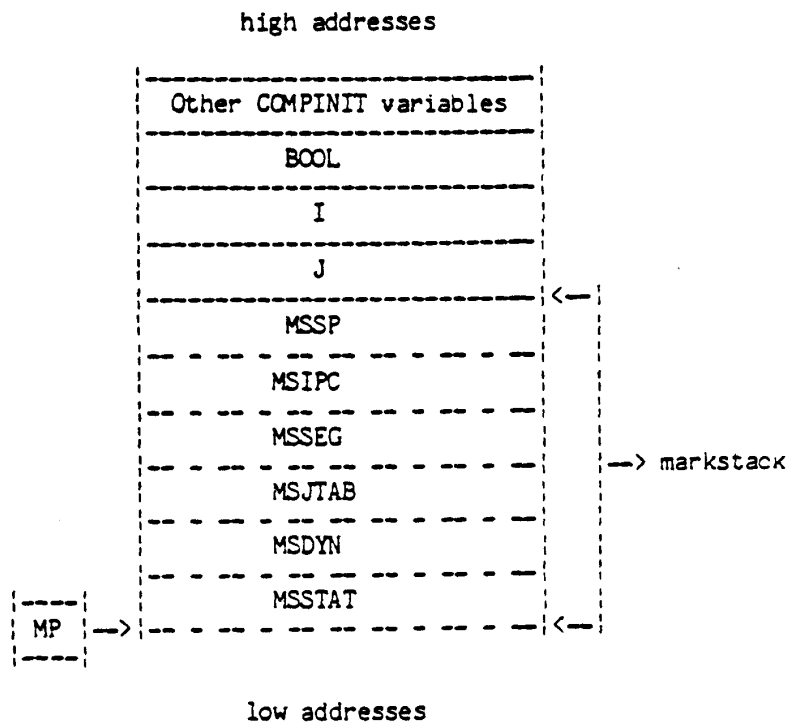
FIGURE 3.5.7. A DATA SEGMENT

In the upper portion of the data segment, space is allocated for variables local to the new procedure. For example,COMPINIT's data segment allocates space for integer variables I and J, as well as boolean BOOL.

In the lower portion of the data segment is a "markstack". When a call to any procedure is made, the current values of the pseudo-variables, which characterize the operating environment of the calling procedure, are stored in the markstack of the called procedure. This is so that the pseudo-variables may be restored to pre-call conditions when control is returned to the calling procedure.

For example, the call to COMPINIT causes conditions in COMPILER just before the call to be stored in COMPINIT's markstack in the following manner:

```
MarkStack DYNamic link (MSDYN) <-- MP
   "      "     IPC(MSIPC) <-- IC
   "      "     SEGment Pointer(MSSEG) <-- SEG
   "      "     Jump TABle (MSJTAB) <-- JTAB
   "      "     Stack Pointer (SP) <-- SP
```

In addition a Static Link field becomes a pointer to the data segment of the lexical parent of the called procedure. In particular, it points to the Static Link field of parent's markstack. After the building of the data segment new values for IC, SEG, SP, MP, and JTAB are established for the new procedure.

When the call to CLEARSCREEN is made on line C, another data segment is added to the stack and again the pseudo-variables are stored in the new markstack, as well as the appropriate Static Link, and updated. Note that now the SEG no longer points to the COMPINIT procedure dictionary, but to the Pascalsystem dictionary.

No code segment for CLEARSCREEN is added to the stack before the data segment since the code for CLEARSCREEN is already present in segment Pascalsystem. Its invocation causes only a data segment to be added to the stack. When CLEARSCREEN and INIT are completed, the COMPILER data segment will again be the top element on the stack.

Figure 3.5.8 is a detailed diagram of the stack during execution of an instruction in CLEARSCREEN, including appropriate pointers for static, dynamic, etc. links of CLEARSCREEN's markstack. Note where the pseudo-variables point in the stack. In particular, JTAB points inside CLEARSCREEN code section which is in the Pascalsystem code segment, IC points inside that CLEARSCREEN code, and SEG points to the base of the Pascalsystem code segment.

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               : top of stack
                :- - - - - - - - - - - - - - -:<——: SP :
                :                               :
                :- - - - - - - - - - - - - - -:<——: NP :
                :       H E A P                 : top of heap
                :————————————————————————:

                        low addresses
```
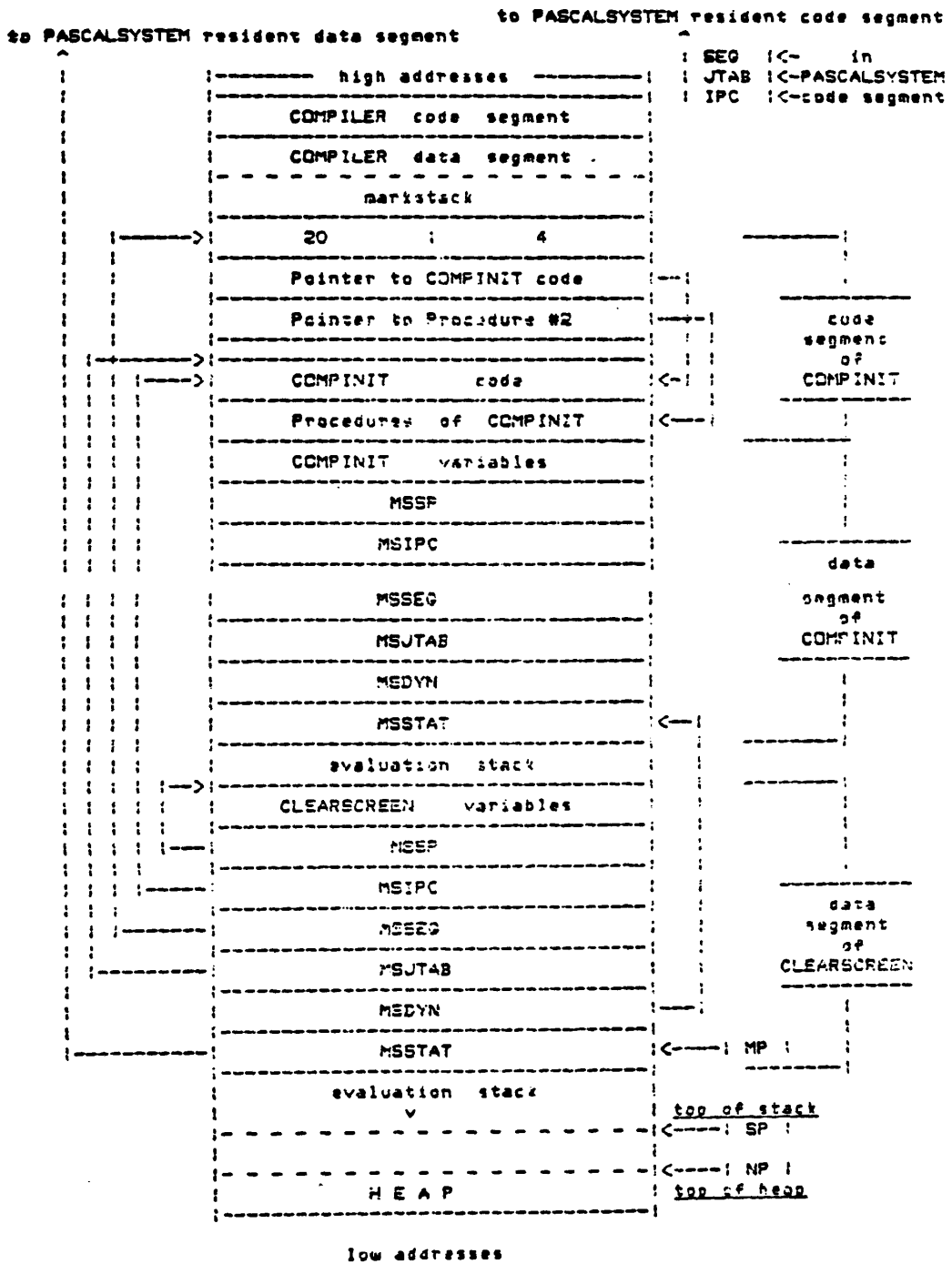
FIGURE 3.3.8   THE STACK DURING CLEARSCREEN

Figure 3.5.9 illustrates a top-down process by showing the relationships among diagrams 2 through 7.

```
            code file
            figure 3.5.2
      !---------------!
      ! PASCALSYSTEM  !--->! figure 3.5.4!
      !---------------!    !             !
      !               !    ! CLEARSCREEN !--->! figure 3.5.5!
      !               !    ! code detail !    ! proc. code  !
      !---------------!    !             !    ! detail      !
      !   segment     !    !             !    !             !
      ! dictionary    !--->!      figure 3.5.3            !
      !---------------!    ! segment dictionary detail    !


            system memory                                    !

              figure 3.5.8
      !  code segment !--->! figure 3.5.4 !
      !---------------!    !              !
      !               !    !              !
      !---------------!    !              !
      !   COMPINIT    !
      ! data segment  !--->!  figure 3.5.7            !
      !---------------!    ! data segment detail      !
```
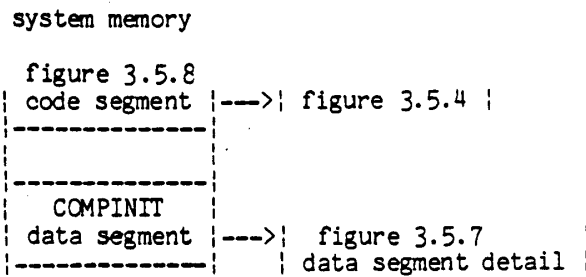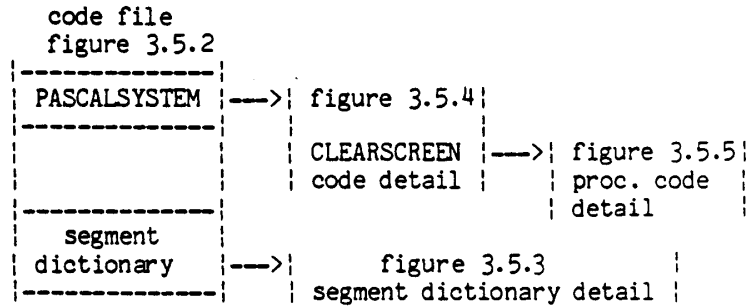
FIGURE 3.5.9. RELATIONSHIP OF DOCUMENT FIGURES

FIGURE 3.5.10. 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.
```

— Notes —

Version II.0 February 1979

Byte-swapping problems occur when code generated on one machine is transferred to another or programs which directly interface with memory (e.g. the Patch utility) are written on or for one machine and transferred to another which has a different ordering for its memory.

There are two different ways to order bytes in a given memory:

A) Byte Zero is the byte containing the least significant half of the word. Byte One contains the most significant half.

B) Byte Zero is the byte containing the most significant half of the word. Byte One contains the least significant half.

The difference between these is the way Byte quantities are read and stored in memory. Word quantities, such as integers, will be read and looked at in the same way on both types of machines. However, byte quantities such as P-code or characters will be reversed within each word.

An example:

```
DEFINITION         (A)                         (B)
              ls*            ms*         ms*            ls*
              ------------------         ------------------
VALUE(Hex)    !  04  !  07  !            !  07  !  04  !
              ------------------         ------------------
BYTE              0      1                   0      1
```

(least/most significant bit, thereby least/most significant byte)

If both of the words shown above were read as an integer , a word quantity, they would give the value 3,588. However, if the value of byte Zero was wanted (as in: C: PACKED ARRAY[0..1] OF CHAR;) then Definition A would show a value of 04H and Definition B would show a value of 07H. Both definitions would show the value 07H if the most significant byte were specified.

Byte-swapping is not a hard problem to solve, it just requires a little thought. The Patch utility has type declarations for both types of machines and a study of it should suffice to show how to satisfy your programming needs.

-- Notes --

```
******************  **************
*  O  O  P  S   *  * Section 4.1 *
******************  **************
```

Out Of Place Section

— Notes —

```
********************** **************
* LIBRARIAN UTILITY * * Section 4.2 *
********************** **************
```

Version II.0   February 1979

        LIBRARY.CODE is a utility program that allows the user to link
separately compiled PASCAL units and separately assembled subroutines
into a LIBRARY file. It is based upon the original pre-I.5 utility
LINKER.CODE and operates in basically the same way.

        To add a segment to *SYSTEM.LIBRARY it is necessary to create a
new file into which each segment that is wanted from the original
*SYSTEM.LIBRARY is first linked. It is then possible to add segments
by linking from another code file into the new file being created.

EXAMPLE

        Consider the case of adding a segment called TURTLE to the
already existing file *SYSTEM.LIBRARY which is assumed to contain the
segments PSGRAPHICS and MOVETO.

        On executing LIBRARY.CODE, the user is prompted for the name of
the output codefile. For this example, respond with the name
NEW.LIBRARY. The program now asks for a 'Link Code File'. The
response here is *SYSTEM.LIBRARY. The names of all segments currently
linked into the input library, i.e. *SYSTEM.LIBRARY, as well as their
length in bytes is now displayed. Currently there are a maximum of 16
segments in any PASCAL program or LIBRARY.

0- MOVETO	2398	4-		0	8-	0	10-	0
1- PSGRAPHI	864	5-		0	9-	0	11-	0
2-	0	6-		0	10-	0	14-	0
3-	0	7-		0	11-	0	15-	0

The following promptline appears:

  Segment # to link and <space>, N(ew file, Q(uit, A(bort

        The user now enters the number of a segment within the link
code file that is to be linked into the new library file, followed by
<space>. Next, the number of the segment in the output file to be
linked into (i.e. NEW.LIBRARY) is typed followed by <space>. For each
segment linked the librarian reads that segment from the input file and
writes it to the output file at the segment requested. It then
displays the segment table for the current state of the output library
file. In this example, respond with the following:

```
0<space>
Seg to link into? 0<space>
1<space>
Seg to link into? 1<space>
```

When all needed segments have been linked a new input file is requested by typing 'N' for N(ew file. In this example, a separately compiled PASCAL UNIT called TURTLE is assumed to exist in a codefile called TGRAPHICS.CODE. See section 3.2, UNITS. On entering the name of this file the following display appears:

0-		0	4-		0	8-		0	10-		C
1-		0	5-		0	9-		0	11-		0
2-		0	6- ·		0	10-	TURTLE	230	14-		0
3-		0	7-		0	11-		0	15-		C

The Unit TURTLE occurs in segment 10 and is to be linked into segment 2 within NEW.LIBRARY. The user responds:

```
10<space>
Seg to link into? 2<space>
```

The final display of the output library segment table is thus:

0- MOVETO	2398	4-	0	8-	0	10-	0
1- PSGRAPHI	864	5-	0	9-	0	11-	0
2- TURTLE	230	6-	0	10-	0	14-	0
3-	0	7-	0	11-	0	15-	0

The output library codefile length is displayed and in this example is 16 (blocks long).

Once the needed segments from all input files have been linked in the user locks the output file by typing 'Q' followed by a return, (unless a copyright notice is desired within the codefile). Type 'A' to abort the linking process. The old *SYSTEM.LIBRARY should either be removed or its name changed if it resides upon the same disk and the name NEW.LIBRARY must be changed to *SYSTEM.LIBRARY in order to be used.

NOTE

   In response to the initial prompt "Output Code File ->" we
could have just as easily said *SYSTEM.LIBRARY followed by another
*SYSTEM.LIBRARY in response to the prompt "Link Code File ->".
However, in this case the original *SYSTEM.LIBRARY will be removed
automatically upon completion of the linking process. Typing just *
is a sufficient abbreviation for *SYSTEM.LIBRARY.

— Notes —

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

Version II.0 March   1979

        The UCSD Pascal Operating System keeps certain information
about the user in a file called SYSTEM.MISCINFO. During each system
initialization this file is read into memory, and from there it is
accessed by many parts of the system, particularly (if the user has a
terminal suitable for it) by the screen oriented editor.

        Much of this information needs to be initially set up by the
user to conform to his particular hardware configuration or his taste
or convenience. Most of this information concerns the nature of his
terminal and keyboard, although there are a few miscellaneous fields.

        SETUP is run like any other compiled Pascal program, by
entering the Command level of the system, typing X for eXecute and
typing the filename SETUP followed by a carriage return. You should
see the following (user input underlined):

Execute what file? SETUP
INITIALIZING....................
....
SETUP: C(HANGE) T(EACH) H(ELP) Q(UIT) [C3]

        If this does not happen it may be because the setup program is
not on the disk. If so, the system will display the message

no file SETUP.CODE

If neither of the above happens, something is drastically wrong.
Contact UCSD. Assuming all is well, continue.

        All commands to the SETUP program are invoked by typing a
single letter chosen from the promptline.

SETUP: C(HANGE) T(EACH) H(ELP) Q(UIT)

        Type 'H' to find out what the commands at this level do. The
program is self teaching, so the rest of this document explains the
information SETUP was designed to change.

        SETUP does not tell the system how to do random access cursor
addressing on the user's terminal (for those terminals which have this
capability). To allow the system to use that feature, please refer to
Section 4.7 of this document package.

4.3.1 MISCELLANEOUS INFORMATION

It is interesting to note that on all PDP-11 systems, the key
which generates ASCII DC1 (or control-R); functions as an alpha-lock.

HAS CLOCK
Values: TRUE, FALSE
A real time clock is available. A real time clock module, such
as the DEC KW11, may be found on many processors. It is assumed to be a
line frequency (60 cycle) clock. If available it is used by the PASCAL
system to optimize disk directory updates. See section 2.1.5 TIME intrinsic.

STUDENT
Values: TRUE, FALSE
If true, tells the system to simplify certain parts of the
system for novice use. E.g., an error detected while compiling sends
student back to the editor without choice.

HAS 8510A
Values: TRUE, FALSE
The system is running on a Terak 8510a hardware configuration.

HAS BYTE FLIPPED MACHINE
Values: TRUE, FALSE
True if low order byte is in bits 0-7 of words on your
processor. (PDP11, 8080, 6502, FALSE. 9900, 6800, GA440, TRUE)

HAS WORD ORIENTED MACHINE
Values: TRUE, FALSE
True if sequential addresses address sequential 16 bit words,
False if sequential addresses address sequential 8 bit bytes.

4.3.2 GENERAL TERMINAL INFORMATION

HAS SLOW TERMINAL
Values: TRUE, FALSE.
When this field is true, the system issues abbreviated
promptlines and messages.
Suggested setting: 600 baud and under -- True, otherwise False.

HAS RANDOM CURSOR ADDRESSING
Values: TRUE, FALSE
Only applies to video terminals. See Section 4.7 in order to
allow the system to make use of this feature.

HAS LOWER CASE
Values: TRUE,FALSE

SCREEN WIDTH
The number of characters per line of a terminal.

SCREEN HEIGHT
The number of lines per display screen of a video terminal.
Set to 0 for a hard copy terminal or other terminal in which paging is

not appropriate.

NONPRINTING CHARACTER
        Values: Any printing character.
        What should be displayed by the terminal to indicate the
presence of a non-printing character.
        Recommended setting: ASCII "?".

VERTICAL MOVE DELAY
        The number of nulls to send after a vertical cursor move. Many
types of terminals require a delay after certain cursor movements which
enables the terminal to complete the movement before the next character
is sent. This number of nulls will be sent after carriage returns,
ERASE TO END OF LINE, ERASE TO END OF SCREEN and MOVE CURSOR UP.

4.3.3   CONTROL KEY INFORMATION

        The user may choose which control keys suit his particular
keyboard arrangement and his taste.

        Some keyboards generate two codes when some single key is
pressed. If that is the case for any of the keys mentioned here, it
must be noted in the field PREFIXED [<fieldname>] which has either the
value TRUE or the value FALSE. The prefix for all such keys must be
the same and must be noted in the field LEAD IN FROM KEYBOARD. This
feature may also be used to access control functions with two-
character sequences if a user's keyboard is unable to generate many
control characters. As an example, suppose the user's keyboard had a
vector pad which generated the value pairs ESC "U", ESC "D", ESC "L"
and ESC "R" for the keys for Uparrow, Downarrow, Leftarrow and
Rightarrow, respectively. Assume also that all other keys on the
keyboard generate only single codes. Then the user would give the
following fields the following values:
        KEY FOR MOVING CURSOR UP   ASCII "U"
        KEY FOR MOVING CURSOR DOWN   ASCII "D"
        KEY FOR MOVING CURSOR LEFT   ASCII "L"
        KEY FOR MOVING CURSOR RIGHT   ASCII "R"
        LEAD IN KEY FOR KEYBOARD   ESC
        PREFIXED[KEY FOR MOVING CURSOR UP]   TRUE
        PREFIXED[KEY FOR MOVING CURSOR DOWN]   TRUE
        PREFIXED[KEY FOR MOVING CURSOR LEFT]   TRUE
        PREFIXED[KEY FOR MOVING CURSOR RIGHT]   TRUE

KEY FOR STOP
        Console output stop character. The STOP character is a toggle;
when pressed, the key will cause output to the file 'OUTPUT' to cease.
When the key is depressed again, the write to file 'OUTPUT' will resume
where it left off. This function is very useful for reading data which
is being displayed faster than one can read.
        Suggested setting: ASCII DC3

KEY FOR FLUSH
    Console output cancel character. Similar in concept and usage
to the STOP key, the FLUSH key will cause output to the file 'OUTPUT'
to go undisplayed until FLUSH is pressed again or the system writes to
file 'KEYBOARD'. Note that, unlike the STOP key, processing continues
uninterrupted while output goes undisplayed.
    Suggested setting: ASCII ACK

KEY FOR BREAK
    Typing the character BREAK will cause the program currently
executing to be terminated with a run-time error immediately.
    Suggested setting: Something difficult to hit accidentally.

KEY TO END FILE
    Console end of file character. When reading from the files
KEYBOARD or INPUT or the unit 'CONSOLE:', this key sets the Boolean
function EOF to TRUE. See section 2.2.4 EOF intrinsic.
    Suggested setting: ASCII ETX

KEY TO DELETE CHARACTER
    Each time you press this key one character is removed from the
current line, until nothing is left on that line.
    Suggested setting: ASCII BS

KEY TO DELETE LINE
    Depressing LINE DELETE will cause the current line of input to
be erased.
    Suggested setting: ASCII DEL

            The rest of this section contains information
            only of interest to users who are using video
            display terminals with a selective erase
            capability and may be safely ignored by users
            having any other kind of terminal, such as
            hardcopy terminals or storage tube terminals.

KEY TO MOVE CURSOR UP
KEY TO MOVE CURSOR DOWN
KEY TO MOVE CURSOR LEFT
KEY TO MOVE CURSOR RIGHT
    These keys are used by the screen oriented editor to control
the basic motions of the cursor. If the keyboard has a vector pad, set
these fields to the values it generates, otherwise, we suggest
choosing 4 keys in the pattern of a vector pad and use the control
codes which correspond to them, for example the keys 'O', '.', 'K' and
';' on most keyboards encircle an imaginary vector pad. You may wish
to use a prefix character before such keys as described above.

EDITOR ESCAPE KEY

The key which, in the system screen oriented editor, is to be used to escape from commands, reversing any action taken.
Suggested setting: ASCII ESC

EDITOR ACCEPT KEY
The key which, in the system screen oriented editor, is to be used to accept commands, making permanent any action taken.
Suggested setting: ASCII ETX

4.3.4 VIDEO SCREEN CONTROL CHARACTERS

This section describes the characters which, went sent to the terminal by the computer, controls the terminals actions. Yoou should consult the manual for your terminal to find the appropriate values. If a terminal does not have one of these characters, the field should be set to 0 unless otherwise directed.

Some screens require a two character sequence to exercise some of their functions. If the first character in all of these sequences is the same, it can be set as the value of the field LEAD IN TO SCREEN and for each <fieldname> which requires that prefix, the user must set the field PREFIX[<fieldname>] to TRUE. For example, suppose ERASE TO END OF LINE and ERASE TO END OF SCREEN were respectively performed by the sequences ESC "L" and ESC "S" but all the other screen controls were single characters. The user would then set the following fields to the following values:

LEAD IN TO SCREEN	ASCII ESC
ERASE TO END OF LINE	ASCII "L"
ERASE TO END OF SCREEN	ASCII "S"
PREFIXED[ERASE TO END OF SCREEN]	TRUE
PREFIXED[ERASE TO END OF LINE]	TRUE.

ERASE TO END OF SCREEN
The character which erases the screen from the current cursor position to the end of the screen.

ERASE TO END OF LINE
The character which, when sent to the screen, erases all characters from the current cursor position to the end of the line the cursor is on.

ERASE LINE
The character which, when sent to the screen, erases all the characters on the line the cursor is currently on.

ERASE SCREEN
The character which, when sent to the screen, erases the entire screen.

BACKSPACE
The character which, when sent to the screen, causes the cursor to move space to the left.

MOVE CURSOR HOME
        The character which moves your cursor to the upper left of the
current page. IMPORTANT: If your terminal does not have such a
character, set this field to CARRIAGE RETURN, ASCII mnemonic CR.

MOVE CURSOR UP
MOVE CURSOR LEFT
        The characters which move your cursor non-destructively one
space in those directions.

4.3.5   QUIT

        The quit mode of SETUP gives many options: Memory update, which
places the definitions in the memory cells which are appropriate.
Disk update, which creates the file NEW.MISCINFO. Return, which takes
the user back to setup, and Exit, which returns the user to the Pascal
command level.

4.3.6 QUICK REFERENCE SUMMARY

BACKSPACE
EDITOR ACCEPT KEY
EDITOR ESCAPE KEY
ERASE LINE
ERASE SCREEN
ERASE TO END OF LINE
ERASE TO END OF SCREEN
HAS 8510A
HAS BYTE FLIPPED MACHINE
HAS CLOCK
HAS LOWER CASE
HAS RANDOM CURSOR ADDRESSING
HAS SLOW TERMINAL
HAS WORD ORIENTED MACHINE
KEY FOR BREAK
KEY FOR FLUSH
KEY FOR STOP
KEY TO DELETE CHARACTER
KEY TO DELETE LINE
KEY TO END FILE
KEY TO MOVE CURSOR DOWN
KEY TO MOVE CURSOR LEFT
KEY TO MOVE CURSOR RIGHT
KEY TO MOVE CURSOR UP
LEAD IN FROM KEYBOARD
LEAD IN TO SCREEN
MOVE CURSOR HOME
MOVE CURSOR RIGHT
MOVE CURSOR UP
NON PRINTING CHARACTER

```
PREFIXED [DELETE CHARACTER]
PREFIXED [EDITOR ACCEPT KEY]
PREFIXED [EDITOR ESCAPE KEY]
PREFIXED [ERASE LINE]
PREFIXED [ERASE SCREEN]
PREFIXED [ERASE TO END OF LINE]
PREFIXED [ERASE TO END OF SCREEN]
PREFIXED [KEY FOR BREAK]
PREFIXED [KEY FOR FLUSH]
PREFIXED [KEY TO MOVE CURSOR DOWN]
PREFIXED [KEY TO MOVE CURSOR LEFT]
PREFIXED [KEY TO MOVE CURSOR RIGHT]
PREFIXED [KEY TO MOVE CURSOR UP]
PREFIXED [KEY FOR STOP]
PREFIXED [KEY TO DELETE CHARACTER]
PREFIXED [KEY TO DELETE LINE]
PREFIXED [KEY TO END FILE]
PREFIXED [MOVE CURSOR HOME]
PREFIXED [MOVE CURSOR RIGHT]
PREFIXED [MOVE CURSOR UP]
PREFIXED [NON PRINTING CHARACTER]
SCREEN HEIGHT
SCREEN WIDTH
STUDENT
VERTICAL MOVE DELAY
```

-- Notes --

```
******************** ***************
* BOOTSTRAP COPIER * * Section 4.4 *
******************** ***************
```

Version I.5   September 1978
The bootstrap copier BOOTER.CODE asks for the unitnumber of the
volume on which to write the bootstrap. Refer to Table 5 for a list of
volume numbers. It will then ask for a file name to write as the
bootstrap. It writes the first two blocks of that file, so in order to
copy the bootstrap from an existing disk, give it the diskname, and it
will copy the bootstrap from the disk named to the unit numbered.

To execute the BOOTER program, type X BOOTER to Command level
(assuming that there a copy of BOOTER.CODE on the disk).

— Notes --

Version I.5 September 1978

PATCH is a utility which was written as a personal piece of software, and has become part of the soul of the system. Even in the wonderful world of Pascal programming, it seems that the need to see disk blocks in the not so wonderful world of HEX remains. The usefulness of this proves itself over and over again. Usually this pertains to studying the output of a Pascal program which has created a file of some structured type, however the data in the output file just doesn't seem right. Patch comes to the rescue. Patch lets you see just exactly what bits are where, and even lets you change them to be the way they should be.

On X(ecuting PATCH, the promptline is

C(onsole, P(atchwrite, W(holewrite, Q(uit

The options available are:

Working with, and altering the file in the C(onsole mode.

Dumping the file in a Hex, Decimal, Octal, or ASCII format, in the P(atchwrite mode.

Dumping/concatenating and/or moving blocks in files with the W(holewrite mode.

Leaving PATCH with the Q(uit command.

In the C(onsole mode, the promptline changes with each command. The promptline always reflects the commands available at any given time, and no more. The full promptline is:

Patch: R(ead, S(ave, H(ex, M(ixed, G(et, Q(uit [nn]

The number in square brackets at the end of the prompt is the current block being patched. The first command to use is G(et. G(et will prompt

Filename: <cr for unit i/o>

Respond to this prompt with the name of the file to be patched. If the disk/device has no directory, or has some problem with the directory, reference it by its Pascal unitnumber. Type a carriage return to this prompt, and the prompt is:

Unitnum to patch [4,5,9..12] (0 will Quit)

Having typed a successful entry to one of the two above prompts, the prompt will now be extended by the R(ead command. R(ead will read up a block from the file/unit. The prompt on entering R(ead command is

BLOCK:

Respond with a block number in the file/unit specified. There is no range checking provided on this read, so exercise care in the number typed. The promptline is now extended with H(ex, M(ixed and the block number in square brackets. H(ex and M(ixed display the block read. Using the H(ex command displays the block entirely in hexadecimal characters, using the M(ixed command will display printing ASCII characters where possible, and hexadecimal values elsewhere. The promptline is:

Alter: H(ex, T(ext, S(tuff, Q(uit

The vector keys on the terminal causes the cursor to move around in the data, notice that there the cursor will remain only on the data, and will not move off the data. On terminals without vector keys, or poorly done setups, the character - motion table is as follows:

U - up
Z - down
L - left
R - right

Typing a hexadecimal character changes the character the cursor is over provided that only one or more of the data positions is changed, when Q(uitting from Alter mode, the Patch promptline will be extended with the S(ave command. Typing S(ave writes the changed data back to from where it was read. In the Alter mode, there is one optional command: S(tuff. Typing the S(tuff command displays the promptline:

Stuff for how many bytes:

Key a number from 0 to 512. Type carriage return to cause patch to accept the number, the promptline changes to:

Fill with what hex pair:

Key a byte value in hexadecimal. The data reappears on the screen, with the number of bytes specified, from the position of the cursor filled with the data value specified, to the hex pair prompt.

Using the Patchwrite command causes a full screen prompt to appear:

This procedure writes out sequential blocks to any file as a patch dump. Type the prefix character of the option to be changed. Type 'P' to PRINT, 'Q' to QUIT.

    A(Input File
    B(Begin Block #
    C(Num. of Blocks

    E(Output File

    G(Hexadecimal
    H(ASCII
    I(Decimal
    J(Octal
    K(Decimal Bytes
    L(Octal Bytes
    M(Krunch
    N(Double Space

Following each of the fields is the current value of that field. Typing the character in front of the field places the cursor after the field, and removes the current value. Typing 'Y' or 'T' sets a boolean value to True, any other character sets the field to False. The Input File and Output File fields require a filename to be typed followed by carriage return. The integer fields (Begin Block, and Num. of Blocks) require a number to be typed followed by carriage return or space. Any other character sets the value of the field to some unspecified value.

The other options at the Patchwrite level are Print and Quit. Both cause Patch to return to the outer level. Quit does it straight away, Print dumps out the file in the requested format on the way. The options available for the dump need to be selected, the default is none. The options Krunch and Double Space affect the formatting of the output. Krunch, when true, removes blank lines between logical output lines. Double Space when true, double spaces all output.

Using the W(holewrite command causes the full page prompt:

This procedure writes any number of blocks from an existing file
to a new file, unchanged. Simply specify the necessary parameters
Type 'P' to PUT, 'Q' to QUIT

    I(nput File
    S(tart Block
    N(umber of Blcks

    O(utput File

        The protocol for changing the fields at this level is the same
as that for the Patchwrite level. The Wholewrite level is that which
allows one to mix/match and mingle files. Put and Quit both cause
Patch to return to the outer level, Put writes to the file on its way,
Quit does not.

        Notice that the Patchwrite and Wholewrite levels remember their
vital parameters across sessions (while remaining in Patch). The
Console level will clear all memory of the session. The Patchwrite
level paginates its output, after each block written, a form-feed is
generated. (Specifically PAGE(OUTPUTFILE)).

```
****************************** ***************
* RT11 to PASCAL CONVERSION KIT * * Section 4.6 *
****************************** ***************
```

Version I.5   September 1978

        The utility file labeled RT11TOEDIT is intended for use with
RT-11 disks. It assumes the presence of an RT-11 directory spanning
blocks 6-7. When the file is executed it asks the user to specify the
Pascal system unitnumber of the volume of which the user wants to view
the directory. Once a legal on-line unit has been specified,
RT11TOEDIT reads each entry on blocks 6-7. The program uses the
UNITREAD intrinsic to read the directory and does not open the file in
the usual manner. It lists on the screen the entire contents of the
directory. For each entry it specifies the file title, file kind, the
size of the file in blocks, and the starting block location of the file
(in base 10). All unused portions are identified as such. The user
will be prompted for an RT-11 file name, a Pascal system file name, and
finally a mode of transfer.

-- Notes --

```
***************  **************
* GOTOXY BINDER *  * Section 4.7 *
***************  **************
```

Version I.5   September 1978

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

An example of a GOTOXY procedure for a relatively stupid
terminal follows. More intelligent terminals will require less effort
to have the proper cursor addressing happen. It is suggested that you
might want to fill an array or string with the appropriate characters
to cause your terminal to do its absolute addressing, and then
UNITWRITE the stream all at once. This will improve the performance
of the screen editor noticably. An example of this for the Datamedia
1520 follows the example for the DECscope VT-50.

If the GOTOXY cursor-addressing scheme for the terminal is not
there, create one. The procedure may not be named GOTOXY because
this identifier is predeclared at the "$U-" level of compilation.

Possible error:	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 the GOTOXY file

Assumptions:

1.) A screen terminal

2.) A PASCAL system

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

4.) GOTOXY corrects for bad input data.

See Section 2.1.2 for more information on GOTOXY.

EXAMPLE:

(*$U-,S+*)(* the psuedo comments inform the compiler of the correct
            state to be in for compiling this little routine *)

```
PROCEDURE MYGOTOXY(X,Y: INTEGER);
              (* the procedure must NOT be called GOTOXY *)
BEGIN
  (* check the input data to see that it is within the screen
     dimensions, on some smarter terminals, if a cursor position
     command is sent for a position that does not exist, the
     results are unpredictable *)
  IF X < 0 THEN X := 0
  ELSE
    IF X > 79 THEN X := 79;
  IF Y < 0 THEN Y := 0
  ELSE
    IF Y > 11 THEN Y := 11;
  (* for a DECscope VT-50, GOTOXY needs to be implemented by: *)
  (* send the cursor home, 0,0 *)
  WRITE(CHR(27),'H');
  (* while TAB is meaningful, use it to move the cursor *)
  WHILE X > 8 DO
    BEGIN
      WRITE(CHR(9));
      X := X-8;
    END;
  (* finish off what portion of the x coordinate could not be absorbed
     with TAB characters *)
  WHILE X > 0 DO
    BEGIN
      WRITE(CHR(27),'C');
      X := X-1
    END;
  (* send line-feeds to access the y coordinate *)
  WHILE Y > 0 DO
    BEGIN
      WRITE(CHR(10));
      Y := Y-1
    END
END;


BEGIN
  (* this dummy body of the operating system is needed to keep the
     Pascal compiler happy about having complete programs to compile.
     The method used for 'binding' the GOTOXY procedure is somewhat
     unclean, and only the code for the above procedure is used by
     the binder to add to SYSTEM.PASCAL *)
END.

(*$U-,S+*)

PROCEDURE ITSGOTOXY(X,Y: INTEGER);
VAR
  T: PACKED ARRAY[0..2] OF CHAR;
BEGIN
  T[0] := CHR(30); (* RS is Datamedias absolute cursor address flag *)
```

```
     (* set appropriate character for x coordinate *)
     IF X < 0 THEN T[1] := CHR(32)
     ELSE
       IF X > 79 THEN T[1] := CHR(32+79)
       ELSE
         T[1] := CHR(X+32);
     (* set appropriate character for y coordinate *)
     IF Y < 0 THEN T[2] := CHR(32)
     ELSE
       IF Y > 23 THEN T[2] := CHR(32+23)
       ELSE
         T[2] := CHR(Y+32);
     (* send the cursor where it belongs..... WHAPPO! *)
     UNITWRITE(1,T,3)
END;

BEGIN
  (* same comment applies *)
END.
```

-- Notes --

```
***********************************  **************
* DUPLICATE DIRECTORY UTILITIES *  * Section 4.8 *
***********************************  **************
```

Version I.5   September 1978

COPYDUPDIR

      This program will copy the duplicate directory into the primary
directory location. If the disk is not currently maintaining a current
directory the program will tell you so. To use this program e(x)ecute
COPYDUPDIR. The program will ask for the drive in which the copy is to
take place (4 or 5). If no duplicate directory is found it will tell
you after you indicate the drive unit. If the duplicate is found then
it will ask you if you are sure you want to destroy the directory in
blocks 2-5. A 'Y' will execute the copy, any other character will
abort the program.

MARKDUPDIR

      This program will mark a disk that is currently not maintaining a
duplicate directory so that it will. Caution must be exercised to be
sure that blocks 6-9 are free for use. If they are not one must re-
arrange the files as to make them free. One can tell if there
available by getting an E)xtended listing in the Filer and checking to
see where the first file starts. If the first file starts at block 6
or the first file starts at block 10 but there is a 4 block unused
section at the top, then the disk has not been marked. If however, the
first file starts at block 10 and there is no unused blocks at the
beginning of the directory then the disk has been marked.

SYSTEM.PASCAL	31	30-Aug-78	6	Codefile
.				
.				
.				

OR

\<unused\>	4		6	
SYSTEM.PASCAL	31	30-Aug-78	10	Codefile
.				
.				
.				

Both of the above cases indicate disks that have not been
marked. Below is the directory of a properly marked disk.

To execute this program e(X)ecute MARKDUPDIR. The program will ask you which unit contains the disk to be marked (4 or 5). The program will check to see if it thinks that the blocks 6-9 are free. If the program doesn't think so it will ask you if you are sure they are free ? Typing 'Y' will execute the mark, any other character will abort the program. Be sure that the space is free before marking it as a duplicate directory.

```
************************* **************
* P-CODE  DISASSEMBLER * * Section 4.9 *
************************* **************
```

Version I.5   September 1978

The disassembler reads a standard UCSD code file and outputs
symbolic psuedo-assembly (P-Code) along with various statistics
concerning opcode frequency, procedure calls, and data segment
references. The disassembler was originally written to collect
statistics on opcode frequency, etc. as an aid in making architecture
improvements. It has since been found helpful in debugging
interpreters, optimizing programs, and provides a source of further
information regarding some of subtleties of our implementation of
Pascal. All statistics gathered are collected by making a pass
through the code file instead of collecting them while the code file
is actually running.

4.9.1   DISASSEMBLY

        The Disassembler reads a code file that has been generated by
the UCSD Pascal Compiler. If a program USES a UNIT the disassembly
will include the UNIT only if the code file has been linked. Assembly
routines linked into a Pascal host will never be included in the
disassembly.

        The Disassembler is invoked by eXecuting DISASM.I5 and requires
the file OPCODES.I5 to be on the system disk. The Disassembler will
first prompt for an input code file, the suffix .CODE being assumed
and thus not required. The next question refers to the byte sex of
the machine the code file is intended to run on, that is whether the
first physical byte (byte 0) of a machine word is the most significant
byte of the word. For more information, see section 3.6 BYTE-
SWAPPING. For the PDP-11 and the 8080 families, physical byte 0 is
the least significant byte. Next the prompt will be for an output
file for the disassembled output. Since the output file is untyped,
CONSOLE: or PRINTER: (if it is on-line) may be used. The final
question at this stage is whether the user wishes to take control of
the disassembly, i.e. decide which procedures are disassembled as
opposed to all the procedures in the file.

        The following question regards the collection of statistics on
references to a particular Procedure's data segment. Should you
decide to control the disassembly you will be warned that all
statistics gathered are only gathered on those procedures which are
disassembled. Next you will be taken into the Segment Guide. This
level displays the segments you have by name and lets you decide on
which one you are interested in. The Procedure Guide follows to let
you decide on the particular procedure(s) that you wish to
disassemble. Typing an "L" at this point will list the procedure(s)

contained in this segment. A more complete description of this step
occurs in the next section. The Segment Guide may be re-entered by
typing "Q" in the Procedure Guide. Thus in this manner you may
disassemble several procedures in several different segments without
disassembling the entire file. The Segment Guide is exited by typing
"Q".

```
! 1    1    1:D     0 (*$L CONSOLE:*)
! 2    1    1:D     1 PROGRAM DISASMDEMO;
! 3    1    1:D     3 VAR  I:INTEGER;
! 4    1    1:D     4     TOMORROW:BOOLEAN;
! 5    1    1:D     5     COMMENT:STRING;
! 6    1    1:C     0 BEGIN
! 7    1    1:C     0   I:=0;
! 8    1    1:C     5   TOMORROW:=FALSE;
! 9    1    1:C     8   REPEAT
!10    1    1:C     8     I:=I+1;
!11    1    1:C    13     WRITELN('Disassembly — a step backwards...');
!12    1    1:C    74   UNTIL TOMORROW;
!13    1    1:C    77 END.
!
```

FIGURE 1 SAMPLE PASCAL PROGRAM

```
!           BLOCK #  1      OFFSET IN BLOCK=  0
!SEGMENT PROC    OFFSET#                              HEX CODE
!    1    1    0(000):   BPT        7               D507
!    1    1    2(002):   SLDC       0               00
!    1    1    3(003):   SRO        3               AB03
!    1    1    5(005):   SLDC       0               00
!    1    1    6(006):   SRO        4               AB04
!    1    1    8(008):   SLDO       3               EA
!    1    1    9(009):   SLDC       1               01
!    1    1   10(00A):   ADI                        82
!    1    1   11(00B):   SRO        3               AB03
!    1    1   13(00D):   LOD        1     3         B60103
!    1    1   16(010):   LCA       42    'Disassembly — a step backwards..
!
!    1    1   60(03C):   SLDC       0               00
!    1    1   61(03D):   CXP     WRITESTR           CD0013
!    1    1   64(040):   CSP     IOCHECK            9E00
!    1    1   66(042):   LOD        1     3         B60103
!    1    1   69(045):   CXP     WRITELN            CD0016
!    1    1   72(048):   CSP     IOCHECK            9E00
!    1    1   74(04A):   SLDO       4               E3
!    1    1   75(04B):   FJP        8               A1F6
!    1    1   77(04D):   RBP        0               C100
```

FIGURE 2   SAMPLE PROGRAM DISASSEMBLED

Figure 1 displays a sample Pascal program that has been listed
during compilation. Figure 2 displays the disassembled code of the
file generated by the compiler. The left 3 columns in figure 2
correspond to the 3 columns to the right of the line number in figure
1. They are segment number, procedure number, and offset within
procedure, respectively. The offset is also given in hex in
parentheses. A complete description of UCSD P-Code mneumonics is given
in section 3.4. The actual code that exists in the file is given in
hex in the rightmost column. The parameters to CXP's and CSP's are
converted to the procedure name if it is a known system procedure or
function. WRITESTR, WRITELN, and IOCHECK are some examples. The
string operand for LCA is printed as a string as evidenced by the line
with offset 16. Jumps have their operand(s) converted to an offset
from the start of the procedure so that the offset may act as a label.
Thus the 8 displayed in the operand field of the FJP at offset 75
really means a jump to the SLDO at offset 8. This is also true of case
jumps (XJP's). The block number and byte offset of the start of the
procedure are given relative to the start of the code file. Thus this
procedure starts at block 1, offset 0 of the code file. The segment
dictionary resides in block 0 for all code files.

4.9.2   DATA SEGMENT REFERENCE STATISTICS

The fourth prompt the Disassembler provides is a question
asking if you would like to keep track of all references to a
particular procedure's data segment. The most common use of these
statistics is in optimization of a given procedure's code file. By
re-arranging the order of declaration of variables one may change the
offset within a data segment that applies to a given variable. For
p-machine architecture reasons the first 16 words offset into the data
segment are the fastest and have optimized 1 byte instructions. Offsets
from 17 to 127 result in instructions as least 2 bytes long, while
references to greater than 127 require at least 3 bytes. By making the
most frequently used variables have the smaller offsets one may save
considerable code file space and possibly time during execution.

Data Segment size:	45	Data references:	5	Lex level	0

For segment DISASMDE Procedure # 1

Offset(word)	Total	%
3	3	60.00
4	2	40.00

FIGURE   3   SAMPLE PROGRAM'S DATA SEGMENT STATISTICS

Figure 3 shows the data segment statistics for our sample program. Clearly there is little to be gained from optimizing such a small program but the general idea can still be presented. By using the compiled listing shown in figure 1 one can match offsets to variables as such:

variable	offset
I	3
TOMORROW	4
COMMENT	5

Now by using the figures in figure 3 one can see that offset 3 or the variable I occurs most frequently and thus deserves it's position. This same idea carried out on a large program may result in substancial size savings. Notice that offset 6 nevers occurs and thus is not included in the statistics in figure 3.

The prompt for the output file for these statistics occurs after the disassembly has been completed. If you elect to collect these statistics you will be taken into the Segment and Procedure Guides as described in the previous section except that the prompt requests the selection of a data segment on which to collect statistics. In the Procedure Guide, "L" gives a listing of all the procedures in the selected segment by number, lex level, and data segment size. After the selection of a data segment, processing continues, as described in the previous section, from the point after the data segment question.

4.9.3   OPCODE, PROCEDURE CALL, AND JUMP STATISTICS

These statistics are collected as an aid in optimizing the architecture of P-Code and although they are interesting to look at they are of no real use to the typical user. For this reason they will be described only superficially.

Each opcode is given with a complete breakdown of which bit was most significant for each operand on any given occurrence of the opcode. These are presented in terms of totals and percentages of the number of occurrences of the opcode. In addition a histogram of the opcode occurrence as a percentage of the total number of opcodes disassembled runs along the righthand margin. There is also a table of jumps in terms of the number of bits required to represent the distance of the jump for both positive and negative jumps. Finally there are counts of all procedure calls listed by segment and procedure number.

The last prompt of the program is the file to which these
statistics are to be written.

-- Notes --

```
************************ ****************
* LIBRARY MAP UTILITY * * Section 4.10 *
************************ ****************
```

Version I.5   September 1978

        The program LIBMAP produces a map of a library (or code) file
and lists the linker information maintained for each segment of the
file. In the case of segments which are Pascal Units the map file
will also contain the interface section of the Unit. See section
3.3.2 for greater detail.

        The program first prompts for a library file name. As in the
linker, this may be an asterisk to indicate "*SYSTEM.LIBRARY". The
".CODE" suffix may be suppressed by appending a period to the full
file name.

        Example

            typing   references file

            *   *SYSTEM.LIBRARY
            FARKLE   :FARKLE.CODE
            OLD.LIBRARY.   :OLD.LIBRARY

        Typically, the map utility will be used to list library
definitions but the option is available to include intra-library symbol
references. Should this feature be desired, type a "Y" when queried
for a reference list. A space (or carriage return) is considered a
"N".

        The user is now prompted for an output file name. (".TEXT"
will be appended unless an extra period is used.) Typing just
carriage return defaults outpt to CONSOLE:. Several libraries may be
mapped at the same time. To quit, type a carriage return when
prompted for any file name.

        A sample map follows

LIBRARY MAP FOR *SYSTEM.LIBRARY

Segment # 0: PASCALIO separate procedure segment
    PASCALIO separate proc P #1
    FSEEK   separate proc P #1
    FSEEK   separate byte reference (once)
    FREADREA separate proc P #2
    FREADREA separate byte reference (once)
    FREADDEC separate proc P #4

```
        FREADDEC separate byte reference (once)
        FWRITERE separate proc  P #3
        FWRITERE separate byte reference (once)
        FWRITEDE separate proc  P #5
        FWRITEDE separate byte reference (once)
        DECOPS   separate byte reference (8 times)


        ------------------------------------------------------------------

Segment # 1:  DECOPS    separate procedure segment
        DECOPS   separate proc  P #1
        DECOPS   global addr    P #1,  I #0
        GDEC     global addr    P #1,  I #0


        ------------------------------------------------------------------
        ------------------------------------------------------------------
Segment # 3:  MAGIC     separate procedure segment
        POWER    separate proc  P #1
        POWER    separate byte reference (once)


        ------------------------------------------------------------------
```
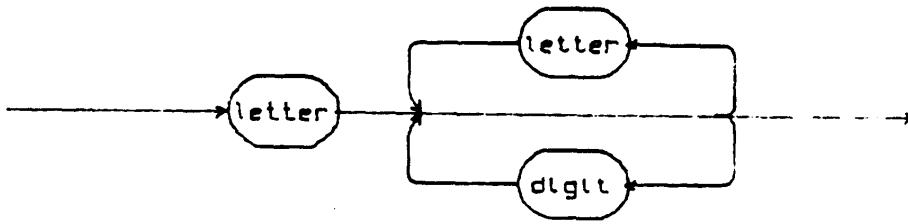
⟨unsigned integer⟩

⟨identifier⟩

⟨unsigned constant⟩

<constant>

<unsigned number>

<field list>

<simple type>

<Factor>

```
                    ┌──────────────────────┐
  ──────────────────│  unsigned constant   │──────────────────→
        │           └──────────────────────┘
        │                ┌────────────┐
        │           ─────│  variable  │──────────────────────→
        │                └────────────┘
        │        ┌──────────────────────┐      ┌────────────┐
        │     ───│ function identifier  │──( )─│ expression │──( )──→
        │        └──────────────────────┘      └────────────┘
        │                                         └──( , )──┘
        │                                                       ──→
        │                           ┌────────────┐
        │              ──( ( )──────│ expression │──( ) )─────────→
        │                           └────────────┘
        │              ┌─────┐     ┌ ─ ─ ─ ─ ─ ┐
        │           ──(  NOT )─────   factor        ─ ─ ─ ─ ─ →
        │              └─────┘     └ ─ ─ ─ ─ ─ ┘
        │
        └───────( [ )──────────────────────────────( ] )─
                   │    ┌────────────┐     ┌────────────┐
                   └────│ expression │──(..)─│ expression │──┘
                        └────────────┘       └────────────┘
                                  └──────( , )──────┘
```

<term>

```
           ┌────────┐
  ─────────│ factor │──────────────────────────────────────────→
      │    └────────┘   │     │     │      │      │
      │                (*)   (/)  (DIV)  (MOD)  (AND)
      │                 │     │     │      │      │
      └─────────────────┴─────┴─────┴──────┴──────┘
```

<expression>

<statement>

Page 260

<block>

<compilation>

<procedure>
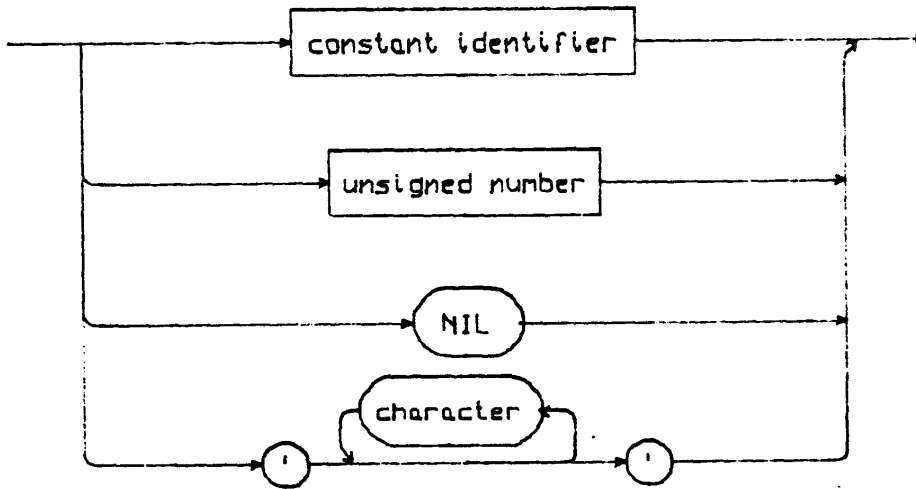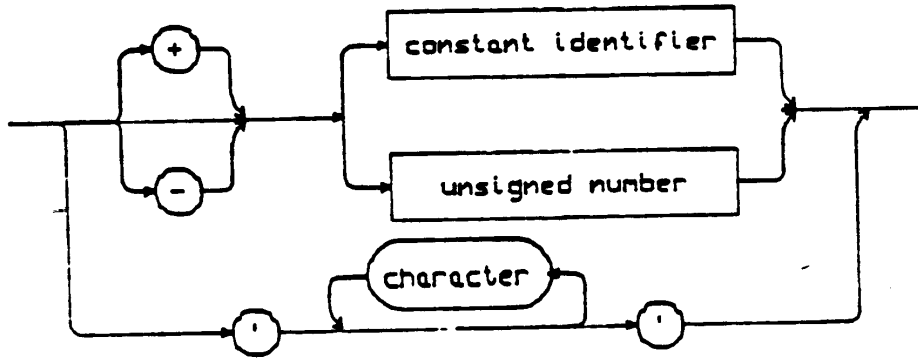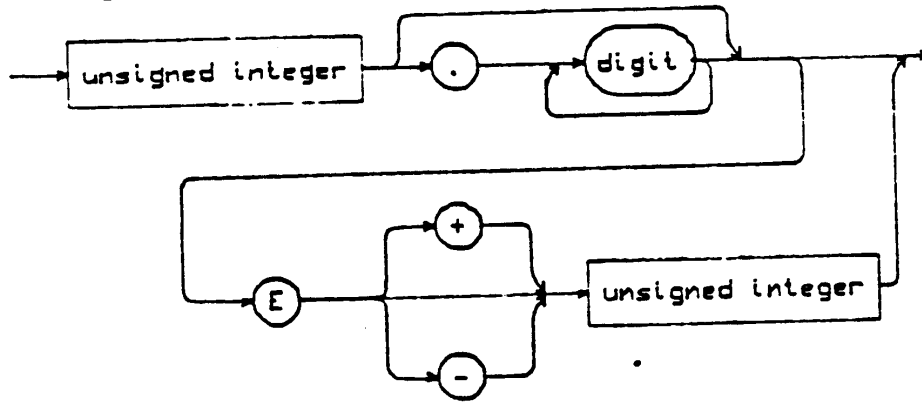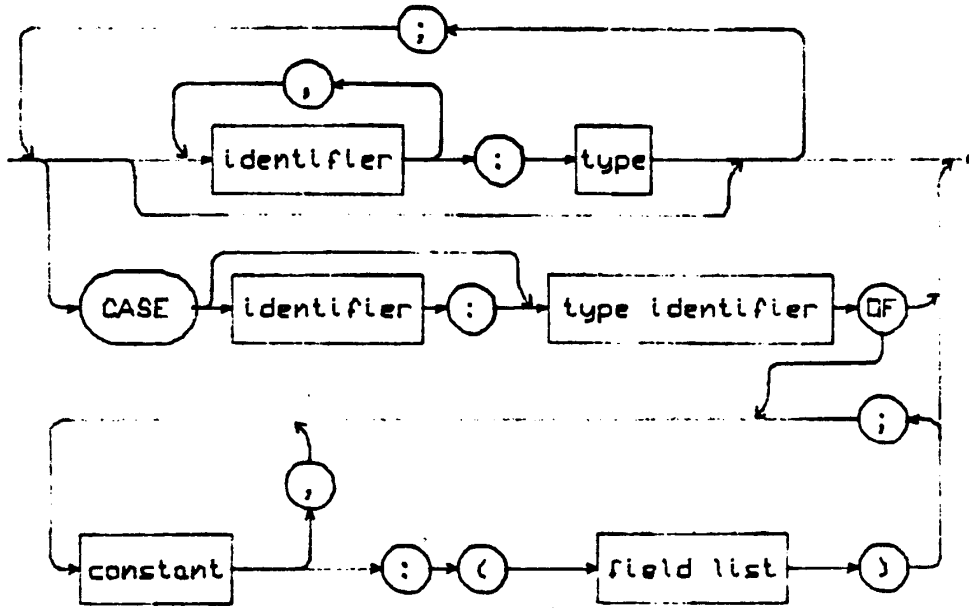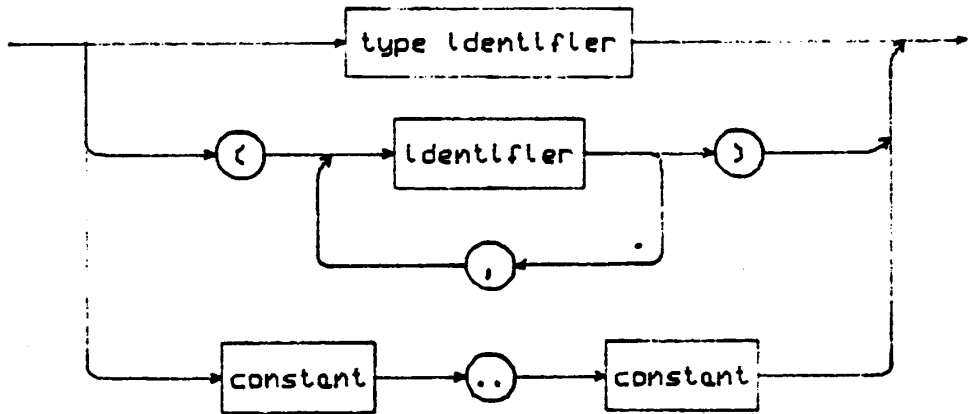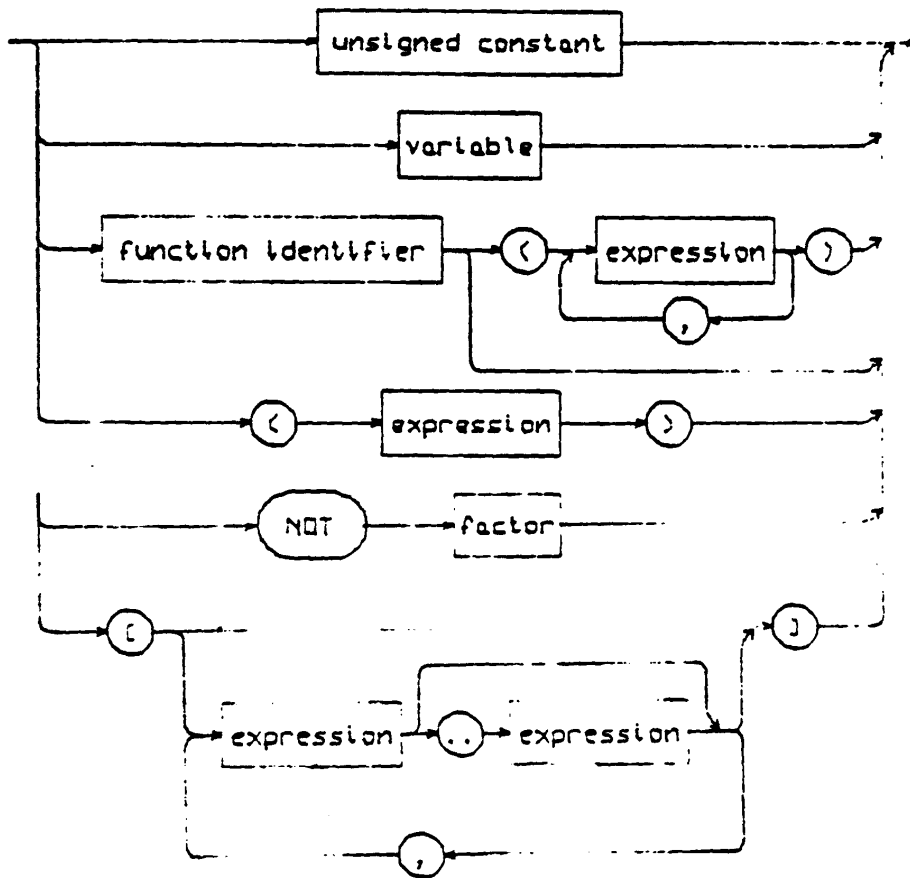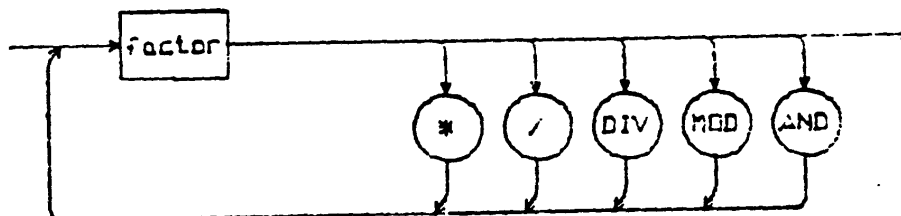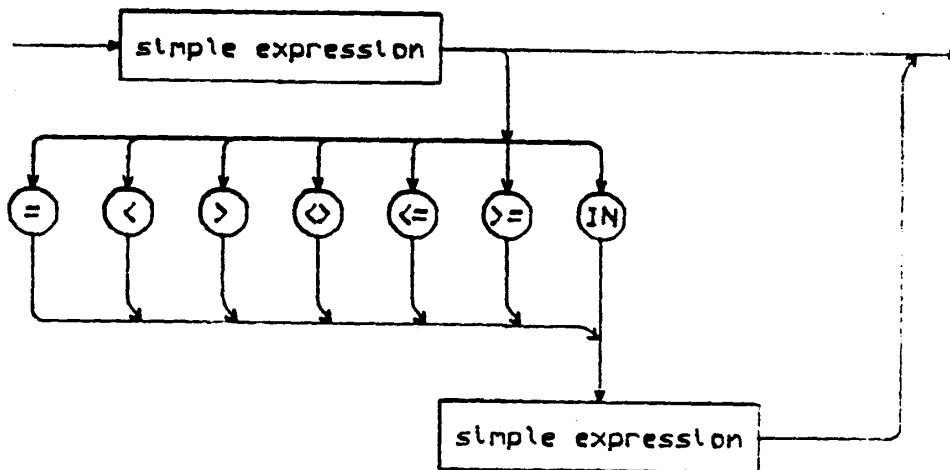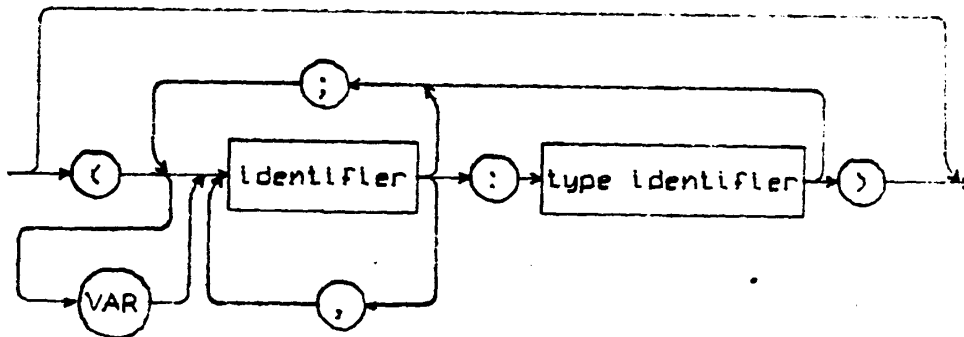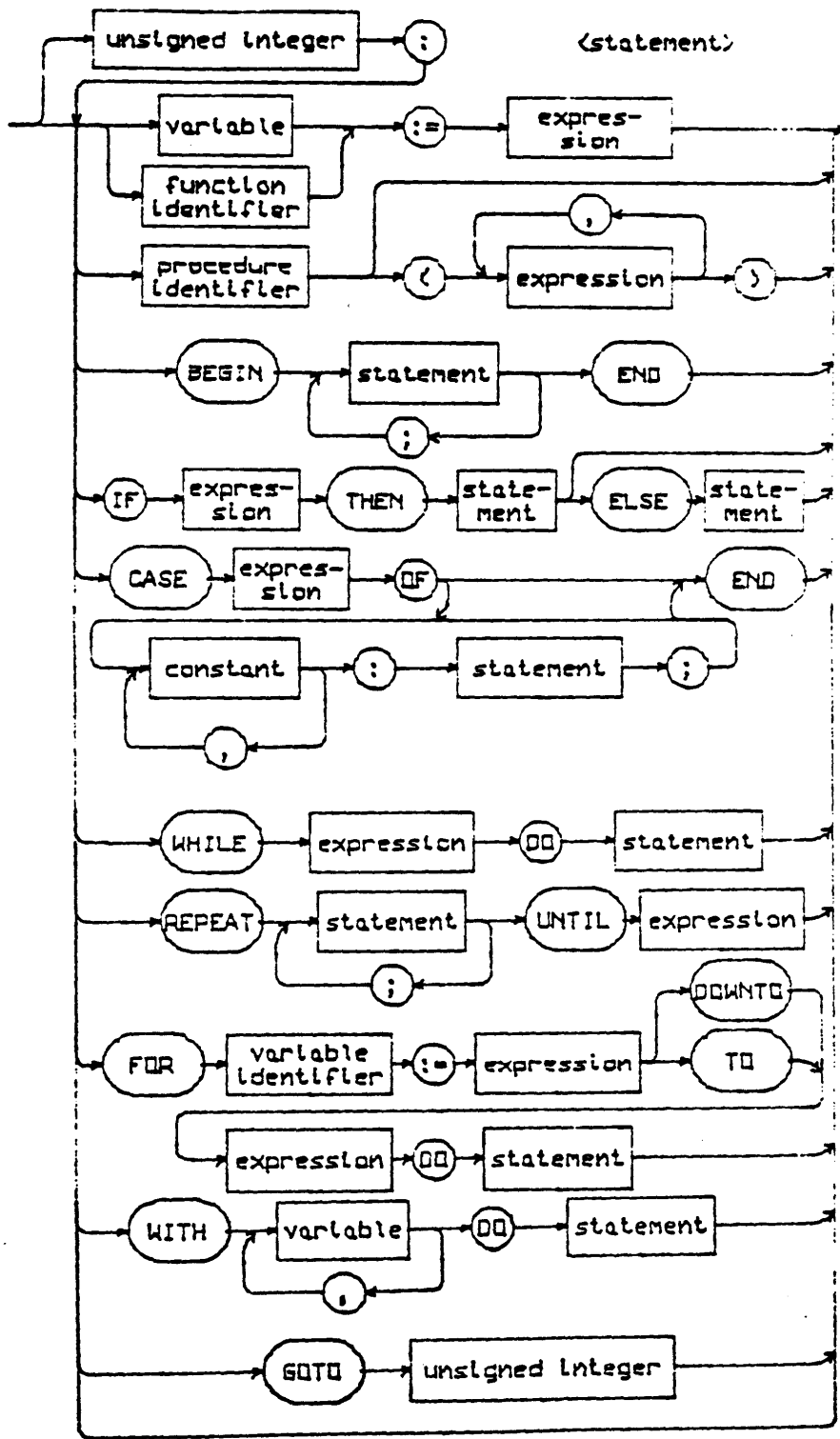
```
*********** ****** ***************
* TABLE 1 * * EXECUTION ERRORS *
*********** ****** ***************
```
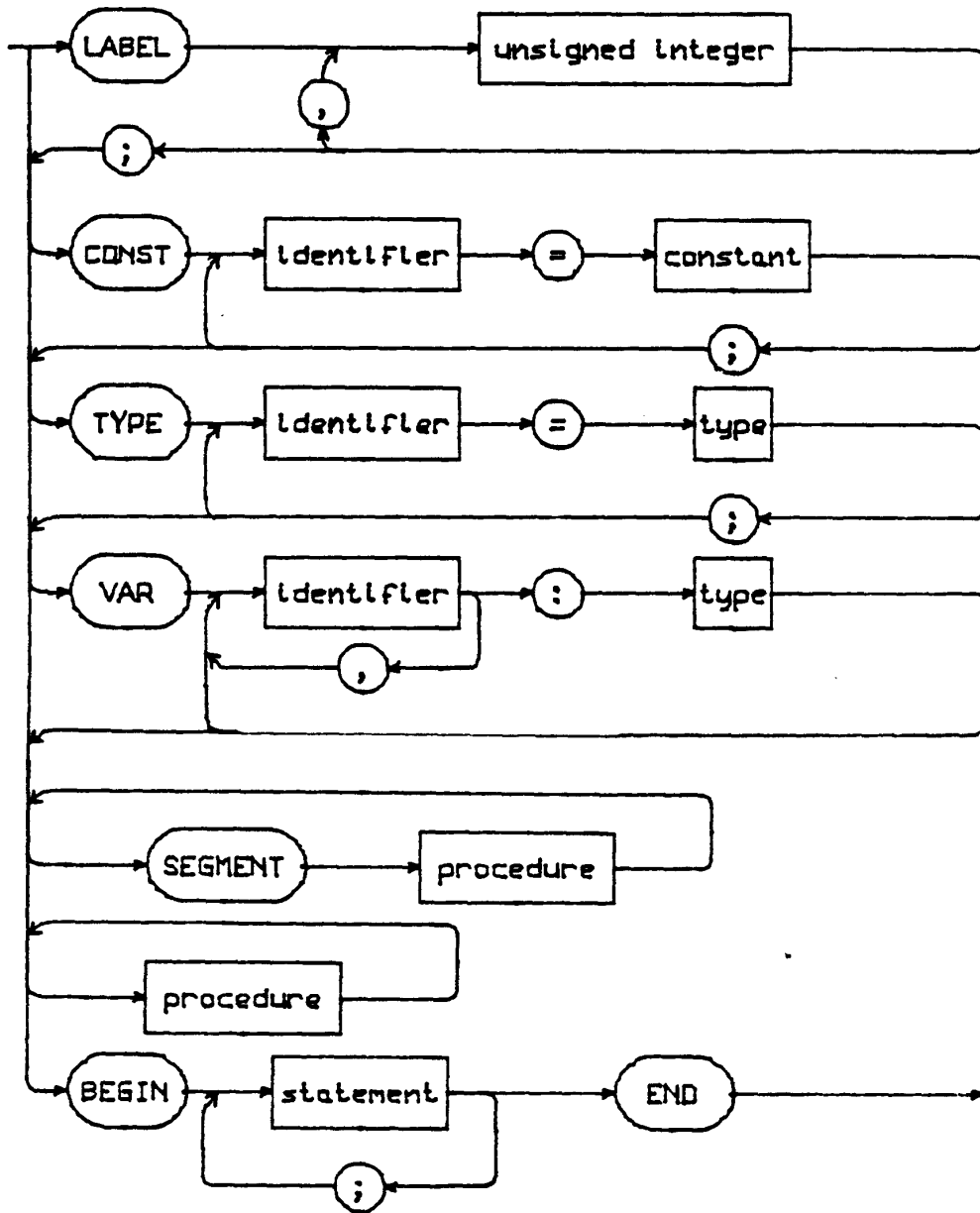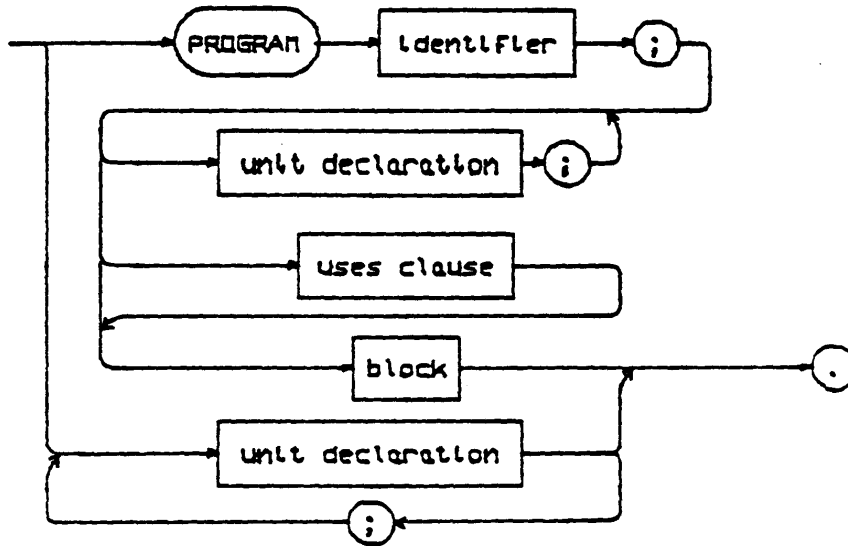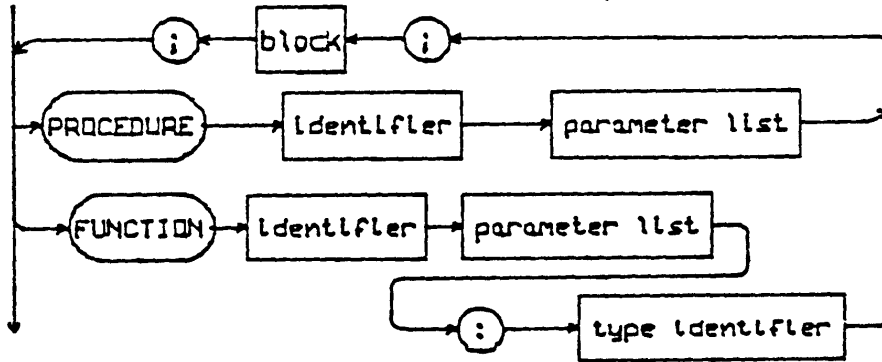
Version I.5   September 1978

0   System error   FATAL

1   Invalid index, value out of range (XINVNDX)

2   No segment, bad code file (XNOPROC)

3   Procedure not present at exit time (XNOEXIT)

4   Stack overflow (XSTKOVR)

5   Integer overflow (XINTOVR)

6   Divide by zero (XDIVZER)

7   Invalid memory reference <bus timed out> (XBADMEM)

8   User break (XUBREAK)

    9   System I/O error (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 —

```
************ *************
* TABLE 2  * * IORESULTS *
************ *************
```

Version I.5   September 1978

0	No error
1	Bad Block, Parity error (CRC)
2	Bad Unit Number
3	Bad Mode, Illegal operation
4	Undefined hardware error
5	Lost unit, Unit is no longer on-line
6	Lost file, File is no longer in directory
7	Bad Title, Illegal file name
8	No room, insufficient space
9	No unit, No such volume on line
10	No file, No such file on volume
11	Duplicate file
12	Not closed, attempt to open an open file
13	Not open, attempt to access a closed file
14	Bad format, error in reading real or integer
15	Ring buffer overflow

— Notes —

```
********** ***************
* TABLE 3 * * UNITNUMBERS *
********** ***************
```

Version I.5   September 1978

NUMBER	VOLUME NAME
0	<empty>
1	CONSOLE
2	SYSTERM
3	GRAPHIC
4	floppy0
5	floppy1
6	PRINTER
7	REMIN
8	REMOUT
9	block1
10	block2
11	block3
12	block4

Devices 9 - 12 are block-structured devices, in most cases (RK-05).

— Notes —

```
********** ******************
* TABLE 4 * * RESERVED WORDS *
********** ******************
```

Version I.5   September 1978

STANDARD PASCAL RESERVED WORDS   UCSD RESERVED WORDS

AND   SEGMENT
ARRAY   SEPERATE
BEGIN
BOOLEAN   UNIT
CASE   INTERFACE
CHAR   IMPLEMENTATION
CONST
DIV
DO
DOWNTO
ELSE
END
FILE
FOR
FUNCTION
GOTO
IF
IN
INTEGER
LABEL
MOD
NIL
NOT
OF
OR
PACKED
PROCEDURE
PROGRAM
REAL
RECORD
REPEAT
SET
STRING
THEN
TO
TYPE
UNTIL
VAR
WHILE
WITH

— Notes —

```
**********   ********************************
* TABLE 5 *   * SYNTAX ERRORS IN UCSD PASCAL *
**********   ********************************
```

Version I.5   September 1978

        The syntax errors this compiler gives are not the best it can
do. When time comes available to do so, the error generation of the
compiler is going to be seriously re-vamped.

    1: Error in simple type
    2: Identifier expected
    3: 'PROGRAM' expected
    4: ')' expected
    5: ': ' expected
    6: Illegal symbol
    7: Error in parameter list
    8: 'OF' expected
    9: '(' expected
   10: Error in type
   11: '[' expected
   12: ']' expected
   13: 'END' expected
   14: ';' expected
   15: Integer expected
   16: '=' expected
   17: 'BEGIN' expected
   18: Error in declaration part
   19: error in <field-list>
   20: '.' expected
   21: '*' expected
   22: 'Interface' expected
   23: 'Implementation' expected
   24: 'Unit' expected

   50: Error in constant
   51: ': =' expected
   52: 'THEN' expected
   53: 'UNTIL' expected
   54: 'DO' expected
   55: 'TO' or '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
134: Illegal type of operand(s)
135: Type of operand must be boolean
136: Set element type must be scalar or subrange
137: Set element types must be compatible
138: Type of variable is not array
139: Index type is not compatible with the declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter solution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be scalar
149: Index type must be integer
150: Assignment to standard function is not allowed

151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable cannot be formal or non-local

156: Multidefined case label
157: Too many cases in case statement
158: No such variant in this record
159: Real or string tagfields not allowed
160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: 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
174: Pascal function or procedure expected
182: Nested units not allowed
183: External declaration not allowed at this nesting level
184: External declaration not allowed in interface section
185: Segment declaration not allowed in unit
186: Labels not allowed in interface section
187: Attempt to open library unsuccessful
188: Unit not declared in previous uses declaration
189: 'Uses' not allowed at this nesting level
190: Unit not in library
191: No private files
192: 'Uses' must be in interface section
193: Not enough room for this operation
194: Comment must appear at top of program
195: Unit not importable

201: Error in real number - digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range
204: 8 or 9 in octal number
250: Too many scopes of nested identifiers
251: Too many nested procedures or functions
252: Too many forward references of procedure entries
253: Procedure too long
254: Too many long constants in this procedure
256: Too many external references
257: Too many externals
258: Too many local files
259: Expression too complicated

300: Division by zero
301: No case provided for this value
302: Index expression out of bounds
303: Value to be 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
405: Call not allowed in separate procedure
406: Include file not legal

```
********** ***************************
* TABLE 6 * * ASSEMBLER SYNTAX ERRORS *
********** ***************************
```

Version I.5   September 1978
        This section lists all the general errors found in the ERRORS
file, specific machine errors are found in the sections below
dealing with machine specifics.

     1: Undefined label
     2: Operand out of range
     3: Must have procedure name
     4: Number of parameters expected
     5: Extra garbage on line
     6: Imput line over 80 characters
     7: Not enough ifs
     8: Must be declared in ASECT before use
     9: Identifier previously declared
    10: Improper format
    11: EQU expected
    12: Must EQU before use if not to a label
    13: Macro identifier expected
    14: Word addressed machine
    15: Backward ORG not allowed
    16: Indentifier expected
    17: Constant expected
    18: Invalid structure
    19: Extra special symbol
    20: Branch too far
    21: Variable not PC relative
    22: Illegal macro parameter index
    23: Not enough macro parameters
    24: Operand not absolute
    25: Illegal use of special symbols
    26: Ill-formed expression
    27: Not enough operands
    28: Cannot handle this relative
    29: Constant overflow
    30: Illegal decimal constant
    31: Illegal octal constant
    32: Illegal binary constant
    33: Invalid key word
    34: Unexpected end of input - after macro
    35: Include files must not be nested
    36: Unexpected end of input
    37: Bad place for an include file
    38: Only labels & comments may occupy column one
    39: Expected local label
    40: Local label stack overflow
    41: String constant must be on 1 line
    42: String constant exceeds 80 chars
    43: Illegal use of macro parameter

```
44: No local labels in ASECT
45: Expected key word
46: String expected
47: Bad block, parity error (crc)
48: Bad unit number
49: Bad mode, illegal operation
50: Undefined hardware error
51: Lost unit, no longer on-line
52: Lost file, no longer in directory
53: Bad title, illegal file name
54: No room, insufficient space
55: No unit, no such volumn on-line
56: No file, no such file on volumn
57: Duplicate file
58: Not closed, attempt to open an open file
59: Not open, attempt to access a closed file
60: Bad format, error in reading real or integer
61: Nested macro definitions not allowed
62: '=' or '<>' expected
63: May not EQU to undefined labels
```

Z80 Based machines

For constants, Hex is the default type,
                a 'B' defines binary ex. 10010B ,
                a '.' defines decimal ex. 5674. .

Location Counter (LC) = $

All reserved words may not be used for any other purpose
such as an identifier. For example, the reserved word "C"
currently is being used as a register and in a condition
code, therefore it may not be used for any other purpose
(this is contrary to usual Zilog assembly language, but is
restricted in the UCSD assembler).

Specific error messages:

```
76: Incorrect operand format
77: Close paren ")" expected
78: Comma "," expected
79: Plus "+" expected
80: Open paren "(" expected
81: Stack pointer "SP" expected
82: "HL" expected
83: Illegal "CC" condition code
84: Register "C" expected
85: Register "R" expected
86: Register "A" expected
```

PDP11 Based machines:

For constants, Octal is the default type for both input
and output,
a 'H' defines hexadecimal ex. 056H ,
a '.' defines decimal   ex. 546. ,
a 'B' defines binary   ex. 1001B .

Location Counter (LC) = *

Specific error messages:

76: Closing paren ")" expected
77: Register expected
78: Too many special symbols
79: Unrecognizable operand
80: Register reference only
81: First operand must be a register
82: Comma expected
83: Unimplimented instruction
84: Must branch backwards to label

— Notes —

```
**********  *****************************************************
* TABLE 7 * * American Standard Code for Information Interchange *
**********  *****************************************************
```

Version I.5   September 1978

```
 0 000 00  NUL    32 040 20  SP    64 100 40  @     96 140 60  `
 1 001 01  SOH    33 041 21  !     65 101 41  A     97 141 61  a
 2 002 02  STX    34 042 22  "     66 102 42  B     98 142 62  b
 3 003 03  ETX    35 043 23  #     67 103 43  C     99 143 63  c
 4 004 04  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
```

— Notes —

```
*********** **********************
* TABLE 8 * * P-MACHINE OP-CODES *
*********** **********************
```

Version II.0   February 1979

```
  0 000 00  SLDC   0
  1 001 01  SLDC   1
  .   .   .    .     .
  .   .   .    .     .
  .   .   .    .     .
126 176 7E  SLDC 126
127 177 7F  SLDC 127
128 200 80  ABI        171 253 AB  SRO        214 326 D6  XIT
129 201 81  ABR        172 254 AC  XJP        215 327 D7  NOP
130 202 82  ADI        173 255 AD  RNP        216 330 D8  SLDL  1
131 203 83  ADR        174 256 AE  CIP        217 331 D9  SLDL  2
132 204 84  AND        175 257 AF  EQU        218 332 DA  SLDL  3
133 205 85  DIF        176 260 B0  GEQ        219 333 DB  SLDL  4
134 206 86  DVI        177 261 B1  GRT        220 334 DC  SLDL  5
135 207 87  DVR        178 262 B2  LDA        221 335 DD  SLDL  6
136 210 88  CHK        179 263 B3  LDC        222 336 DE  SLDL  7
137 211 89  FLO        180 264 B4  LEQ        223 337 DF  SLDL  8
138 212 8A  FLT        181 265 B5  LES        224 340 E0  SLDL  9
139 213 8B  INN        182 266 B6  LOD        225 341 E1  SLDL 10
140 214 8C  INT        183 267 B7  NEQ        226 342 E2  SLDL 11
141 215 8D  IOR        184 270 B8  STR        227 343 E3  SLDL 12
142 216 8E  MOD        185 271 B9  UJP        228 344 E4  SLDL 13
143 217 8F  MPI        186 272 BA  LDP        229 345 E5  SLDL 14
144 220 90  MPR        187 273 BB  STP        230 346 E6  SLDL 15
145 221 91  NGI        188 274 BC  LDM        231 347 E7  SLDL 16
146 222 92  NGR        189 275 BD  STM        232 350 E8  SLDO  1
147 223 93  NOT        190 276 BE  LDB        233 351 E9  SLDO  2
148 224 94  SRS        191 277 BF  STB        234 352 EA  SLDO  3
149 225 95  SBI        192 300 C0  IXP        235 353 EB  SLDO  4
150 226 96  SBR        193 301 C1  RBP        236 354 EC  SLDO  5
151 227 97  SGS        194 302 C2  CBP        237 355 ED  SLDO  6
152 230 98  SQI        195 303 C3  EQUI       238 356 EE  SLDO  7
153 231 99  SQR        196 304 C4  GEQI       239 357 EF  SLDO  8
154 232 9A  STO        197 305 C5  GRTI       240 360 F0  SLDO  9
155 233 9B  IXS        198 306 C6  LLA        241 361 F1  SLDO 10
156 234 9C  UNI        199 307 C7  LDCI       242 362 F2  SLDO 11
157 235 9D             200 310 C8  LEQI       243 363 F3  SLDO 12
158 236 9E  CSP        201 311 C9  LESI       244 364 F4  SLDO 13
159 237 9F  LDCN       202 312 CA  LDL        245 365 F5  SLDO 14
160 240 A0  ADJ        203 313 CB  NEQI       246 366 F6  SLDO 15
161 241 A1  FJP        204 314 CC  STL        247 367 F7  SLDO 16
162 242 A2  INC        205 315 CD  CXP        248 370 F8  SIND  0
163 243 A3  IND        206 316 CE  CLP        249 371 F9  SIND  1
164 244 A4  IXA        207 317 CF  CGP        250 372 FA  SIND  2
165 245 A5  LAO        208 320 D0  LPA        251 373 FB  SIND  3
166 246 A6  LSA        209 321 D1             252 374 FC  SIND  4
```

```
167 247 A7              210 322 D2              253 375 FD  SIND 5
168 250 A8  MOV         211 323 D3  EFJ         254 376 FE  SIND 6
169 251 A9  LDO         212 324 D4  NFJ         255 377 FF  SIND 7
170 252 AA  SAS         213 325 D5  BPT
         170 252 AA  SAS       213 325 D5  BPT
```

```
************** *********
* Appendix A * * Index *
************** *********
```