RCSL No: 42-i1515

Edition: September 1980

Author: -

Title:

UCSD Pascal
User's Guide

A/S REGNECENTRALEN
af 1979

Keywords:

UCSD Pascal, User's Guide.

Abstract:

One Pascal for all microcomputers.

(452 printed pages)

Copyright © 1980, A/S Regnecentralen af 1979
RC Computer A/S
Printed by A/S Regnecentralen af 1979, Copenhagen

Users of this manual are cautioned that the specifications contained herein are subject to change by RC at any time without prior notice. RC is not responsible for typographical or arithmetic errors which may appear in this manual and shall not be responsible for any damages caused by reliance on any of the materials presented.

4241341

FOREWORD

A written statement from SOFTECH dated the 1st of October 1980 permits A/S REGNECENTRALEN af 1979/RC COMPUTER to copy and sell this manual.

Kaj Riis
A/S REGNECENTRALEN af 1979/RC COMPUTER, February 1981

SOFTECH MICROSYSTEMS

A SUBSIDIARY OF SOFTECH

UCSD PASCAL

VERSION II.0

A PRODUCT FOR MINI- AND MICRO-COMPUTERS

USERS' MANUAL

Third printing with revisions: February 1980

UCSD Pascal is a trademark of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only, and any unauthorized use is contrary to the laws of the State of California.

Copyright© 1978 by the Regents of the University of California (San Diego)

New material copyright© 1979,1980 by SofTech Microsystems, Inc.
All rights reserved.
No part of this work may be reproduced in any form or by any means or used to make a derivative work (such as a translation, transformation, or adaptation) without the permission in writing of SofTech Microsystems, Inc.

9494 Black Mountain Rd., San Diego, CA 92126 (714) 578-6105 TWX: 910-335-1594

DISCLAIMER: These documents and the software they describe are subject to change without notice. No warranty express or implied covers their use. Neither the manufacturer nor the seller is responsible or liable for any consequences of their use.

SofTech Microsystems provides telephone and mail support for those users who have purchased their system from either SofTech Microsystems or UCSD (Version I.5 or later). This includes users who purchased their system from retail outlets of SofTech Microsystems. All other users of UCSD Pascal should contact their supplier for support. SofTech Microsystems will not support users who purchased their system from other vendors.

ACKNOWLEDGEMENTS:

The work described in these notes was supported significantly by the following organizations;

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

This work was made possible by the drive and direction of the director of the Institute for Information Systems:

Professor Kenneth L. Bowles.

DOCUMENTATION AUTHORS:

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

At SofTech Microsystems:
Randy Clark, Barry Demchak, Rich Gleaves.

SOFTWARE AUTHORS:

At UCSD: Mark Allen, Lucia A. Bennett, David Berger, Marc Bernard, J. Greg Davidson, Barry Demchak, Gary J. Dismukes, William P. Franks, Julie E. Erwin, Rich Gleaves, Robert J. Hofkin, Albert A. Hoffman, Richard S. Kaufmann, Peter A. Lawrence, Joel J. McCormack, Robert A. Nance, Mark D. Overgaard, David A. Reisner, Keith A. Shillington, David M. Steinore, Roger T. Sumner, Steven S. Thompson, David B. Wollner.

At SofTech Microsystems:
Mark Allen, David Berger, Barry Demchak, Rich Gleaves, Richard Kaufmann, Mark Overgaard.

COLLECTED AND EDITED BY:

At UCSD: Keith Allan Shillington and Gillian M. Ackland.

At SofTech Microsystems: Randy Clark.

WITH SPECIAL THANKS TO:
Tracy Barrett and the entire support staff, past and present . . .

At SofTech Microsystems:
H. Blake Berry, Jr., David Barto, Carolyn Chase, Randy Clark, Nancy Lanning, Bruce Sherman, George Symons.

TABLE OF CONTENTS

SECTION PAGE

ADDED SECTION: BOOTING UNDER THE CP/M OPERATING SYSTEM

ADDED SECTION: ADAPTABLE SYSTEM MANUAL

IV. ADDING CAPABILITIES

A. VECTOR LISTS AND REGISTER ASSIGNMENTS

B. SAMPLE BOOTSTRAP LOADERS

C. GENERAL NOTES

D. RECONFIGURING THE UCSD PASCAL INTERPRETER

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

The UCSD Pascal system described in this document is a system intended to run on standalone mini- and micro-computers. This system is highly machine independent since its code is executed by a pseudo-machine interpreter known as the "P-machine". All system software is written in Pascal, except for the P-machine interpreter itself, and a few runtime support routines -- these are distributed in the native code of a variety of processors. This configuration results in highly transportable software that is also straightforward to maintain and improve.

The system is designed to be used with an interactive terminal known as CONSOLE: -- this is preferably a CRT terminal, although the system may be reconfigured to use a slower hardcopy terminal. The system does require some kind of fast mass storage such as a floppy disk system or something more sophisticated. For further information on machine-specific matters, see the section on terminal handling, 4.2, and section A at the end of this document.

This document is aimed at system users who have some familiarity with using computers in general, and with the Pascal language. If you wish to do additional reading as background for this manual, here are a few books on the subject:

The standard reference (highly recommended):

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

Tutorials on Pascal:

A Practical Introduction to Pascal
I.R. Wilson and A.M. Addyman
Springer-Verlag, New York, 1978;

Microcomputer Problem Solving Using Pascal
Kenneth L. Bowles
Springer-Verlag, New York, 1977.

An introduction to the UCSD system:

Beginner's Manual for the UCSD Pascal System
Kenneth L. Bowles
Byte Books (McGraw-Hill), Peterborough, New Hampshire, 1979.

All these books are available from stores or from their publishers; some of them are provided by SofTech Microsystems, but only as supplements to software packages. This is only a partial list of the current literature.

Section 2.2 in this manual details the differences between UCSD Pascal and Standard Pascal.

1.1.1 THE UCSD PASCAL SYSTEM: AN OVERVIEW

The structure of the UCSD Pascal system is best pictured as the treelike form of a structure diagram. The "root" corresponds to the outermost level, while the "leaves" correspond to commands and called programs which have no internal structure visible to the user. When a user is at the outer level, or one of the intermediate levels, the system displays a list of available commands -- the prompt line or menu. If the system console is a CRT, the prompt line appears at the top of the screen. Commands are usually invoked by typing a single character at the console keyboard.

This is the prompt line for the outermost level of the system:

```
Command: E(dit, R(un, F(ile, C(omp, L(ink, X(ecute, A(ssem, D(ebug, ? [II.0]
```

Typing "F" causes the user to descend a level in the tree, and enter the system program called the Filer. This is itself a level with its own prompt line and set of commands. The Q(uit command will cause the user to exit the Filer and ascend back to the outermost system level. Some commands require further information, and will prompt the user to enter the name of a file. In these cases, a filename should be entered followed by a carriage return. The command may be aborted by hitting return with no filename at all. Mistakes made when typing the name may be corrected by backspacing and retyping, or by using the line delete key (also called "rubout") to delete the filename and start over.

The system level is the level that appears when you first bootstrap the system.

Some promptlines contain more commands than will fit on the screen. In this case, typing a question mark (?) will cause the remainder of the prompt to appear; typing a space restores the original prompt line.

The system will maintain a conceptual area known as the workfile. This is where work-in-progress is kept. There may be only one workfile, and it may consist of a text file, or a file of source program text accompanied by a code file. Certain of the system commands described immediately below refer directly to the workfile. More direct manipulation of old and new workfiles is done within the Filer.

You should think of the workfile as a scratchpad area for keeping new and unnamed material. It is a convenient place for doing quick work, as it relieves the user of having to specify filenames every time some editing or program running is begun or ended.

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 the workfile or any other textfile. See Section 1.3 for details. The workfile text, if present, is read into the editor's buffer, otherwise the editor prompts for a new text 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, see Section 1.2. Filer capabilities include listing directories, getting old files, creating new workfiles, saving old workfiles, and transfering files from disk to disk.

C. C(omp

This command initiates the system compiler, which compiles the user's workfile. If there is no current workfile, the user is asked for a source file name (source files are text files). 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" (escape character) causes the compiler to abort and return to the system level. Typing "E" will call the editor, and if this is the screen editor, place the cursor at the point where the error was detected. If the compilation is successful, a codefile called *SYSTEM.WRK.CODE is written out to the user's disk, and becomes part of the workfile. This codefile is used by the R(un command. For more documentation on the compiler, see Section 1.6.

D. R(un

This command causes the codefile associated with the current workfile to be executed. If no such codefile exists, the compiler is called, just as with the "C" command. If the compilation requires linkage to separately compiled code, the linker will be automatically invoked, and it will use the file *SYSTEM.LIBRARY. When a successfully compiled codefile exists, it will be executed.

E. X(ecute

This command will prompt the user for the filename of a previously compiled codefile. If the file exists, the codefile is executed. If it does not, the message "Can't find file" is returned. If all the code necessary to execute the codefile has not been linked in, the message "Must L(ink first" is returned.

The .CODE suffix on a codefile is implicit, and when the user answers the prompt, it need not be specified. X(ecute can work with any compiled codefile — not merely *SYSTEM.WRK.CODE. It is thus a more general command than R(un.

F. A(ssem

Works exactly like C(omp, except the system assembler is invoked rather than the system compiler. See Section 1.7 for a detailed description of the 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, just as in the case of the R(un command. While the program is running, a run-time error or a user-defined breakpoint or halt will cause a call to the system debugger. This is a program which has not yet been implemented: this command is therefore a stub for a future part of the system.

H. L(ink

This command starts the system linker program directly. It allows users to link in compiled or assembled routines from files other than *SYSTEM.LIBRARY. See Section 1.6 for more information on the Linker.

1.1.3 UTILITY PROGRAMS

Rather than clutter the commands at the system level with a variety of special-purpose functions, these functions have been made available as compiled utility programs. These are executed with the X(ecute command, just as any other precompiled program. A description of each utility is given in Section 4. These programs include the Librarian, the Setup program for tailoring a system to its hardware and the related GOTOXY Binder, a Bootstrap Copier, Patch/Dump utility, RT11ltoedit and EdittoRT11, Duplicate Directory, the Disassembler, and the Library Map utility.

1.1.4 AN INTRODUCTION TO THE UCSD PASCAL SYSTEM

II.0 is the name of the current release -- it is a stabler version of I.5, which was the first release to contain units and separate compilation and assembly. I.4b was the first release which supported a variety of processors, and I.3 was the system first released by UCSD into the outside world.

The great bulk of the system software is written in Pascal, and runs on a relatively simple pseudo-machine, the P-machine (see Sections 3.5 and 3.6). P-machines may be emulated on a variety of processors; both software (native code) emulations and firmware emulations have been successfully implemented.

Complete portability is thwarted by hardware-related aspects of the P-machine which must be emulated in different ways by different processors. Section A at the end of this manual details hardware-specific features of some currently supported processors. This is only a partial list, as SofTech Microsystems currently markets an Adaptable System which can be tailored to a greater variety of processors and peripherals. The Adaptable System is described in two sections bound in with this document -- see the Table of Contents.

Differences in the CONSOLE: hardware can be dealt with by the system's software; see Section 4.2.

Specific questions about hardware compatibility should be addressed to your personal supplier, or to SofTech Microsystems' Pascal Support.

The software that comprises the system itself is contained in files prefixed by the word "SYSTEM.". Some of these files relate to commands already discussed:

SYSTEM.FILER
SYSTEM.EDITOR
SYSTEM.COMPILER
SYSTEM.ASSMBLER
SYSTEM.LINKER

...are all files directly accessed by single-letter commands at the system level.

SYSTEM.WRK.TEXT
SYSTEM.WRK.CODE
SYSTEM.WRK.INFO

...are the files which may be present when you use a workfile. The ".TEXT" and ".CODE" suffixes are reserved as well. File types are described in Sections 2.2.6, 1.2.1, and 3.1.

SYSTEM.PASCAL

...contains the operating system, and

SYSTEM.SYNTAX

...all the compiler error messages.

SYSTEM.LIBRARY

... contains previously compiled or assembled routines to be linked in with other programs.

SYSTEM.SWAPDISK

...is a file used as a scratchpad for swapping portions of main memory out to disk during file operations. It contains 2048 bytes.

SYSTEM.LST.TEXT

...is the default file for compiler listing output. It is not always present.

SYSTEM.STARTUP

...is a user-definable file which is understood as a codefile. If it is present on disk, the operating system will execute it automatically at each bootstrap or I(nitialize. This allows you to specify a program to be run before the main command comes up, and any time thereafter by typing "I".

It is the filename itself which causes the system to recognize a file. Therefore, a system file may be changed by simply changing the name. For example, the line oriented editor is shipped as YALOE.CODE. To use YALOE as the system's editor, you can change SYSTEM.EDITOR to BACKUP.EDITOR using the Filer, and then change YALOE.CODE to SYSTEM.EDITOR. After this change, typing "E" at the system level causes YALOE to be executed.

The interpreters have machine-specific names.

SYSTEM.PDP-11

...is the name for the PDP-11 and LSI-11 interpreters. The situation with other processors is more complicated -- refer to the CP/M or Adaptable System documents that are added at the end of this manual.

Any other files on your disks will be user text or program code, utility programs, backup files, or user data. There are two hidden files, transparent to the user: one is the directory, present on every disk, and the other is the bootstrap, present on any disk you use as a system disk. The directory resides at block 2 on the disk, and is 4 blocks long for a single directory, 8 blocks long for a duplicated (backed-up) directory. The bootstrap's location is dependent upon the host machine's hardware, but it is usually in blocks 0 and 1.

The Filer is capable of transferring a whole disk at a time (albeit slowly!). This is important to know, for disk backup is a highly recommended practice. More on this is in Section 1.2 on the Filer, and more on the bootstrap is given in Section A and Section 4.3.

The system lends itself to ease of use. Sections 1.3.1 and 1.3.2 will get you started on the editor, and the tree-structure of the operating system is virtually self-explanatory. It is suggested that newcomers to the system temper their use of this manual with hands-on playing with the system. That is the quickest way to become a fluent user.

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 workfile, 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.

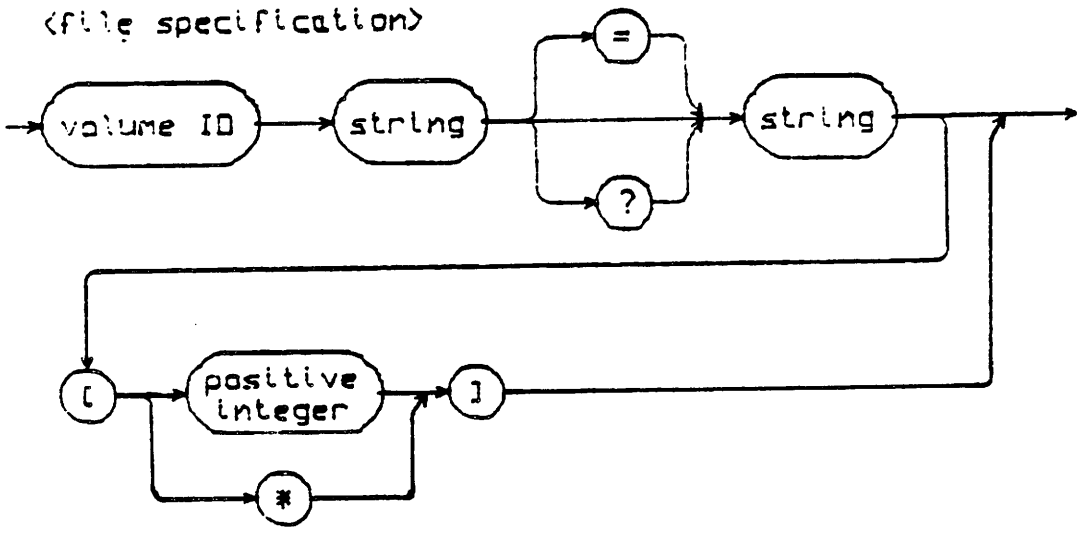


FIGURE 2

FIGURE 2

Volume i.d. syntax can be expanded thusly:

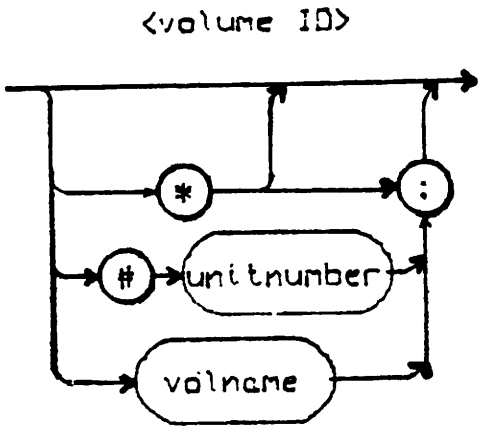


FIGURE 3

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.

WARNING:

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.

EXAMPLE:

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
```

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 ABSURD, 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    38   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    38   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    38   1-Sep-78
FILERDOC1.TEXT    24   1-Sep-78
2/10 files <listed/in-dir>, 62 blocks used, 432 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    38   1-Sep-78
FILERDOC1.TEXT    24   1-Sep-78
2/10 files <listed/in-dir>, 62 blocks used, 432 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, 356 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	494
Northwest Micro	Double-density soft-sectored 8" floppy	1102
Zilog	Single-density hard-sectored 8" floppy	608
North Star	Single-density hard-sectored 5 1/4" floppy	168
DEC	RK05 / per volume	4872

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

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]
```

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.TEXT 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.

MOVING THE CURSOR

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 4.1 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.

USING INSERT

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". (DC1) forces cursor to left margin.

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

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

Figure 3.6

```
WRITE('TOO SMSE ');
```

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

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

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 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.

<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.

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.

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

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

Command arguments are as follows:

n — n stands for any integer. It may be preceded by a + or -. If no sign precedes n, it is assumed to be a positive number. Whenever an argument is acceptable in a command, its absence implies an argument of 1 (or -1 if only the - is present).

m — m is a number 0..9.

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.

*0CAPPLES$$	Replaces the characters from the beginning of the line up to the cursor with 'APPLES', (equivalent to using 0X).
*BGHOSE$=CLIZARD$$	Searches for the first occurrence of 'HOSE' in the Text Buffer and replace it with 'LIZARD'.

The eX(change command exchanges n lines, starting at the cursor, with the indicated text string. The cursor 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 --

********** **********
* PASCAL COMPILER * * Section 1.5 *
********** **********

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 4.1) 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.

1.5.1 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.

F:

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 outputing 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 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 --

********** **********
* THE LINKER * * Section 1.6 *
********** **********

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.

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.

1.6.1 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>

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.

1.6.2 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 3.2)

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.

UCSD ADAPTABLE ASSEMBLER

1. Assembly Language Definition

An assembly language consists of symbolic names which can represent machine instructions, memory addresses, or program data. The main advantage of assembly language programming over machine coding is that programs can be organized in a more readable and hence easier to understand fashion.

An assembly language program (called source code) is translated by an assembler into a sequence of machine instructions (called object code). Assemblers can create either relocatable or absolute object code. Relocatable code includes information that allows a loader to place it in any available area of memory, while absolute code must be loaded into a specific area of memory. Symbolic addresses in programs that are assembled to relocatable object code are called relocatable addresses.

2. Assembly Language Applications

Users of the UCSD Pascal system are interested in developing assembly language programs for one of two purposes:

a) assembly language procedures running under the control of a host Pascal program.

b) stand-alone assembly language programs for use outside of the operating system's environment.

The UCSD Adaptable Assembler, in conjunction with the system linker and some support programs, has been designed to meet these needs. The assembler is adaptable in the sense that different versions built around one adaptable kernel exist for each processor supported by the UCSD Pascal system; new versions can be quickly generated for any new 8 or 16 bit processors that are introduced.

The Adaptable Assembler is a one pass assembler modeled after The Last Assembler (TLA) developed at the University of Waterloo. The basic concept behind both the TLA and the UCSD Adaptable Assembler 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 modules to handle the architecture of each target machine.

This document is intended to be used in conjunction with the processor software manual of the user's machine. For information concerning differences from the processor's standard software syntax, see Section 9 (all section references in this chapter refer to this chapter alone).

GENERAL PROGRAMMING INFORMATION

1. Object Code Format

1.1 Byte Organization

A byte consists of eight bits. The bits may represent eight binary values, or a single character of data. The bits may also represent a one byte machine instruction or a number which is interpreted either as a signed two's complement number in the range of -128 to 127 or an unsigned number in the range of 0 to 255.

1.2 Word Organization

A word consists of sixteen bits, or two adjacent bytes in memory. A word may contain a one word machine instruction, any combination of byte quantities, or a number which may be interpreted either as a signed two's complement number in the range of -32,768 to 32,767 or an unsigned number in the range of 0 to 65,535.

1.3 Memory Organization

1.3.1 Byte Versus Word Addressing

The Adaptable Assembler kernel is designed to accommodate byte addressed, word addressed, and byte addressed/word oriented processors - the instructions and data words of the latter two processor types are constrained to word boundaries. A word boundary on a word oriented processor is defined as an even byte address.

Word alignment is enforced throughout word addressed versions of the assembler by defining all data directives to emit integral numbers of words. With word oriented assemblers, the programmer is responsible for maintaining word alignment of instructions and data words; failure to do so will be flagged with an error message. Nonalignment occurs when a directive creates an odd number of data bytes.

1.3.2 Byte sex

The two bytes that make up a 16 bit word are termed the least significant and most significant byte, or LSB and MSB respectively. Unfortunately, the various manufacturers of processors have different conventions for whether the first (or lower addressed) byte of a word is used as the LSB or the MSB; hence, the 'byte sex' problem has arisen. See Section 9 to determine the byte sex of processors supported with Adaptable Assembler versions.

The UCSD Pascal system is designed to handle most byte sex conflicts; however, the user cannot encounter such problems unless software is moved between machines with different byte ordering.

2. Source Code Format

2.1 Character Set

The following characters are used to construct source code:

- upper case alphabetic: ('A'...'Z')
- numerals: '0'...'9'
- special symbols: | @ # $ % ^ & * ( ) < > ~ [ ] . , / ; : " ' + - = ? _
- space (' ') character and tab character

2.2 Identifiers

Identifiers consist of an alphabetic character followed by a series of alphanumeric characters and/or underscore characters. Unlike identifiers in UCSD Pascal programs, the underscore character is significant.

Identifiers are used in:

- label and constant definitions.
- machine instructions, assembler directives, and macro identifiers.
- label and constant references.

2.2.1 Predefined Symbols and Identifiers

Predefined identifiers are reserved by the assembler as symbolic names for machine instructions and registers. They may not be used as names for labels, constants, or procedures. Each assembler also has a predefined location counter symbol. This is a character which, when used in an expression, represents the current value of the location counter in the program. Section 9 lists the location counter character for each assembler version.

2.3 Character Strings

A character string is written as a series of ASCII characters delimited by double quotes. A string may contain up to eighty characters, but cannot cross source lines. The assembler directive .ASCII requires a character string as its operand. Strings also have limited uses in expressions.

2.4 Constants

2.4.1 Decimal Integer Constants

A decimal integer word constant is written as a series of numerals (0..9) followed by a period. Its range of values is -32768 to 32767 as a signed two's complement number. As a byte constant, its range of values is -128 to 127 as a signed two's complement number or 0 to 255 as an unsigned number.

2.4.2 Hexadecimal Integer Constants

A hexadecimal integer word constant is written as a series of up to four significant hexadecimal numerals (0..9, A..F) followed by the letter 'H'. The leading numeral of a hex constant must be a numeric character. The range of values is 0 to FFFF. These are examples of valid hex constants:

```
0AH
100H
0FFFEH          ; leading zero is required here
```

Byte constants posess similar syntax, but can have at most two significant hex numerals, with a range of 0 to FF.

2.4.3 Octal Integer Constants

An octal integer word constant is written as a series of up to six significant octal numerals (0..7) followed by the letter 'O'. Its range of values is 0 to 177777. Byte constants can have at most three significant octal numerals, with a range of 0 to ~~477~~ 377.

2.4.4 Default Radix Integer Constants

The radix of an integer constant lacking a trailing radix character is set to the assembler's current default radix. Initial default radices for all assemblers are listed in Section 9.

2.4.5 Character Constants

Character constants are special cases of character strings and may be used in expressions. The maximum length is two characters for a word constant, and one character for a byte constant.

2.4.6 Assembly time Constants

An assembly time constant is written as an identifier that has been assigned a constant value by the .EQU directive (Section 3.2). Its value is completely determined at assembly time from the expression following the directive. Assembly time constants must be defined before they may be referenced.

2.5 Expressions

Expressions are used as symbolic operands for machine instructions and assembler directives. An expression can be:

- a label, which might refer to a defined address or an address further down in the source code (implying that the label is presently undefined), an externally referenced address, or an absolute address.
- a constant.

- a series of labels or constants separated by arithmetic or logical operators.

- the null expression, which evaluates to a constant of value 0.

2.5.1 Relocatable and Absolute Expressions

An expression containing more than one relocatable label is valid only if it posesses the form: label1 - label2. All other expressions containing more than one relocatable label are disallowed. Subexpressions that evaluate to relocatable quantities may not be used as arguments to a multiplication, division, or logical operator. Unary operators may not be applied to relocatable quantities.

2.5.2 Linking and One Pass Restrictions

An expression may contain no more than one externally defined label, and its value must be added to the expression. An expression containing an external reference may not contain any additional relocatable labels.

An expression may contain no more than one forward referenced identifier. A forward referenced identifier is assumed to be a label defined further down in the source code; any other identifiers must be defined before they are used in an expression.

2.5.3 Arithmetic & Logical Operators

The following operators are available for use in expressions:

unary operations:
- '+' plus
- '-' minus (two's complement negation)
- '~' logical not (one's complement negation)

binary operations:
- '+' plus
- '-' minus
- '^' exclusive or
- '*' multiplication
- '/' truncating division (DIV)
- '%' remainder division (MOD)
- '|' bit wise OR

'&' bit wise AND

The following operators are available for use only with conditional assembly directives:

'=' equal
'<>' not equal

The assembler performs left to right evaluation of expressions; there is no operator precedence. All operations are performed on word quantities. Usage of unary operators is limited to constants and absolute addresses. Angle brackets must enclose subexpressions which contain embedded unary operators.

2.5.4 Subexpression Grouping

Angle brackets ('<' and '>') may be used in expressions to override the left to right evaluation of operands. Subexpressions enclosed in angle brackets are completely evaluated before inclusion in the rest of the expression.

2.5.5 Examples

The following are examples of valid expressions. The default radix is decimal.

```
MARK+4          ; The sum of the value of identifier MARK plus 4

BILL-2          ; The result of subtracting 2 from the value
                ; of identifier BILL.

2-BARRY         ; The result of subtracting the value of
                ; identifier BARRY from 2.  BARRY must be
                ; absolute.

3*2+MACRO       ; The sum of the value of identifier MACRO plus
                ; the product of 3 times 2.

DAVID+3*2       ; 2 times the sum of the identifier DAVID
                ; and 3.  DAVID must be absolute.

650/2-RICH      ; The result of dividing 650 by 2 and sub-
                ; tracting the value of identifier RICH from
                ; the quotient.  RICH must be absolute.

                ; Null expression - result is constant 0

-4*12+<6/2>     ; evaluates to -45 (decimal)

85+2+<-5>       ; evaluates to 82 (decimal)
```

```
0|1&<~0>      ; evaluates to 1
```

3. Source Statement Format

An assembly language source program consists of source statements which may contain machine instructions, assembler directives, comments, or nothing (a blank line). Each source statement is defined as one line of a textfile. Assembly language identifiers are restricted to upper case alphabetic characters, but lower case characters may be used in the comment field.

3.1 Label Field

The assembler supports the use of both standard labels and local (i.e., reuseable) labels. The label field begins in the leftmost character position of each source line. Macro identifiers and machine instructions must not appear in the start of the label field, but assembler directives and comments may appear there.

3.1.1 Standard label usage

A standard label is an identifier that appears in the label field of a source statement. It may be terminated by an optional colon character, which is not used when referencing the label. As in Pascal, only the first eight characters of the label are important; the rest are ignored by the assembler. Unlike Pascal, the underscore character is significant.

EXAMPLE:

```
BIOS
L3456:          ; referenced as 'L3456'
THE_KIND
LONG_LABEL      ; last two characters are ignored
```

A standard label is a symbolic name for a unique address or constant; it may be declared only once in a source program. A label is optional for machine instructions and for many of the assembler directives. A source statement consisting of only a label is a valid statement; it has the effect of assigning the current value of the location counter to the label. This is equivalent to placing the label in the label field of the next source statement that generates object code. Labels defined in the label field of the .EQU directive (Section 3.2) are assigned the value of the expression in the operand field.

3.1.2 Local Label Usage

Local labels allow source statements to be labeled for reference by other instructions without taking up storage space in the symbol table. They contribute to the conceptual cleanliness of source program design by allowing the creation of nonmnemonic labels for use by iterative and decision constructs, thus emphasizing the use of mnemonic label names for delimiting the conceptually more important sections of code.

Local labels must have "$" in the first character position; the remaining characters must be digits. As in regular labels, only the first eight digits are significant. The scope of a local label is limited to the lines of source statements between the declaration of consecutive standard labels; thus, the jump to label $4 in the following example is illegal:

```
LABEL1
        LDA     3
$3      STA     4
        JP      NZ,$3 ; legal use of local label
        NOP
        JP      $4    ; illegal use
LABEL2
        LDA     5
$4      STA     6
```

Up to 21 local labels may be defined between 2 occurrences of a standard label. On encountering a standard label, the assembler purges all existing local label definitions; hence, all local label names may be redefined after that point. Local labels may not be used in the label field of the .EQU directive (Section 3.2).

3.2 Opcode Field

The opcode field begins with the first non-blank character following the label field, or with the first nonblank character following the leftmost character position when the label is omitted. It is terminated by one or more blanks. The opcode field contains an identifier which can be of the following types:

- machine instruction
- assembler directive
- macro call

3.3 Operand Field

The operand field begins with the first nonblank character following the opcode field, and is terminated by zero or more blanks. It can contain zero or more expressions, depending on the requirements of the preceding opcode.

3.4 Comment Field

The comment field can be preceded by zero or more blanks, begins with a semicolon (';'), and extends to the end of the current source line. It may contain any printable ASCII characters. The comment field is listed on assembled listings, and has no other effect on the assembly process.

4. Source File Format

Assembly source files are generated using the system editor and saved as files of type TEXT. A source file is constructed from the following entities:

- assembly routines (procedures and functions).
- global declarations.

4.1 Assembly Routines

A source file may contain more than one assembly routine; in this case, a routine ends upon the occurrence in the source code of another program delimiting directive (i.e., the start of the following routine). Each routine in a source file is a separate entity; it contains its own relocation information and may be individually referenced by a Pascal host program during linking.

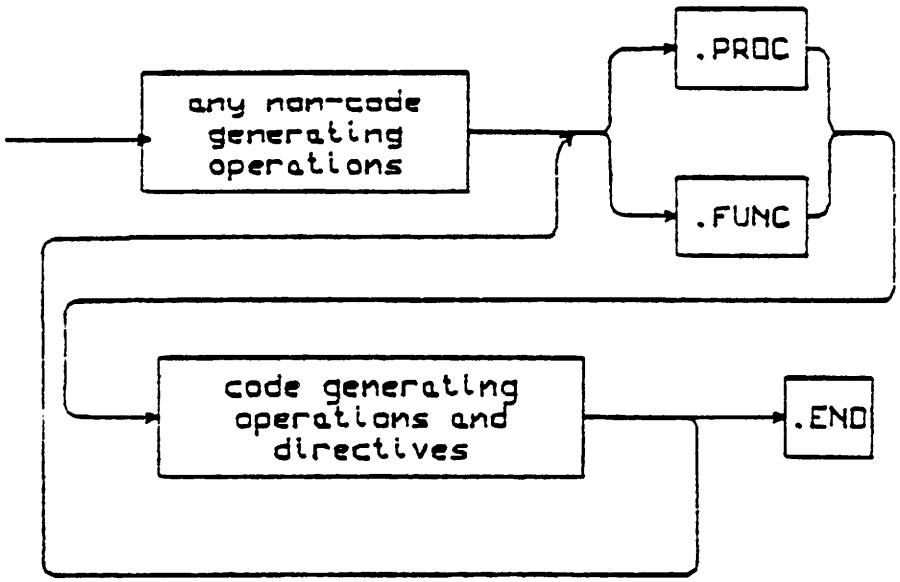
Assembly routines must begin with a .PROC or .FUNC directive. The last routine in the source file must be terminated by the .END directive. Section 6 gives a detailed description of these directives.

At the end of each routine, the assembler's symbol table is cleared of all but predefined and globally declared symbols, and the location counter (LC) is reset to zero.

4.2 Global declarations

An assembly routine may not directly access objects declared in another assembly routine, even if the routines are assembled in the same source file; however, occasions arise when it is desirable for a set of routines to share a commmon group of declarations. Therefore, the assembler allows global data declarations.

Any objects declared before the first occurrence of a .PROC or .FUNC directive in a source file may be referenced by all subsequent assembly routines. No code may be generated before the first procedure delimiting directive; hence, the 'global' objects are limited to the non-code-generating directives (.EQU, .REF, .DEF, .MACRO, .LIST, etc.).


any non-code generating operations
.PROC
.FUNC
code generating operations and directives
.END

ASSEMBLER DIRECTIVES

Assembler directives (sometimes referred to as pseudo-ops) enable the programmer to supply data to be included in the program and exercise control over the assembly process. The following directives are common to all assembler versions. Assembler directives appear in the source code as predefined identifiers preceded by a period (.).

The following metasymbols are used below in the syntax definitions for assembler directives:

- special characters and items in capital letters must be entered as shown.
- items within angle brackets (<>) are defined by the user.
- items within square brackets ([]) are optional.
- the word 'or' indicates a choice between two items.
- items in lower case letters are generic names for classes of items.

The following terms are names for classes of items:

b =
the occurrence of one or more blanks.

integer =
any legal integer constant as defined in Section 2.2.4.

label =
any legal label as defined in Section 2.3.1.

expression =
any legal expression as defined in Section 2.2.5.

value =
any label, constant, or expression. Its default value is 0.

valuelist =
a list of zero or more values delimited by commas.

identifier =
a legal identifier as defined in Section 2.2.2.

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

id:integer list =
a list of one or more identifier-integer pairs separated by a colon and delimited by a comma. The colon:integer part is optional; its default value is 1.

comment =
any legal comment as defined in Section 2.3.4.

character string =
any legal character string as defined in Section 2.2.3.

file identifier =
any legal name for a Pascal text file.

Example:

[<label>] [b] .ASCII b <character string> [<comment>]

... indicates that a label may be included in the label field (but is not necessary), and that a character string must be included as an operand.

Small examples are included after each definition to supply the user with a reference to the specific syntax of the directive.

1. Procedure Delimiting Directives

Every source program (including those intended for use as stand-alone code files) must contain at least one set of procedure delimiting directives. The most frequent use of the assembler is in assembling small routines intended to be linked with a Pascal host. In this case, the .PROC and .FUNC directives are used to identify and delimit an assembly procedure for eventual use as a Pascal external procedure or function. Section 6 has a more detailed description of this context.

.PROC Identifies the beginning of an assembly language procedure. The procedure is terminated by the occurrence of another delimiting directive in the source file.

FORM: [b] .PROC b <identifier> [,<integer>] [<comment>]

<identifier> is the name associated with the assembly procedure.

<integer> indicates the number of words of parameters passed to this routine. The default is 0.

EXAMPLE: .PROC DLDRIVE,2

.FUNC Identifies the beginning of an assembly language function which is expected by the Pascal host program to return a function result on the stack; otherwise, equivalent to the .PROC directive.

FORM: [b] .FUNC <identifier>[,<integer>] [<comment>]

<identifier> is the name associated with the assembly procedure.

<integer> indicates the number of words of parameters passed to this routine. The default is 0.

EXAMPLE: .FUNC RANDOM

.END Marks the end of an assembly source file.

FORM: [<label>] [b] .END

2. Data and Constant Definition Directives

.ASCII Converts character strings to a series of ASCII byte constants in memory. The bytes are allocated in the order that they appear in the string. An identifier in the label field is assigned the location of the first character allocated in memory. If an odd number of data bytes are generated on a word addressed assembler version, an extra zero byte will be emitted to word align the LC.

FORM: [<label>] [b] .ASCII b <character string> [<comment>]

<character string> is any string of printable ASCII characters delimited by double quotes.

EXAMPLE: .ASCII "HELLO"

.BYTE Allocates and initializes values in one or more bytes of memory. Values must be absolute byte quantities. The default value is zero. An identifier in the label field is assigned the location of the first byte allocated in memory.

FORM: [<label>] [b] .BYTE b [valuelist] [<comment>]

```
EXAMPLE:  TEMP .BYTE    4    ; code would be:  04 hex

          TEMP1 .BYTE        ; code would be:  00 hex
```

.BLOCK Allocates and initializes a block of consecutive bytes/words in memory (bytes for byte addressed processors, words for word addressed processors). A byte value must be an absolute quantity. The default value is zero. An identifier in the label field is assigned the location of the first byte/word allocated.

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

<length> is the the number of bytes to allocate with the initial value <value>.

EXAMPLE:
```
TEMP .BLOCK   4,6H
```

the output code would be:
```
    06 06 06 06     ;four bytes with value 06 hex
```

.WORD Allocates and initializes values in one or more consecutive words of memory. Values may be relocatable quantities. The default value is zero. An identifier in the label field is assigned the location of the first word allocated.

FORM: [<label>] [b] .WORD b <valuelist> [<comment>]

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

the output code would be:
```
     0000
     0002
     0000          ; this is a default value.
     0004

L1   .WORD  L2
```

the output code would be a word containing the address of the label L2.

.EQU Equates a value to a label. Labels may be equated to an expression containing relocatable labels, externally referenced labels, and/or absolute constants. The general rule is that labels equated to values must be defined before use. The exception to this rule is for labels equated to expressions containing another label. Local labels may not appear in the label field of an equate statement.

FORM: <label> [b] .EQU b <value> [<comment>]

EXAMPLE:
```
BASE    .EQU    R6
```

3. Location Counter Modification Directives

These directives affect the value of the location counter (LC or ALC) and the location in memory of the code being generated.

.ORG — If used at the beginning of an absolute assembly program, .ORG initializes the location counter to <value>. Used anywhere else, .ORG will generate zero bytes until the value of the location counter equals <value>.

FORM: [b] .ORG b <value> [<comment>]

EXAMPLE:
```
.ORG    1000H
```

.ALIGN — Outputs sufficient zero bytes/words to set the location counter to a value which is a multiple of the operand value (bytes are emitted for byte addressed processors, words are emitted for word addressed processors).

FORM: [b] .ALIGN b <value> [<comment>]

EXAMPLE:
```
.ALIGN  2
```

On a byte addressed processor, this would align the LC on a word boundary.

4. Listing Control Directives

These directives allow the user to exercise control over the format of the assembled listing file generated by the assembler. No code is generated by these directives, and their source lines do not appear on assembled listings. See Section 8 for a more detailed description of an assembled listing.

.TITLE Changes the title printed on the top of each page of the assembled listing. The title may be up to 80 characters long. The assembler will change the title to 'SYMBOLTABLE DUMP' when printing a symbol table; the title reverts back to its former value after the symbol table is printed. The default value for the title is ' '.

FORM: [b] .TITLE b <character string> [<comment>]

EXAMPLE: .TITLE "P-CODE INTERPRETER"

.PAGE Continue the assembled listing on the next page by sending an ASCII form feed character to the assembled listing.

FORM: [b] .PAGE

EXAMPLE: .PAGE

.LIST Enables output to the list file, if a listing is not already being generated. .LIST and .NOLIST can be used to examine certain sections of source and object code without creating an assembled listing of the entire program. Assembly begins with an implicit .LIST directive.

FORM: [b] .LIST

EXAMPLE: .LIST

.NOLIST Suppresses output to the list file, if it is not already off.

FORM: [b] .NOLIST

EXAMPLE: .NOLIST

.MACROLIST Specifies that all following macro definitions will have their macro bodies printed when they are invoked in the source program. Assembly begins with an implicit .MACROLIST directive. Section 5 has a detailed description of macro language.

FORM: [b] .MACROLIST

EXAMPLE: .MACROLIST

.NOMACROLIST Specifies that all following macro definitions will not have their macro bodies printed when they are invoked in the source program. Only the macro identifier and parameter list are included in the listing.

FORM: [b] .NOMACROLIST

EXAMPLE: .NOMACROLIST

.PATCHLIST List occurences of all back patches of forward referenced labels in the list file. Assembly begins with an implicit .PATCHLIST directive. Section 8 has a detailed description of back patches.

FORM: [b] .PATCHLIST

EXAMPLE: .PATCHLIST

.NOPATCHLIST Suppress the listing of back patches of forward references.

FORM: [b] .NOPATCHLIST

EXAMPLE: .NOPATCHLIST

5. Program Linkage Directives

Linking directives enable communication between separately assembled and/or compiled programs. Section 6 has a detailed description of program linking.

.CONST Allows access to globally declared constants in the PASCAL host program by the assembly procedure.

FORM: [b] .CONST <idlist> [<comment>]

Each <id> is the name of a global constant declared in the Pascal host.

EXAMPLE: .CONST LENGTH

.PUBLIC Allows variables declared in the global data segment of the PASCAL host program to be referenced by an assembly language routine.

FORM: [b] .PUBLIC <idlist> [<comment>]

Each <id> is the name of a global variable declared in the Pascal host.

EXAMPLE: .PUBLIC I,J,LENGTH

.PRIVATE Allows an assembly language routine to store variables in the global data segment of the host program that are accessable only to the assembly language routine.

FORM: [b] .PRIVATE <id:integer list> [<comment>]

EXAMPLE: .PRIVATE PRINT,BARRAY:9

Each <id> is treated as a label defined in the source code. <integer> determines the number of words of space allocated for <id>.

.INTERP Allows an assembly language procedure to access code or data in the P-code interpreter. .INTERP is a predefined symbol for a processor dependent location in the resident interpreter code; offsets from this base location may be used to access any code in the interpreter. Correct usage of this feature requires a knowledge of the interpreter's jump vector for this location. Its domain is generally restricted to systems applications.

FORM: valid when used in <expression>

EXAMPLE:

```
EXECERR  .EQU  12        ; hypothetical routine offset

BOMBINT  .EQU  .INTERP+EXECERR
         JMP   BOMBINT
```

.REF Provides access to one or more labels defined in other assembly language routines.

FORM: [b] .REF <idlist> [<comment>]

EXAMPLE:

```
.REF   SCHLUMP
```

.DEF Makes one or more labels to be defined in the current routine available to other assembly language routines for reference.

FORM: [b] .DEF <idlist> [<comment>]

EXAMPLE:

```
.DEF   FOON,YEEN
```

6. Conditional Assembly Directives

Section 4 has a detailed description of conditional assembly features.

.IF Marks the start of a conditional section of source statements.

FORM: [b] .IF <expression> [= or <> <expression>] [<comment>]

EXAMPLE: .IF Z80

.ENDC Marks the end of a conditional section of source statements.

FORM: [b] .ENDC [<comment>]

EXAMPLE: .ENDC

.ELSE Marks the start of an alternative section of source statements.

FORM: [b] .ELSE [<comment>]

EXAMPLE: .ELSE

7. Macro definition directives

Section 5 has a detailed description of macro language.

.MACRO Indicates the start of a macro definition

FORM: [b] .MACRO <identifier> [<comment>]

<identifier> is used to invoke the macro being defined.

EXAMPLE: .MACRO ADDWORDS

.ENDM Marks the end of a macro definition.

FORM: [b] .ENDM [<comment>]

EXAMPLE: .ENDM

8. Miscellaneous Directives

.INCLUDE Causes the assembler to start assembling the file named as an argument of the directive; when the end of this file is reached, assembling resumes with the source code that follows the directive in the original file. This feature is useful for including a file of macro definitions or for splitting up a source program too large to be edited as a single text file. .INCLUDE may not be used in an included source file (i.e., nested use of the directive) and may not be used in a macro definition.

FORM: [b] .INCLUDE <file identifier>

The comment field of the .INCLUDE directive must remain empty!

EXAMPLE: .INCLUDE MYDISK:MACROS

.ABSOLUTE Causes the following assembly routine to be assembled without relocation information. Labels become absolute addresses and label arithmetic is allowed in expressions. Usage is valid only before the occurrence of the first procedure delimiting directive. .ABSOLUTE must not be used when creating a Pascal external procedure. Section 6 has a detailed description of absolute code files.

FORM: [b] .ABSOLUTE [<comment>]

EXAMPLE: .ABSOLUTE

SECTION 1.7.4

CONDITIONAL ASSEMBLY

Conditional assembly directives are used to selectively exclude or include sections of source code at assembly time. Conditional sections are initiated with the .IF directive and terminated with the .ENDC directive, and may contain the .ELSE directive. Control over the inclusion of conditional sections is determined by the use of conditional expressions. Conditional sections may contain other conditional sections.

When the assembler encounters an .IF directive, it evaluates the associated expression to determine the condition value. If the condition value is false, the source statements following the directive are discarded until a matching .ENDC or .ELSE is reached. If the .ELSE directive is used in a conditional section, source code before the .ELSE is assembled if the condition is true, and source code after the .ELSE is assembled if the condition is false.

Overall syntax for a conditional section (using the metalanguage described in Section 3) is as follows:

```
.IF <conditional expression>
<source statements>
[.ELSE
<source statements>]
.ENDC
```

1. Conditional Expressions

A conditional expression can take one of two forms: a single expression, or comparison of two character strings or expressions. The first form is considered false if it evaluates to zero; otherwise, it is considered true. The second form of conditional expression is comparison for equality or inequality (indicated by the symbols '=' and '<>', respectively).

2. Example

```
.IF     LABEL1-LABEL2   ; arithmetic expression

    ; This code is assembled only if
    ; difference is zero

  .IF     "%1" = "STUFF"  ; comparison expression

      ; This code is assembled only if outer condition
      ; is true and text of first macro parameter
      ; is equal to 'STUFF'.

  .ENDC                   ; terminate nested section

    ; This code is assembled if outer condition
    ; is true

.ELSE

    ; This code is assembled if first condition
    ; is false

.ENDC                   ; terminate outer section
```

MACRO LANGUAGE

The assembler supports the use of a macro language in source programs. A macro language allows the programmer to associate a set of source statements with an identifying symbol; when the assembler encounters this symbol (known as a macro identifier) in the source code, it substitutes the corresponding set of source statements (known as the macro body) for the macro identifier, and assembles the macro body as if it had been included directly in the source program. A carefully designed set of macro definitions can be used in all source programs to simplify the development of assembly language routines.

Macro language is enhanced by including a mechanism for passing parameters (known as macro parameters) to the macro body while it is being expanding, allowing a single macro definition to be used for an entire class of subtasks.

Here is a simple example:

```
                        ; macro definition...
.MACRO  STRING          ; macro identifier is STRING
                        ; macro body
                        ; %1 and %2 are parameter declarations
.BYTE   %2              ; 2nd parameter is length byte
.ASCII  "%1"            ; 1st parameter is string argument
.ENDM                   ; end macro definition
```

Further down in the source code...

```
STRING  WRITE,5.        ; 1st macro call
                        ; parameters are 'WRITE' and '5.'
STRING  TYPE SPACE,10.  ; 2nd macro call
                        ; parameters are 'TYPE SPACE' and '10.'
```

This is what gets assembled...

```
.BYTE   5.              ; data string declarations
.ASCII  "WRITE"

.BYTE   10.
.ASCII  "TYPE SPACE"
```

1. Macro Definitions

Macro definitions may occur anywhere in a source program and are delimited by the directives .MACRO and .ENDM. The macro identifier must be unique to the source program, except when the programmer is redefining a predefined machine instruction name as a macro identifier. A macro definition may not include another macro definition; however, it may include macro calls. Macro calls may be nested to a maximum depth of five levels. A macro definition must occur before any calls to that macro are assembled, but macro calls may be forward referenced within the bodies of other macro definitions.

2. Macro Calls

Macro calls may occur anywhere in a source program that code may be generated. A macro call consists of a macro identifier followed by a list of parameters. The parameters are delimited by commas and terminated by a carriage return or semicolon. Upon encountering a macro call, source code is read from the text of the corresponding macro body. Macro parameters within the macro body are substituted with the text of the matching parameter listed after the macro identifier which initiated the call.

3. Parameter Passing

Macro parameters are referenced in a macro body by using the symbol '%n' in an expression, where this symbol, the assembler replaces it with the text of the n'th macro parameter.
Three cases are possible:

1) The parameter exists - make the substitution.

2) The n'th parameter doesn't exist in the parameter list being checked (less than n parameters were passed); a null string is substituted.

3) Another symbol of the form '%m' is encountered in the parameter list. If nested macro calls exist, the text of the m'th parameter at the next higher level of macro nesting is substituted; otherwise, the symbol itself is assembled.

Parameters are passed without leading and trailing blanks. All assembly symbols except macro calls may be passed as parameters.

The following is an example of parameter passing in macros:

```
.MACRO DOS
UNO    %2,UN
SRL    %2
.ENDM

.MACRO  UNO
MOV     %1,%2
SLA     %3
.ENDM
```

In a program, the macro call...

```
DOS   TROIS,DEUX
```

assembles as...

```
MOV    DEUX,UN        ; UNO got UN directly, but had to
                      ; use DOS's 2nd param
SLA                   ; 3rd param doesn't exist
SRL    DEUX           ; DOS used its own 2nd param
```

4. Scope Of Labels In Macros

A problem arises in the use of macro language when the definition of a macro body requires the use of branch instructions and thus the presence of labels. Declaring a regular label in a macro body is incorrect if the macro is called more than once, for the label would be substituted twice into the source program and flagged by the assembler as a previously defined label. Location-counter-relative addressing can be used, but is prone to errors in nontrivial applications. The solution is to generate labels that are local to the macro body; the assembler's local labels have this capability.

Local label names declared in a macro body are local to that macro; thus, a section of code that contains a local label $1 and a macro call whose body also has the local label $1 will assemble without errors (contrast this with what happens when two occurrences of $1 fall between two regular labels). This feature allows local labels to be used freely in macros without fear of conflicts with the rest of the program.

Note - the maximum of 21 local labels active at any instant still applies.

4.1 Local Labels As Macro Parameters

The passing of local labels as parameters has a special property. Unlike other macro parameters, local labels are not passed as uninterpreted text. The scope of a local label passed in a macro call does not change as it is passed through increasing levels of macro nesting, regardless of naming conflicts along the way. One use of this property is passing an address to a macro which simulates a conditional branch instruction.

The following is an example of passing local labels as macro parameters:

```
    .MACRO EIN
    BEQ    $1
    BNE    %1
 $1
    .ENDM
```

In a program, the code...

```
 TWIE
    MOV    ICHI,NI
    EIN    $1
    RTS
 $1
```

```
   JSR   SAN

assembles as...

TWIE
   MOV   ICHI,NI        ; looks confusing, but if listing
                        ; was off, result is what programmer
                        ; meant to occur
   BEQ   $1             ; this references macro local label
   BNE   $1             ; this references outside $1
$1                      ; macro local label

   RTS
$1                      ; outside $1
   JSR   SAN
```

SECTION 1.7.6

PROGRAM LINKING AND RELOCATION

The Adaptable Assembler produces either absolute or relocatable object code that may be linked as required to create executable programs from separately assembled or compiled modules.

Program linking directives generate information required by the system linker to link modules. Some of the advantages of linking are:

Long programs can be divided into separately assembled modules to avoid a long assembly, reduce the symbol table size, and encourage modular programming techniques.

Modules can be shared by other linked modules.

Utility modules can be added to the system library for use as external procedures by a large number of programs.

Pascal programs can directly call assembly language procedures.

The assembler generates linker information in both relocatable and absolute code files. The system linker accesses this information during the linking process and removes it from the linked code file.

Relocatable code includes information that allows a loader program to place it anywhere in memory, while absolute (also called core image) code files must be loaded into a specific area of memory to execute properly. Assembly procedures running in the Pascal system environment are always relocatable; the loading and relocation process is performed by the interpreter at a load address determined by the system state.

1. Program Linking Directives

This section describes overall usage of linking directives. All linking of assembly procedures involves word quantities; it is not possible to externally define and reference data bytes or assembly time constants. Arguments of these directives must match the corresponding name in the target module (a lower case Pascal identifier will match an upper case assembly name) and must not have been used before their appearance in the directive; all following references to the arguments are treated by the assembler as special cases of labels. These external references are resolved by the linker and/or interpreter by adding the link time and run time offsets to the existing value of the word quantity in question; thus, any initial offsets generated by the inclusion of external references and constants in expressions are preserved.

1.1 Pascal Host Communication Directives

The directives .CONST, .PUBLIC, and .PRIVATE allow the sharing of constants and data between an assembly procedure and its Pascal host program. See 6.2.1 for examples.

.CONST Allows an assembly procedure to access constants declared in the global data segment of the Pascal host program. All references to arguments of .CONST are patched by the linker with a word containing the value of the host's compile time constant.

.PUBLIC Allows an assembly procedure to access variables declared in the global data segment of the Pascal host program. Note - this directive can be used to set up pointers to the start of multi-word variables in the host programs; it is not limited to single word variables.

.PRIVATE Allows an assembly procedure to declare variables in the global data segment of the pascal host program that are inaccessable to the host. The optional length attribute of the arguments allows multi-word data spaces to be allocated; the default data space is one word.

1.2 External Reference Directives

The directives .REF and .DEF allow separately assembled modules to share data space and subroutines. See 6.2.2 for examples.

.DEF declares a label to be defined in the current program as accessable to other modules. One restriction is imposed on usage - it is invalid to .DEF a label that has been equated to a constant expression or an expression containing a .REF'ed label.

.REF declares a label existing and .DEF'ed in another module to be accessable to the current program.

1.3 Program Identifier Directives

The directives .PROC, .FUNC, and .END serve as delimiters for source programs. Every source program (relocatable or absolute) must contain at least one pair of delimiting directives (see Section 2.4.1).

The identifier argument of the .PROC directive serves two functions: it is referenced by the linker when linking an assembly procedure to its corresponding Pascal host program, and it can be referenced as an externally declared label by other modules. Specifically, the declaration:

```
        .PROC  FOON      ; procedure heading
```

... in a source program is functionally equivalent in the assembly environment to the following statements:

```
        .DEF   FOON      ; FOON may be externally referenced
FOON                     ; declare FOON as a label
```

This feature allows an assembly module to call other (external and eventually linked in) assembly modules by name. The .FUNC directive is used when linking an assembly function directly to a Pascal host program; it is not intended for uses which involve linking with other assembly modules.

The optional integer argument after the procedure identifier is referenced by the linker to determine if the number of words of parameters passed by the Pascal host's external procedure declaration matches the number specified by the assembly procedure declaration; it is not relevant when linking with other assembly modules.

2. Linking Program Modules

2.1 Linking With A Pascal Host Program

External procedures and functions are assembly language routines declared in Pascal programs. In order to run Pascal programs with external declarations, it is necessary to compile the Pascal program, assemble the external procedure or function, and link the two code files. The linking process can be simplified by adding the assembled routine to the system library with the librarian program.

A host program declares a procedure to be external in a syntactically similar manner to a forward declaration. The procedure heading is given (with parameter list, if any), followed by the keyword 'EXTERNAL'. Calls to the external procedure use standard Pascal syntax, and the compiler checks that calls to the external procedure agree in type and number of parameters with the external declaration. All parameters are pushed on the stack in the order of their appearance in the parameter list of the declaration; thus, the rightmost parameter in the declaration will be on the top of stack. Section 6.2.1.1 has a detailed description of parameter passing conventions.

It is the programmer's responsibility to assure that the assembly language routine maintains the integrity of the stack. This includes removing all parameters passed from the host, preserving any machine resources in use by the interpreter , and making a clean return to the Pascal run time environment using the return address originally passed to it. The price of nonconformance in these matters is a potentially fatal system crash, as assembly routines are outside the scope of the Pascal environment's run time error facilities. Section 9 has a detailed description of Pascal/assembly language protocols for all machines.

An external function is similar to that of a procedure, but with some differences that affect the way in which parameters are passed to and from the Pascal run time environment; first, the external function call will push two words on the stack after all parameters have been pushed. The two words are part of the P-machine's function calling mechanism, and are irrelevant to assembly language functions; the assembly routine must throw these away to get at the function parameters. Secondly, the assembly routine must push the proper number of words (2 for type real, 1 otherwise) containing the function result onto the stack before passing control back to the host.

2.1.1 Parameter Passing Conventions

The ability of external procedures to pass any variables as parameters gives the assembly programmer complete freedom to access the machine dependent representations of machine independent Pascal data structures; however, with this freedom comes the responsibility of respecting the integrity of the Pascal run time environment. This section attempts to enumerate the P-machine's parameter passing conventions for all data types in order that the programmer may gain a better understanding of the Pascal/assembly language interface; it does not actually describe data representations. Machine dependent data representations are described in another section of the user manual.

Parameters may be passed either by value or by name (also known as variable parameters). For purposes of assembly language manipulation, variable parameters are handled in a more straightforward fashion than value parameters.

The word 'tos' is used in the following sections as an abbreviation for 'top of stack'.

2.1.1.1 Variable Parameters

Variable parameters are referenced through a one word pointer passed to the procedure. Thus, the procedure declaration:

```
procedure pass_by_name(var i,j : integer; var q : some_type);
external;
```

...would pass 3 one word pointers on the stack; tos would be a pointer to q, followed by pointers to j and i.

A Pascal external procedure declaration is allowed to contain variable parameters lacking the usual type declaration; this enables variables of different Pascal types to be passed through a single parameter to an assembly routine. Untyped parameters are not allowed in normal Pascal procedure declarations.

The procedure declaration:

```
procedure untyped_var(var i; var q : some_type);
external;
```

...contains the untyped parameter i.

2.1.1.2 Value Parameters

Value parameters are handled in a manner dependent upon their data type. The following types are passed by pushing copies of their current values directly on the stack: boolean, char, integer, real, subrange, scalar, pointer, set, and long integer. Other sections of the user manual describe the number of words per data type and the internal data format. For instance, the declaration:

```
procedure pass_by_value(i : integer; r : real); external;
```

...would pass 2 words on tos containing the value of the real variable r followed by one word containing the value of the integer variable i.

Variables of type record, array, and string are passed by value in the same manner as variable parameters; pointers to the actual variable are pushed onto the stack. Pascal procedures protect the original variables by using the passed pointer to copy their values into a local data space for processing; assembly procedures should respect this convention and not alter the contents of the original variables.

2.2 Example Of Linking To Pascal Host

Note that in the following example the host program passes control to the beginning of an assembly procedure whether or not machine instructions are present there; therefore, all data sections allocated in the procedure must either occur after the end of the machine instructions or have a jump instruction branch around them.

```
PROGRAM EXAMPLE;            { Pascal host program }
const size = 80;
var   i,j,k : integer;
      lst1 : array [0..9] of char;
      { PRT and LST2 get allocated here }

  procedure do_nothing; external;
  function null_func(xxyxx,z : integer) : integer; external;

begin
  k := 45;
  do_nothing;
  j := null_func(k,size);
end.
```

```
        .PROC    DONOTHING ; underscores are not significant
                           ; in Pascal

        .CONST   SIZE      ; can get at size constant in host...
        .PUBLIC  I,LST1    ; and also these two global vars
        .DEF     TEMP1     ; this allows NULLFUNC to get at temp1
                           ; code starts here...
        POP      RETADR    ; assume return addr pushed on stack

        ; does nothing

        PUSH     RETADR    ; set up stack for return
        RETURN
                           ; data area
        RETADR   .EQU  TEMP1
        TEMP1    .WORD
                           ; end of procedure DONOTHING


        .FUNC    NULLFUNC,2

        .PRIVATE  PRT,LST2:9 ; 10 words of private data
        .REF      TEMP1      ; references data temp in DONOTHING
                             ; code starts here

        POP       RETURN     ; save return address

        POP       TEMP1      ; toss 2 words of junk
        POP       TEMP1

        POP       PRT        ; get parameter 'z'
        POP       LST2+4     ; get parameter 'xxyxx'

        ;  performs null action

        PUSH      LST2+4     ; return xxyxx as result
        PUSH      RETURN     ; restore subr link
        RETURN               ; return to calling program
                             ; data starts here...
RETURN  .WORD
                             ; end of assembly
        .END
```

2.3 Stand Alone Applications

The Adaptable Assemblers were originally developed to allow the Pascal project to maintain all interpreters and I/O systems on the Pascal system in order to be completely self-supporting; thus, in their current configuration, the assemblers have the capability to produce absolute (core image) code files for use outside of the Pascal system's runtime environment.

The Pascal system does not include a linking loader or an assembly language debugger, as the P-machine architecture is not conducive to running programs (Pascal or assembly language) that must reside in a dedicated area of memory. The user is responsible for loading and executing the object code file; this can be done using the Pascal system, with the understanding that the existing run time environment may be jeopardized in the process. Section 6.2.3.2 provides some ideas on how to create a Pascal loader program.

2.3.1 Assembling

The .ABSOLUTE and .ORG directives are used to create an object code file suitable for use as an absolute core image. .ABSOLUTE causes the creation of nonrelocatable object code, and .ORG may be used to initialize the location counter to any starting value. A source file headed by .ABSOLUTE should not have more than one assembly routine; sequential absolute routines do not produce continuous object code and cannot be successfully linked with one another to produce a core image.

The code file format consists of a 1 block code file header followed by the absolute code, and is terminated by one block of linker info; thus, stripping off the first and last block of the code file will leave a core image file. The use of .ABSOLUTE should be limited to one routine; though linker information is generated, it is difficult to link absolute code files so as to produce a correct core image file.

2.3.2 Loading And Executing Core Image Files

The following section describes one method of loading and executing absolute code files using the Pascal system. The program outlined is not the only solution. It is also feasible to use the system intrinsics to read and/or move the code file into the desired memory location, but this requires a knowledge of where the interpreter, operating system, and user program reside in order to prevent system crashes by accidentally overwriting them. The program outlined below allows the most freedom in loading core images; the only constraint is that the assembly code itself is not overwritten while being moved to its final location. This possiblity can be detected before loading proceeds.

It must be emphasized that in most cases loading object code into arbitrary memory locations while a Pascal system is resident will adversely affect the system; the absolute assembly language program is then on its own, and rebooting may be necessary to revive the Pascal system.

The loader program consists of:

1) A Pascal host program that calls two external procedures.

2) One or more linkable absolute code files to be loaded.

3) A small assembly procedure MOVE_AND_GO that moves the above object code files from their system load address to their proper locations and transfers control to them.

4) A small assembly language procedure LOAD_ADDRESS that returns the system load addresses of the aforementioned assembly code to the host program.

The absolute code files are assembled to run at their desired locations, and MOVE_AND_GO contains the desired load addresses of each core image. Both LOAD_ADDRESS and MOVE_AND_GO have external references to the core images; these are used to calculate the system load address and code size of each image file. The whole collection is linked and executed, with the Pascal host performing the following actions:

Print the result of calling LOAD_ADDRESS to determine whether the area of memory in which the Pascal system loaded the assembly code overlays the known final load address of the core images. Issue a prompt to continue, so that the program can be aborted if a conflict does arise.

Call MOVE_AND_GO.

2.3.3 Byte Sex Considerations

All assembler versions and the system linker are designed to produce assembly code files with the correct byte sex of the target processor, regardless of the byte sex of the machine underlying the Pascal system in use; thus, there is no need for 'code flipping' software of the sort required for P-code.

SECTION 1.7.7

OPERATION OF THE ASSEMBLER

The system assembler is invoked by typing 'A' at the command level of the operating system. This command will execute the file named SYSTEM.ASSMBLER (note the missing 'E' in the file name; this is required for conformance with the file system's restrictions on file name lengths); if this is not the name of the desired assembler version, be sure to save the existing file 'SYSTEM.ASSMBLER' under a different name before changing the desired assembler's name to 'SYSTEM.ASSMBLER'. Assemblers that are not in use are usually saved with the file name 'ASM <processor #>.CODE' (e.g., 'ASM6809.CODE').

1. Support Files

Each assembler version has two associated support file: an opcodes file and an error file. These should always be stored along with the assembler code file.

In order for the assembler to run correctly, it is necessary that the proper opcodes file be present on some on-line disk; the assembler will search all units in increasing order of the unit number until it finds it. The opcode file must have the name '<processor #>.OPCODES', where <processor #> matches the processor of the current system assembler. The opcode file contains all predefined symbols (instruction and register names) and their corresponding values for the associated assembly language. If the proper opfile is not on-line, the assembler will write '<opfilename> not on any vol' and abort the assembly.

Each assembler also has its own error file which contains a list of machine specific error messages. The error file must have the name '<processor #>.ERRORS', where <processor #> matches the processor of the current system assembler. The presence of an error file is not necessary for running the assembler, but it can greatly aid the chore of squeezing the syntax errors out of a freshly written program.

2. Setting Up Input And Output Files

When the assembler is first invoked from the prompt line, it will attempt to open the work file as its input file; if a work file exists, the first prompt will be the listing prompt described in Section 7.3 and the generated code file will be named 'SYSTEM.WRK.CODE'. If not, this prompt will appear:

```
Assemble what text?
```

Type in the file name of the input file followed by a carriage return. Typing only a carriage return will abort the assembly; otherwise, the next prompt will then appear:

```
To what codefile?
```

Type in the desired name of the output code file followed by a carriage return. Typing only a carriage return here will cause the assembler to name the output 'SYSTEM.WRK.CODE', but typing '$' will cause the code file to be created with the same filename prefix as the source file. The assembler will then display its standard listing prompt.

3. Responses To Listing Prompt

Before assembling begins, the following prompt will appear on the console:

```
xxxx   Assembler  [yy]
Output file for assembled listing: (<CR> for none)
```

xxxx is the processor number and yy is the release level of the assembler. At this point, the user may respond with one of the following:

0) The escape key will abort the assembly and return the user to the operating system prompt.

1) 'CONSOLE:' or '#1:' will send an assembled listing of the source program to the screen during assembly.

2) 'PRINTER:' or '#6:' will send an assembled listing to the printer unit.

3) 'REMOUT:' or '#8:' will send an assembled listing to the REMOTE unit.

4) A carriage return will cause the assembler to suppress generation of an assembled listing and ignore all listing directives.

5) All other responses will cause the assembler to write the assembled listing to a text file of that name; any existing textfile of that name will be removed in the process. For instance, the following responses will cause a list file named 'LISTING.TEXT' to be created on disk unit 5:

```
#5:listing.text
#5:listing
```

In all cases, it is the responsibility of the user to ensure that the specified unit is on-line; the assembler will print an error message and abort if it is requested to open an off-line I/O unit.

4. Output Modes

If the user sends an assembled listing to the console, logic dictates that this is what will be displayed on the screen during the assembly process; however, if the listing is sent to some other unit or if no listing is generated, the assembler writes a running account of the assembly process to the screen for the user's benefit. One dot is written to the screen for every line assembled; on every 50'th line, the number of lines currently assembled is written on the left hand side of the screen (delimited by angle brackets), along with the current number of memory words available to the assembler (delimited by square brackets).

When an include file directive is processed by the assembler, the console displays the current source statement:

```
.INCLUDE  <file name>
```

This allows the user to keep track of which include file is currently being assembled.

At the end of the assembly, the console displays the total number of lines assembled in the source program, the number of lines assembled per minute (if possible), and the total number of errors flagged in the source program.

5. Responses To Error Prompt

When the assembler uncovers an error, it will print the error number and the current source statement (if applicable to the error; this does not apply to undefined labels and system errors). It then attempts to retrieve and print an error message from the errors file. If the errors file cannot be opened (file is nonexistent or lack of memory), no message will appear. This is followed by the prompt:

E(dit, <space>, <esc>

Typing an 'E' will invoke the editor, a space will continue the assembly, and an escape character will abort the assembly. Some restrictions exist when either invoking the editor or attempting to continue:

1) In most cases, typing a space character restarts the assembly process with no problems; since assembly language source statements are independent of one another with respect to syntax, it is not a difficult task for the assembler to continue generating a code file. Thus, a code file will exist at the end of an assembly if the user types a space for every (nonfatal) error prompt that appears; of course, the code produced may not be a correct translation of the user's source program. Certain system errors are considered fatal by the assembler; these errors will abort the assembly regardless of the response given to the above prompt.

2) If an 'E' is typed, the system automatically invokes the editor, which opens the file containing the offending error and positions the cursor at the location where the error occurred. This feature will always work correctly when the source program is wholly contained in one file; however, when include files are used, the user should set up the input and output files manually (see Section 7.2) in order for the editor to position the cursor in the file that contains the error.

5.1 Miscellany

At the end of an assembly, an error message for each undefined label is printed. In some cases, occurences of undefined labels can be ignored by the user if the labels in question are semantically irrelevant to the desired execution of the code file; the resulting code file will be perfectly valid, but the references to the nonexistent labels will not be completely resolved.

In addition to generating a codefile, the assembler makes use of a scratch file, which is always removed from the disk upon normal termination of the assembly. Occasionally though, a system error may occur that will prevent the assembler from removing this file; if this happens, a new file may appear named 'LINKER.INFO'. It may be removed without anxiety, as it is entirely useless outside of the assembler's domain. This should be a rare (if not nonexistent) phenomenon.

SECTION 1.7.8

ASSEMBLER OUTPUT

The assembler can generate two varieties of output files. A code file is always produced, but the user controls whether an assembled listing of the source file is produced.

An assembled listing displays each line of the source program, the machine code generated by that line, and the current value of the location counter. The listing may display the expanded form of all macro calls in the source program. Any errors that occur during the assembly process have messages printed in the listing file, usually immediately following the line of source code that caused the error. A symbol table is printed at the end of the listing; it serves as a directory for locating all labels declared in the source program.

An assembled listing of a source program printed on hard copy is one of the most effective debugging aids available for assembly language programs; it is equally useful for off-line, 'mental' debugging and in conjunction with system debuggers.

A description of the code file format is beyond the scope of this document.

1. Source Listing

A paginated assembled listing is produced when the user responds to the assembler's listing prompt with a listfile name. The listing is 102 characters wide and 55 lines per page. Each line of a source program is included in the assembled listing, except for source lines that contain list directives. Source statements that contain the equate directive listed to the left of the source line.

Macro calls are always listed, including the list of macro parameters and the comment field, if any. The macro is expanded by listing the body (with all formal parameters replaced by their passed values) if the macro list option was enabled when the macro was defined. Macro expansion text is marked in the assembled listing by the character '#' just to the left of the source listing. Comment fields in the definition of the macro body are not listed in macro expansions.

Source lines with conditional assembly directives are listed; however, source statements in an unassembled part of a conditional section are not listed.

2. Error Messages

Error messages in assembled listings have the same format as the error messages sent to the console (see Section 7), except that the user prompt is not included.

3. Code Listing

The code field lies to the left of the source program listing. It always contains the current value of the location counter, along with either code generated by the matching source statement or the value of an expression occuring in a statement that includes the equate directive .EQU; all are printed in the default list radix of the assembler version being used (either hex or octal - see Section 9). Separately emitted bytes and words of code on the same line are delimited by spaces.

3.1 Forward References

When the assembler is forced to emit a byte or word quantity that is the result of evaluating an expression that includes an undefined label, it lists a '*' for each digit of the quantity printed (e.g., an unresolved hex byte is listed as '**', while an unresolved octal word appears as the assembler lists patch messages every time it encounters a label declaration that enables it to resolve all occurrences of a forward reference to that label. The messages (one for every backpatch performed) appear before the source statement that contains the label in question, and are of the form:

<location in code file patched>* <patch value>

With this feature, the listing describes the contents of each byte or word of emitted code; if neatness of the assembled listing is more desirable, the .NOPATCHLIST directive will suppress the patch messages.

3.2 External References

When the assembler emits a word quantity that is the result of evaluating an expression that contains an externally referenced label, the value of that label (which cannot be determined until link time) is taken as zero; therefore, the emitted value will reflect only the result of any assembly time constants that were present in the expression.

3.3 Multiple Code Lines

Sometimes, it is possible for one source statement to generate more code than will fit in the code field; in most cases, the code is listed on successive lines of the code field (with corresponding blank source listing fields). Three exceptions are the .ORG, .ALIGN, and .BLOCK directives; because most uses of these directives generate large numbers of uninteresting byte values, the code field for these arguments is limited to as many bytes as will fit in the code field of one line.

4. Symbol Table

The symbol table is an alphabetically sorted table of entries for all symbols declared in the source program. Each entry consists of three fields; the symbol identifier, the symbol type, and the value assigned to that symbol. The symbol identifiers are defined in a dictionary printed at the top of the symbol table. Symbols equated to constants have their constant values in the third field, while program labels are matched with their location counter offsets; all other symbols have dashes in their value field, as they possess no values relevant to the listing.

5. Example

The following is a small example of an assembled listing:

```
0000|
0000|                          .ABSOLUTE
0000|
0000|                          .PROC   PRIMARYZ
0000|
0000| 0BFD            FLOPPY   .EQU   0BFDH          ;Rom-based floppy driver
0000| 9000            SECMEM   .EQU   9000H          ;First location in memory
0000| 9000            SECENT   .EQU   9000H          ;Entry point of bootstrap
0000| 1708            SECDSK   .EQU   08H + 1700H    ;Sector start of 2nd bootstrap
0000| 1710            B1DSK    .EQU   10H + 1700H    ;Sector start of BIOS part 1
0000| 1718            B2DSK    .EQU   18H + 1700H    ;Sector start of BIOS part 2
0000|
0000|                          .ORG   1000H          ;Primary boot for GRISWICH 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
1029| 0002                     .WORD  200H          ;Number of bytes in BIOS part
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
```

ASSEMBLER OUTPUT

```
1036| 0002                    .WORD 200H          ;Number of bytes in BIOS part
1038| 0000                    .WORD               ;Completion return address
103A| 0010                    .WORD PRIMARY       ;Error return address
103C| 00                      .BYTE               ;Completion result code
103D| 1817                    .WORD B2DSK         ;Disk block of BIOS part 2
103F|
103F|                         .END
```

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

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

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

SECTION 1.7.9

MACHINE SPECIFIC INFORMATION

This document is intended to be used in conjunction with processor manuals distributed by the manufacturers of the various processors. These manuals provide syntax conventions for the instruction sets and address modes used by the corresponding Adaptable Assembler versions. The company chosen as a base for syntax conventions is listed for each version, along with a list of deviations from that company's syntax conventions.

1. LSI-11/PDP-11 Assembler

1.1 Syntax Conventions

The 11 assembler adheres to DEC standard syntax for instruction names, register names, and address modes. The location counter symbol is an asterisk '*'.

1.2 Sharing of Machine Resources with Interpreter

The return address to the system is passed on the stack. Registers 0 and 1 are available to the assembly routine; other registers must be saved on entry and restored on exit.

1.3 Memory Organization

The 11 processor is byte addressed and word oriented; machine instructions and data words must be aligned to start on an even byte boundary. The byte sex is least-significant-byte-first.

1.4 Default Constant and List Radices

The default constant radix and default list radix are octal.

2. Z80 Assembler

2.1 Syntax Conventions

The Z80 assembler adheres to Zilog standard syntax for instruction names, register names, and address modes. The following conventions may deviate from this standard:

- the syntax for exchanging the register pair AF and the alternate register pair AF' is the following:

```
EX   AF
```

The location counter symbol is a dollar sign '$'.

2.2 Sharing of Machine Resources with Interpreter

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

2.3 Memory Organization

The Z80 processor is byte addressed and byte oriented. The byte sex is least-significant-byte-first.

2.4 Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

3. 6500 Assembler

3.1 Syntax Conventions

The 6500 assembler adheres to Rockwell standard syntax for instruction names and register names. The following conventions may deviate from this standard:

- immediate operands are specified by using a preceding pound sign '#' character:

```
LABEL   .EQU   5
 LDA    #LABEL       ; immediate
```

- zero-page addressing is achieved only by using absolute operands (i.e., assembly time constants) with values between 0 and 255:

```
LABEL   .EQU   5
 LDA    LABEL        ; zero-page
```

- indirect addressing has the following form:

```
 LDA    @LABEL,X     ; indexed-indirect (preindexing)
 LDA    @LABEL,Y     ; indirect-indexed (postindexing)
 JMP    @LABEL       ; indirect jump
```

The location counter symbol is an asterisk '*'.

3.2 Sharing of Machine Resources with Interpreter

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

3.3 Memory Organization

The 6502 processor is byte addressed and byte oriented. The byte sex is least-significant-byte-first.

3.4 Default Constant and List Radices

The default constant radix and default list radix are hexadecimal.

4. 6800 Assembler

4.1 Syntax Conventions

The 6800 assembler adheres to Motorola standard syntax for instruction names and register names. The following conventions may deviate from this standard:

- all instructions which can specify the A and B registers have the register name separated from the instruction name:

```
LDA   A,LABEL
STA   A,14,X
PUL   A
ASL   B
```

- immediate operands are specified by using a preceding pound sign '#' character:

```
LABEL   .EQU   5
 LDA    A,#LABEL     ; immediate
```

- zero-page addressing is achieved only by using absolute operands (i.e., assembly time constants) with values between 0 and 255:

```
LABEL   .EQU   5
 LDA    B,LABEL      ; zero-page
```

The location counter symbol is an asterisk '*'.

4.2 Sharing of Machine Resources with Interpreter

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

4.3 Memory Organization

The 6800 processor is byte addressed and byte oriented. The byte sex is most-significant-byte-first.

4.4 Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

5. 8080 Assembler

5.1 Syntax Conventions

The 8080 assembler adheres to Intel standard syntax for instruction names, register names, and address modes. The location counter symbol is a dollar sign '$'.

5.2 Sharing of Machine Resources with Interpreter

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

5.3 Memory Organization

The 8080 processor is byte addressed and byte oriented. The byte sex is least-significant-byte-first.

5.4 Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

6. 9900 Assembler

6.1 Syntax Conventions

The 9900 assembler adheres to TI standard syntax for instruction names, register names, and address modes. The following conventions may deviate from this standard:

- in operand fields, the lack of an address mode character (i.e., a '@' or '*' preceding the operand) defaults to '@'.

The location counter symbol is a dollar sign '$'.

6.2 Sharing of Machine Resources with Interpreter

The return address to the system is passed in register 11. Registers 0 thru 5 are available to the assembly routine; other registers must be saved on entry and restored on exit.

6.3 Memory Organization

The 9900 processor is byte addressed and word oriented; machine instructions and data words must be aligned to start on an even byte boundary. The byte sex is most-significant-byte-first.

6.4 Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

-- Notes --

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.

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.

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

INTEGERPOINTER : ↑INTEGER

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 3.4 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 --

```
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 2.2.6, 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.

NOTE: SEEK only works for structured files as defined in Section 3.2. (This excludes .TEXT files.)

```
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: INTEGERPOINTER);
PROCEDURE RELEASE (VAR HEAPPTR: INTEGERPOINTER)
```

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. See also section 5.2.2.3

```
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 4.1.

```
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.

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 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.)

This section is a summary and quick reference 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.

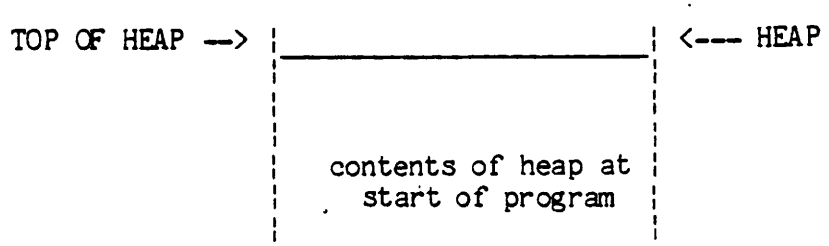
```
PROGRAM SMALLHEAP;

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.
```

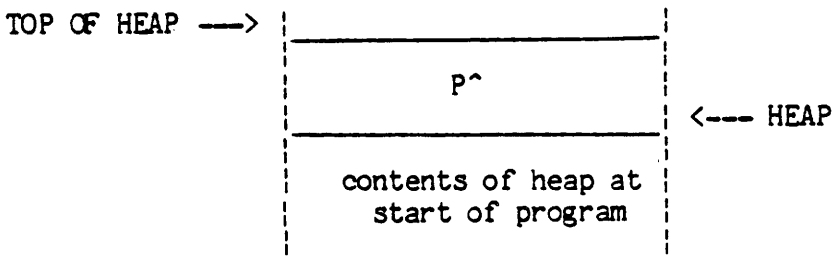
The above program first calls MARK to place the address of the current top of heap into the variable HEAP. HEAP must be declared to be a pointer to an INTEGER. The parameter supplied to MARK must be a pointer to an INTEGER. This is a UCSD restriction. This is a particularly handy construct for deliberately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



contents of heap at start of program

<--- HEAP

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:

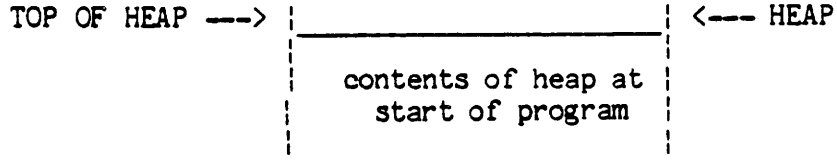


P^

<--- HEAP

contents of heap at start of program

After the RELEASE the heap is as follows:



<--- 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 2.1.4 "CHARACTER ARRAY MANIPULATION INTRINSICS".

B. PACKED RECORDS

The following RECORD declaration declares a RECORD with 4 fields. The entire RECORD occupies one 16 bit word as a result of declaring it to be a PACKED RECORD.

```
VAR R: PACKED RECORD
         I,J,K: 0..31;
         B: BOOLEAN
       END;
```

The variables I, J, K each take up 5 bits in the word. The boolean variable B is allocated to the 16'th bit of the same word.

In much the same manner that PACKED ARRAYs can be multidimensional PACKED ARRAYs, PACKED RECORDS may contain fields which themselves are PACKED RECORDS or PACKED ARRAYS. Again, slight differences in the way in which declarations are made will affect the degree of packing achieved. For example, note that the following two declarations are not equivalent:

```
VAR A:PACKED RECORD
      C:INTEGER;
      F:PACKED RECORD
                 R: CHAR;
                 K: BOOLEAN
               END;
      H:PACKED ARRAY[0..3] OF CHAR
    END;
```

```
VAR B:PACKED RECORD
      C:INTEGER;
      F:RECORD
          R:CHAR;
          K:BOOLEAN
        END;
      H:PACKED ARRAY[0..3] OF CHAR
    END;
```

As with the reserved word ARRAY, the reserved word PACKED must appear with every occurrence of the reserved word RECORD in order for the PACKED RECORD to retain its packed qualities throughout all fields of the RECORD. In the above example, only RECORD A has all of its fields packed into one word. In B, the F field is not packed and therefore occupies two 16 bit words. It is important to note that a packed or unpacked ARRAY or RECORD which is a field of a PACKED RECORD will always start at the beginning of the next word boundary. This means that in the case of A, even though the F field does not completely fill one word, the H field starts at the beginning of the next word boundary.

A case variant may be used as the last field of a PACKED RECORD, and the amount of space allocated to it will be the size of the largest variant 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;
VAR X,Y:TEXT;
    CH: CHAR;
BEGIN
  RESET(X,'SOURCE.TEXT');
  REWRITE(Y,'SOMETHING.TEXT');

  WHILE NOT EOF(X) DO
    BEGIN
      WHILE NOT EOLN(X) DO
        BEGIN
          READ(X,CH);
          WRITE(Y,CH);
        END;
      READLN(X);
      WRITELN(Y);
    END;
  CLOSE(Y,LOCK);
END.
```

```
PROGRAM UCSDVERSION;
VAR X,Y:INTERACTIVE;
    CH:CHAR;
BEGIN
  RESET(X,'CONSOLE:');
  REWRITE(Y,'SOMETHING.TEXT');
  READ(X,CH);
  WHILE NOT EOF(X) DO
    BEGIN
      WHILE NOT EOLN(X) DO
        BEGIN
          WRITE(Y,CH);
          READ(X,CH);
        END;
      READLN(X);
      WRITELN(Y);
    END;
  CLOSE(Y,LOCK);
END.
```

Note that the textfiles X and Y in the above two programs had to be opened by using the 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 3.2 - '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';
```

*characters will be read until EOLN or EOF is detected.

```
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 compatible array or record structure.

2.2.20 LONG INTEGERS.

UCSD Pascal allows integers of up to 36 digits. See section 3.4 for details regarding long integers.

2.2.21 UNITS.

UCSD Pascal now supports the modularity concept of UNITs. See section 3.3 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 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.

— Notes —

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.
NOTE: Text files are considered unstructured files; SEEK will not work with them.

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 3.5 and 3.6 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.

********** **********
* SEGMENT PROCEDURE NOTES * * Section 3.2 *
********** **********

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 3.6, INTRODUCTION TO THE PASCAL PSEUDO MACHINE.

EXTERNAL COMPILATION UNITS

The UCSD Pascal 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. 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, XO, YO, DX, DY, PEN: Integer );

    EXTERNAL; (* External declaration *)


  PROCEDURE INITTURTLE;
  BEGIN
    Fillchar( SCREEN, Sizeof(SCREEN), 0 );
    Unitwrite( 3, SCREEN, 63 );
    HEADING := 0;
    TGXPOS := 0;
    TGYPOS := 0;
  END;

  PROCEDURE MOVE;(* Public procedure, parameters declared above *)
  BEGIN
    MOVETO( Round(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   *)
                                              (*   turtlegraphics       *)
BEGIN
  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

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 1.6 LINKER or section 4.1 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 1.6 explains the format of this information.

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 1.6 Linker and 1.7 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 Z80 assembly language routines, all registers are available for use, and the recommended interface sequence follows: (This code would work for both 8080's and Z80's. Optimizations are possible if the Z80 instructions are available.)

```
        .PROC   ENTRY, 2
        .PRIVATE        RETADDR,LOCAL1,LOCAL2,PARAM1,PARAM2
        ;Reserve static storage for this routine.  Much easier to
        ;reference objects like this rather than relative to
        ;register as on PDP-11
        POP     HL              ;Get return address
        LD      (RETADDR),HL    ;and save it
        POP     HL              ;Get and save PARAM2
        LD      (PARAM2),HL
        POP     HL              ;Get and save PARAM1
        LD      (PARAM1),HL
        .
        .
        .
        LD      HL,(PARAM2)     ;Move PARAM2
        LD      (LOCAL1),HL     ;to LOCAL1
        .
        .
        .
EXIT:   LD      HL,(RETADDR)    ;Get return address
        JP      (HL)
        .END
```

For assembly language functions (.FUNC's) the sequence is essentially the same, except that:

1) Two words of zeros are pushed by the compiler 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 .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 1.7 ASSEM

-- Notes --

********** **********
* LONG INTEGERS * * SECTION 3.4 *
********** **********

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: INTEGER[N];{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 = INTEGER[8];
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.

****
* PSUEDO-MACHINE ARCHITECTURE * * Section 3.5 *
****

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 3.6, describes the P-machine "hardware," communication with the operating system, exceptional condition handling, the instruction set, the I/O system, and the bootloading process.

3.5.1 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 3.6, figure 5.)

SEG: Segment Pointer points to the procedure dictionary of the segment to which the currently executing procedure belongs. (See Section 3.6, figure 6.)

MP: Most recent Procedure is a pointer to the activation record of the currently executing procedure. (See Section 3.6, 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.

3.5.2 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.

3.5.3 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^.BOMBP to that. (The size of EXECERROR's activation record must be known by the P-machine.)
3) stores the current value of IPC into SYSCOM^.BOMIPC.
4) points IPC to a CXP 0,2 (call operating system procedure EXECERROR) instruction.
5) resumes execution of interpreter code, starting with the CXP.

3.5.4 OPERAND FORMATS

Although an element of a structure may occupy as little as one bit, as in a PACKED ARRAY OF boolean, variables in the P-machine are always aligned on word boundaries. All top-of-stack operations expect their operands to occupy at least one word, even if not all the information in a word is valid. The least significant bit of a word is bit 0, the most significant is bit 15.

BOOLEAN: One word. Bit 0 indicates the value (false=0, true=1), and this is the only information used by boolean comparisons. However, the boolean operators LAND, LOR, and LNOT operate on all 16 bits.

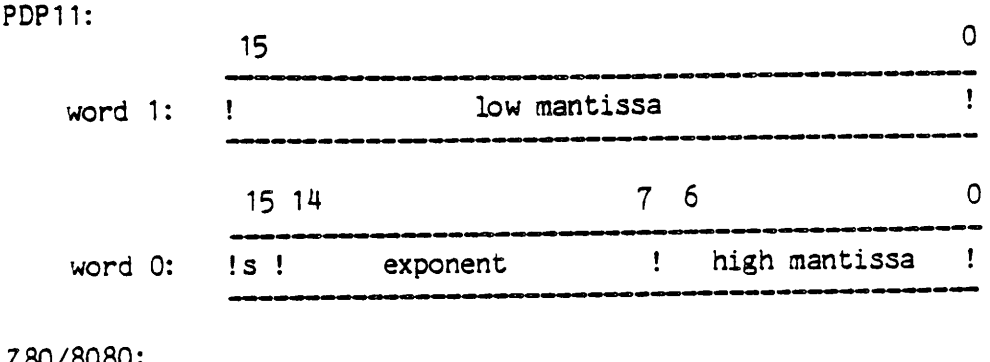
INTEGER: One word, two's complement, capable of representing values in the range -32768..32767.

SCALAR (user-defined): One word, in range 0..32767.

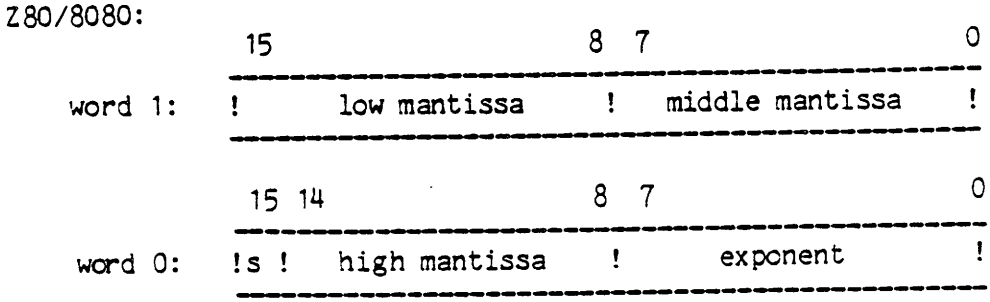
CHAR: One word, with low byte containing character. The internal character set is "extended" ASCII, with 0..127 representing the standard ASCII set, and 128..255 as a user-defined character set.

REAL: Two words, with format implementation dependent. The system is arranged so that only the interpreter needs to know the detailed internal format of REALs (beyond the fact that they occupy two words) Following are the two detailed formats for the CPUs we now (as of I.4) support.

PDP11:



Z80/8080:



Both representations have an excess-128 exponent, a fractional mantissa that is always normalized, exponent base 2, an implicit 24th mantissa bit, and zero represented by a zero exponent. (See PDP11 processor manual or Z80/8080 interpreter listing for greater detail.)

POINTER: One or three words, depending on type of pointer.
Pascal pointers, internal word pointers: one word, containing a word address.
Internal byte pointers: one word, containing a byte address.
Internal packed field pointers: three words.
word 2: word pointer to word field is in.
word 1: field_width (in bits).
word 0: right_bit_number of field.

SET: 0..255 words in data segment, 1..256 words on stack. Sets are implemented as bit vectors, always with a lower index of zero. A set variable declared as set of m..n is allocated (n+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.

3.5.5 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.

3.5.6 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 3.6, 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 .. SLDL16	216 .. 231		Short load local word. SLDLx fetches the word with offset x in MP activation record and pushes it.
LDL	202	B	Load local word. Fetches the word with offset B in MP activation record and pushes it.
LLA	198	B	Load local address. Fetches address of the word with offset B in MP activation record and pushes it.
STL	204	B	Store local word. Stores tos into word with offset B in MP activation record.

5.A.1.c GLOBAL ONE WORD LOADS AND STORE

SLDO1 .. SLDO16	232 .. 247		Short load global word. SLDOx fetches the word with offset x in BASE activation record and pushes it.
LDO	169	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	188	UB	Load multiple words. Tos is a pointer to the beginning of a block of UB words. Push the block onto the stack.
STM	189	UB	Store multiple words. Tos is a block of UB words, tos-1 is a word pointer to a similar block. Transfer the block from the stack to the destination block.

5.A.3 BYTE ARRAYS

LSA	166		Load string address. Put byte-length address of string constant on tos. Compiler has aligned address to a word.
LPA	208		Load packed array address. Put address of first data byte of a string constant on tos. Compiler has aligned address to a word.
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.

5.A.4 STRINGS

SAS	170	UB	String assign. Tos is either a source byte pointer or a character. (Characters always have a high byte of zero, while pointers 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.
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.
SIND0 ... SIND7	248 ... 255		Short index and load word. SINDx indexes the word pointer tos by x words, and pushes the word pointed to by the result.
IND	163	B	Static index and load word. Indexes the word pointer tos by B words, and pushes the word pointed to.
INC	162	B	Increment field pointer. The word pointer tos is indexed by B words and the resultant pointer is pushed.
IXA	164	B	Index array. Tos is an integer index, tos-1 is the array base word pointer, and B is the size (in words) or 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. (PDP11 quotient defined as in Jensen and Wirth; Z80/8080 quotient defined by floor(tos-1/tos).)
MODI	142	Modulo integers. Divide tos-1 by tos and push the remainder (as defined in Jensen and Wirth).
CHK	136	Check against subrange bounds. Insure that tos-1 <= tos-2 <= tos, leaving tos-2 on the stack. If conditions are not satisfied a run-time error occurs.
EQUI	195	Integer =, <>, <=, <, >=, and > comparisons. Compare tos-1 to tos and push true or false.
NEQI	203	
LEQI	200	
LESI	201	
GEQI	196	
GTRI	197	

5.8.3 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 1 less than the procedure being called. Use that activation record's static link as the

static link of the new MSCW.

CBP 194 UB — Call base procedure. Call procedure UB, which is at lex level -1 or 0. The static link of the MSCW is set to the static link in BASE's activation record. The BASE is saved, after which it is pointed at the activation record just created.

CXP 205 DB_1,UB_2 — Call external procedure. Used to call 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 --

****
* INTRODUCTION TO THE PASCAL PSEUDO-MACHINE * * Section 3.6 *
****

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 3.6.10 (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 3.6.1. (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> <segment block> .

<segment block> ::= <label declaration part>
     <constant declaration part> <type definition part>
     <variable declaration part> <segment declaration part>
     <segment body>

<segment declaration part> ::= SEGMENT <procedure heading>
     <segment block>; \ SEGMENT <function heading>
     <segment block>;

<segment body>::= <procedure and function declaration part>
     <statement part>
```

FIGURE 3.6.1. SEGMENT DECLARATION SYNTAX.

Segment declaration syntax (figure 3.6.1) 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).

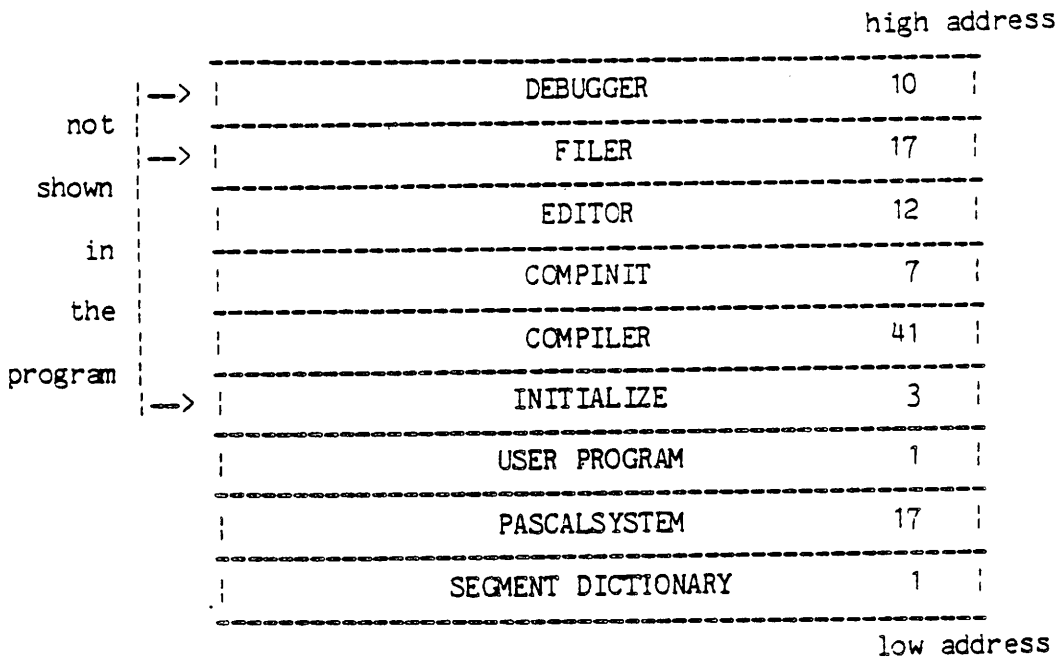


FIGURE 3.6.2. PASCAL SYSTEM CODE FILE.

The code file resulting from compilation of The Program is diagrammed in figure 3.6.2*. 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 3.6.2 through 3.6.8 (to be discussed in the following pages) is given in figure 3.6.9 at the end of this section. It is helpful to study figure 3.6.9 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 3.6.3 details the layout of the table and shows representative contents for the Pascal system code file.

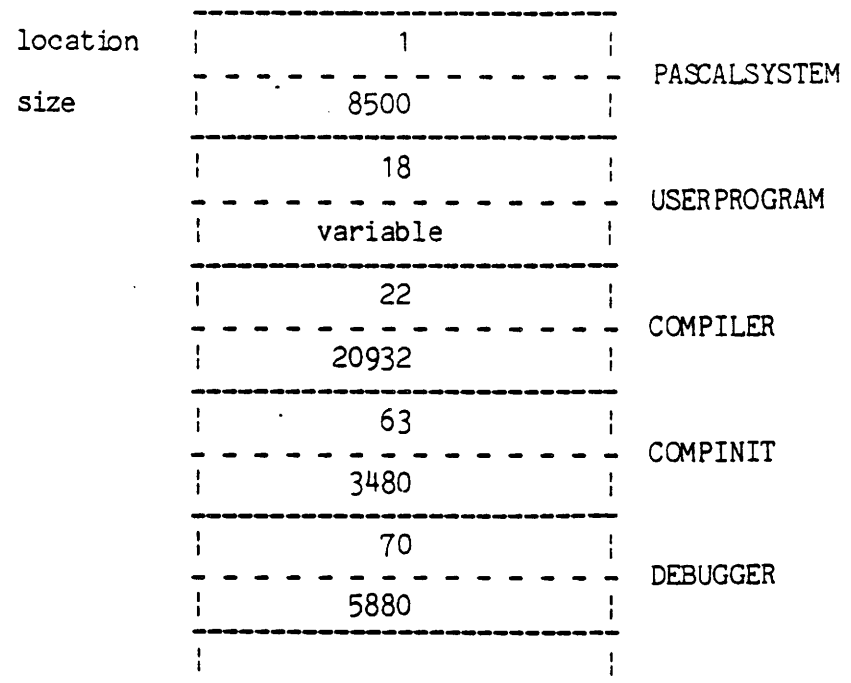


FIGURE 3.6.3. THE SEGMENT DICTIONARY

A code segment contains the code for the body of each of its procedures, including the segment procedure,itself. Figure 3.6.4 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.

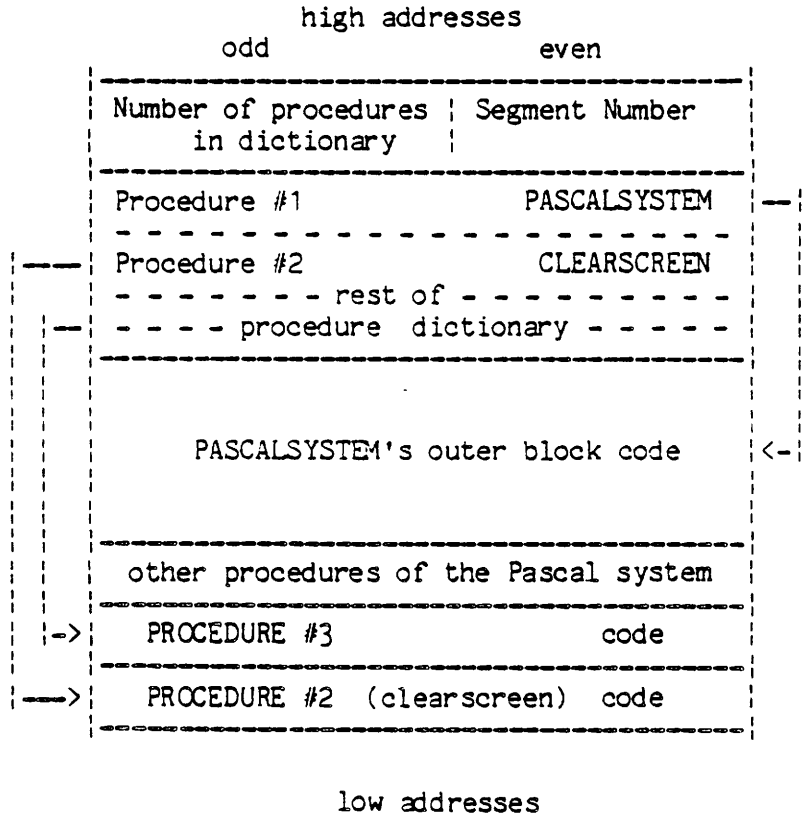


FIGURE 3.6.4. A CODE SEGMENT

A more detailed diagram of a single procedure code section is seen in figure 3.6.5. 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 3.6.4).

ENTER IC:This is a self-relative pointer to the first instruction to be executed for this procedure.

EXIT IC:This is a self-relative pointer to the beginning of the block of procedure instructions which must be executed to terminate procedure properly.

PARAMETER SIZE:The param size is the number of bytes of parameters passed to a procedure from its caller.

and DATA SEGMENT SIZE:The data size is the size of the data segment (See below) in bytes, excluding the markstack and PARAM SIZE.

Between these attributes and the procedure code there may be an optional section of memory called the "jump table". Its entries are addresses within the procedure code. JTAB is a term commonly applied to the six attributes just discussed and the jump table itself.

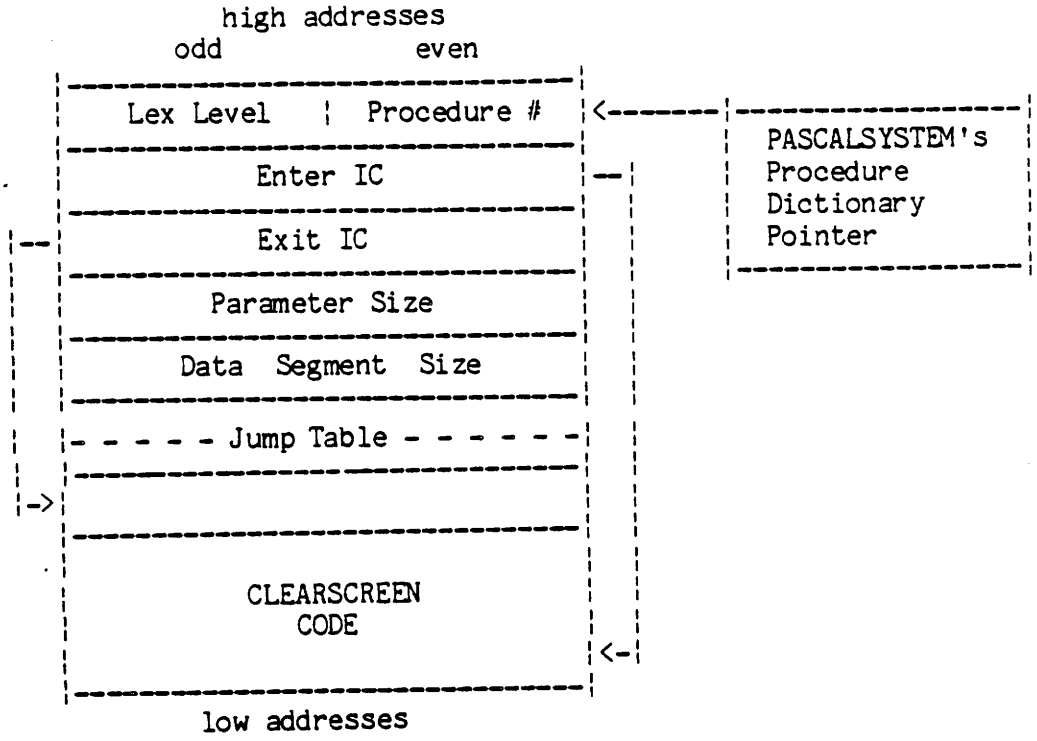


FIGURE 3.6.5. PROCEDURE CODE SECTION (OF CLEARSCREEN)

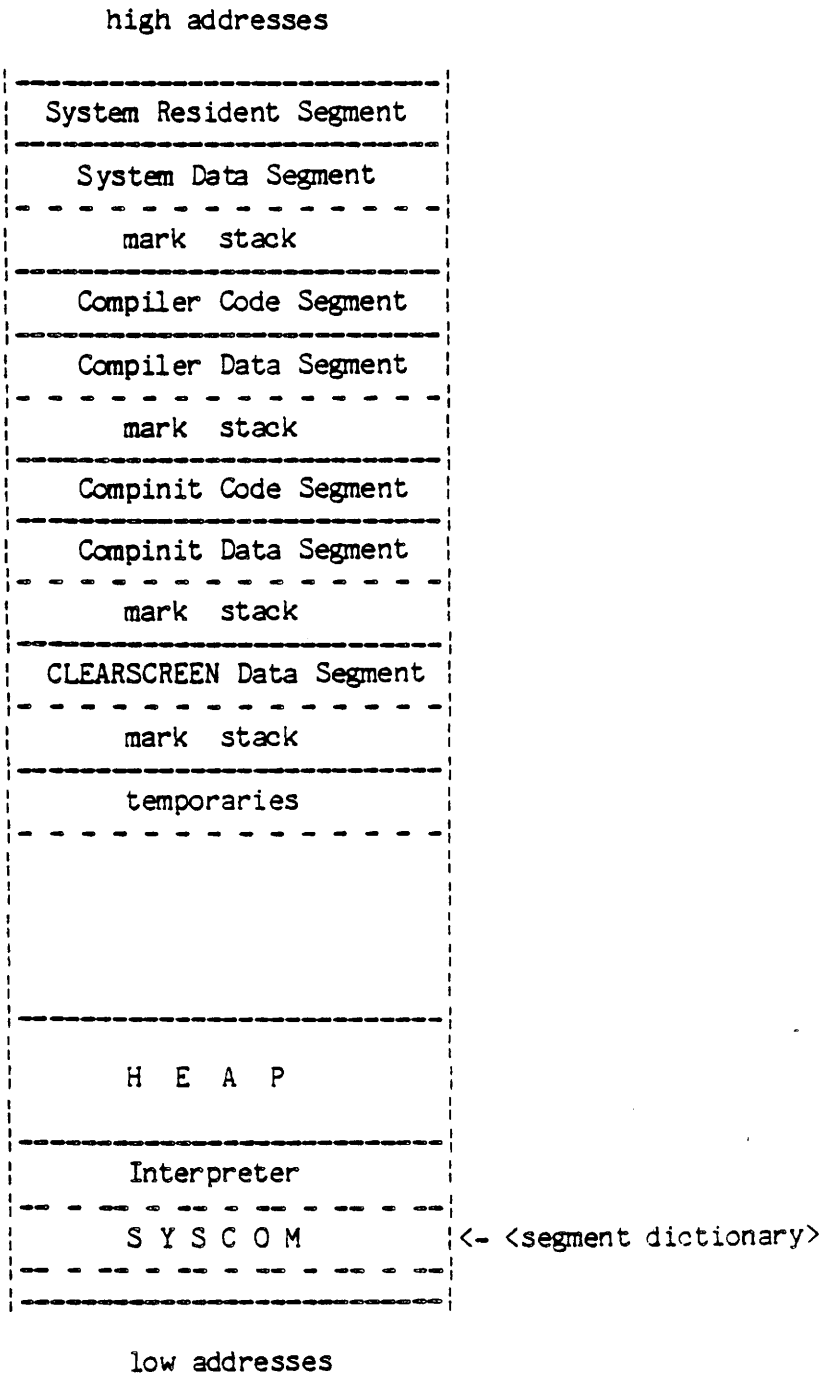

high addresses
System Resident Segment
System Data Segment
mark stack
Compiler Code Segment
Compiler Data Segment
mark stack
Compinit Code Segment
Compinit Data Segment
mark stack
CLEARSCREEN Data Segment
mark stack
temporaries
H E A P
Interpreter
S Y S C O M
<- <segment dictionary>
low addresses

FIGURE 3.6.6. SYSTEM MEMORY DURING CLEARSCREEN EXECUTION

Figure 3.6.6 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 3.6.6)

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 3.6.7 is a diagram of the data segment for COMPINIT.

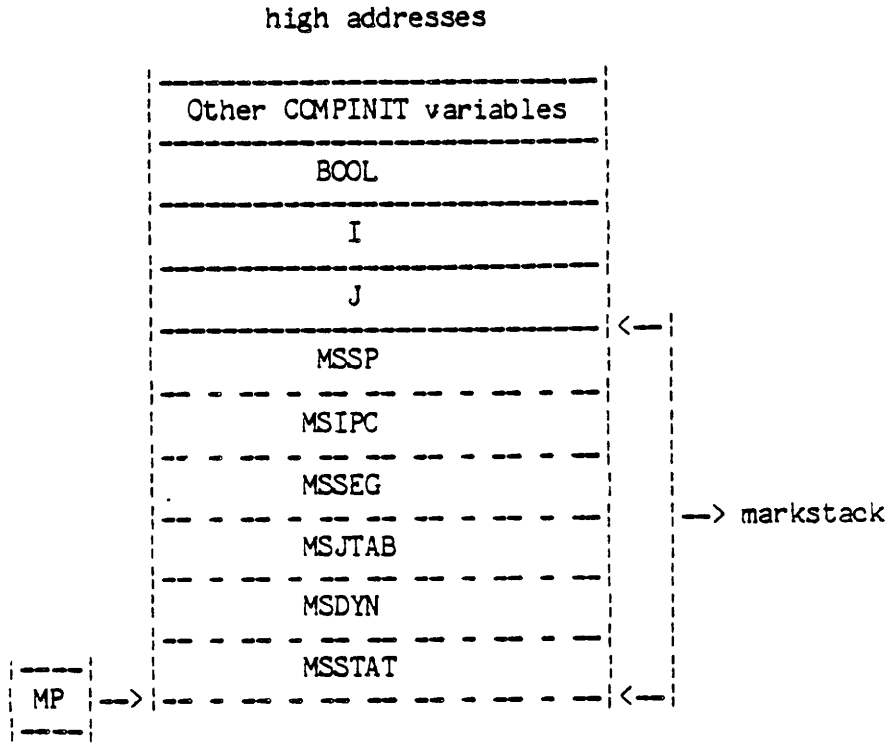


FIGURE 3.6.7. A DATA SEGMENT

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 3.6.8 is a detailed diagram of the stack during execution of an instruction in CLEARSCREEN, including appropriate pointers for static, dynamic, etc. links of CLEARSCREEN's markstack. Note where the pseudo-variables point in the stack. In particular, JTAB points inside CLEARSCREEN code section which is in the Pascalsystem code segment, IC points inside that CLEARSCREEN code, and SEG points to the base of the Pascalsystem code segment.

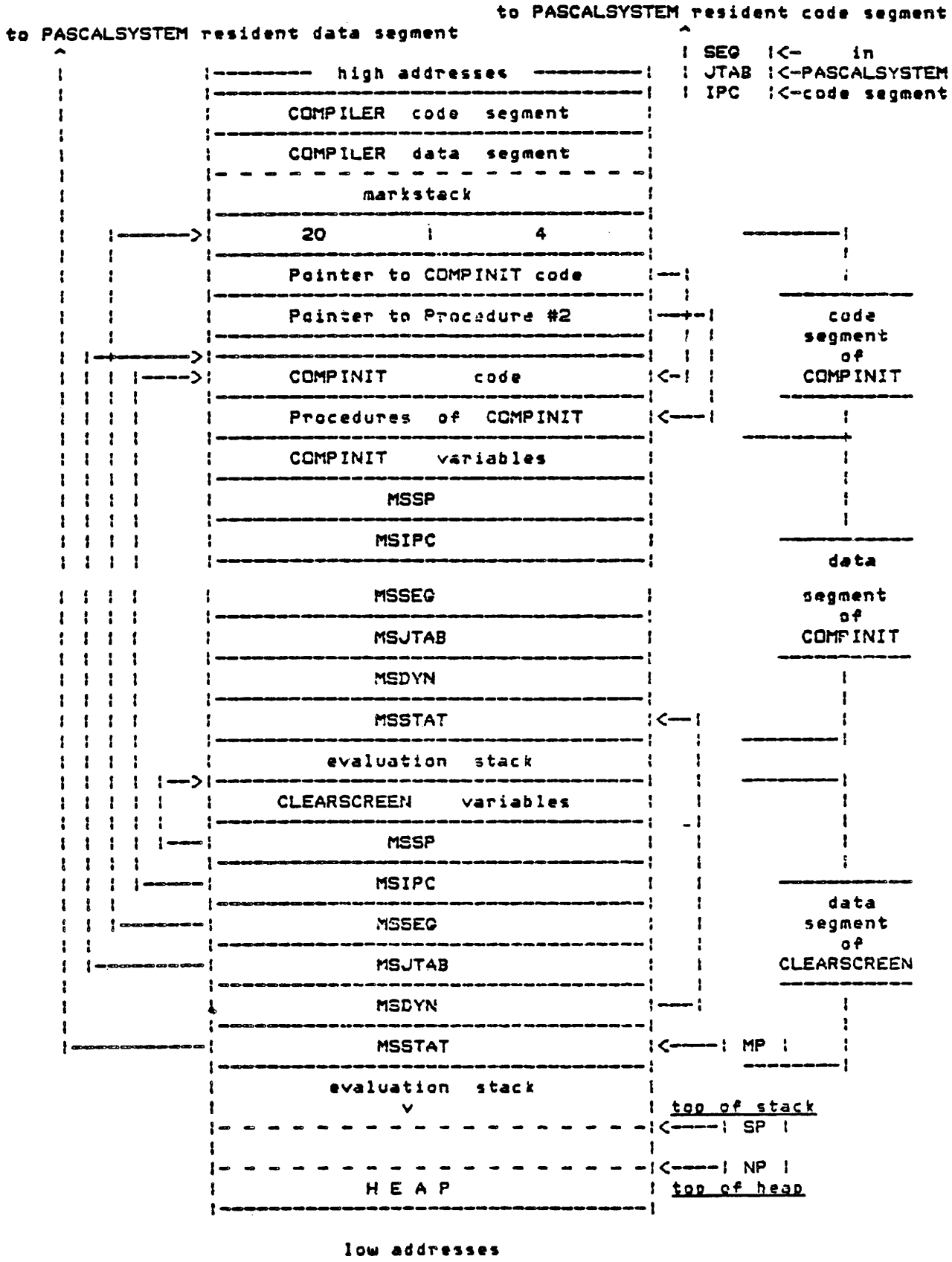


FIGURE 3.6.8. THE STACK DURING CLEARSCREEN

Figure 3.6.9 illustrates a top-down process by showing the relationships among diagrams 2 through 7.

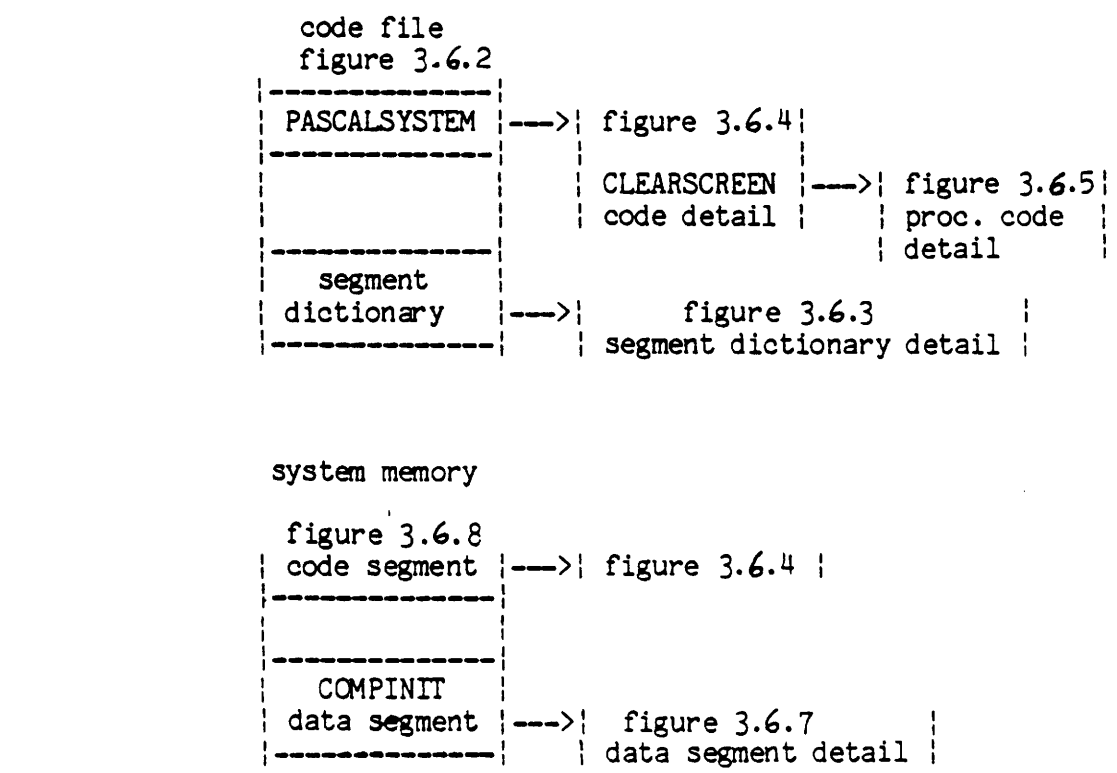


FIGURE 3.6.9. RELATIONSHIP OF DOCUMENT FIGURES

FIGURE 3.6.10. 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 --

BYTE-SWAPPING * Section 3.7

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) ls*		ms*	(B) 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.

********** **********
* LIBRARIAN UTILITY * * Section 4.1 *
********** **********

LIBRARY.CODE is a utility program that allows the user to link separately compiled PASCAL units and separately assembled subroutines into a LIBRARY file. It is based upon the original pre-I.5 utility LINKER.CODE and operates in basically the same way.

To add a segment to *SYSTEM.LIBRARY it is necessary to create a new file into which each segment that is wanted from the original *SYSTEM.LIBRARY is first linked. It is then possible to add segments by linking from another code file into the new file being created.

EXAMPLE

Consider the case of adding a segment called TURTLE to the already existing file *SYSTEM.LIBRARY which is assumed to contain the segments PSGRAPHICS and MOVETO.

On executing LIBRARY.CODE, the user is prompted for the name of the output codefile. For this example, respond with the name NEW.LIBRARY. The program now asks for a 'Link Code File'. The response here is *SYSTEM.LIBRARY. The names of all segments currently linked into the input library, i.e. *SYSTEM.LIBRARY, as well as their length in bytes is now displayed. Currently there are a maximum of 16 segments in any PASCAL program or LIBRARY.

0-	MOVETO	2398	4-	0	8-	0	10-	0
1-	PSGRAPHI	864	5-	0	9-	0	11-	0
2-		0	6-	0	10-	0	14-	0
3-		0	7-	0	11-	0	15-	0

The following promptline appears:

Segment # to link and <space>, N(ew file, Q(uit, A(bort

The user now enters the number of a segment within the link code file that is to be linked into the new library file, followed by <space>. Next, the number of the segment in the output file to be linked into (i.e. NEW.LIBRARY) is typed followed by <space>. For each segment linked the librarian reads that segment from the input file and writes it to the output file at the segment requested. It then displays the segment table for the current state of the output library file. In this example, respond with the following:

```
0<space>
Seg to link into? 0<space>
1<space>
Seg to link into? 1<space>
```

When all needed segments have been linked a new input file is requested by typing 'N' for N(ew file. In this example, a separately compiled PASCAL UNIT called TURTLE is assumed to exist in a codefile called TGRAPHICS.CODE. See section 3.2, UNITS. On entering the name of this file the following display appears:

```
0-          0   4-          0   8-          0   10-         0
1-          0   5-          0   9-          0   11-        0
2-          0   6-          0   10- TURTLE  230 14-        0
3-          0   7-          0   11-         0   15-        0
```

The Unit TURTLE occurs in segment 10 and is to be linked into segment 2 within NEW.LIBRARY. The user responds:

```
10<space>
Seg to link into? 2<space>
```

The final display of the output library segment table is thus:

```
0- MOVETO   2398   4-          0   8-          0   10-         0
1- PSGRAPHI  864   5-          0   9-          0   11-        0
2- TURTLE    230   6-          0   10-         0   14-        0
3-             0   7-          0   11-         0   15-        0
```

The output library codefile length is displayed and in this example is 16 (blocks long).

Once the needed segments from all input files have been linked in the user locks the output file by typing '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 --

SECTION 4.2.1
UCSD PASCAL -- TERMINAL HANDLING

You will want to read this document if
* You are new to the system,
* You want to change or improve the way the system handles your terminal, or
* You want to convert to a new variety of terminal.

The first thing you will be concerned with is SETUP, a utility program that modifies some terminal handling information stored in a file called SYSTEM.MISCINFO. The next thing to tailor is GOTOXY, a Pascal procedure within the Operating System that provides random addressing for your terminal's cursor. The system comes with its own defaults, but for more convenient or more efficient use of your console, you will almost certainly want to specify your own characteristics. Changing SYSTEM.MISCINFO with SETUP does not require much knowledge or preparation. Changing the GOTOXY procedure requires a little more familiarity with your terminal, and a knowledge of UCSD Pascal.

To tailor terminal handling to your own needs, you will first run SETUP. SETUP creates a file called NEW.MISCINFO which contains information about your own terminal. You will then go into the Filer, change SYSTEM.MISCINFO to a backup file, and change the name of NEW.MISCINFO to SYSTEM.MISCINFO. After this, you reboot or I(nitialize: the new SYSTEM.MISCINFO is loaded into main memory, and your terminal is now controlled according to the information in this file. To see if you have run SETUP correctly, you might want to run the SCREENTEST diagnostic immediately, or you might want to wait until you have bound in a new GOTOXY. To create your own GOTOXY, you will write a Pascal procedure that does cursor addressing, create a codefile by C(ompiling it, and bind the codefile into the Operating System by running the utility program BINDER.CODE. After binding, you should reboot, and then test the terminal handling by running SCREENTEST.

The SCREENTEST utility checks that characters are being sent and received properly, and that the Screen Oriented Editor interface will work. If you encounter problems, it is easy to go back into SETUP and change your specifications, or modify your GOTOXY procedure and bind it in again.

If you don't feel confident, you might do a little more reading. Check your own terminal manual, Section 2.1.2 of the Pascal User's Manual, which covers I/O intrinsics, and have a glimpse at Sections 1.3 and 1.4, which describe our two editors. YALOE can be used on virtually anything, but the Screen Oriented Editor, which is more convenient and is usually used as the system editor, requires a fairly efficient GOTOXY.

The rest of this document describes the care and feeding of SETUP, SCREENTEST, and GOTOXY.

SECTION 4.2.2
SETUP

This is the first Pascal procedure you will use when you set out to write your own terminal specifications. It is provided as a system utility called SETUP.CODE. SETUP changes a file that contains details about your terminal, and a few miscellaneous details about the system in general. SETUP can be run, and the data changed, as many times as you desire. After running it, it is important to reboot (or I(nitialize) so that the system will start using the new information. It is also important to backup old data, at least until after SCREENTEST, so that any hole you dig for yourself can be climbed back out of!

The file that SETUP uses to store all of this information is called SYSTEM.MISCINFO. Each system initialization loads it into main memory. New versions of SYSTEM.MISCINFO are created by SETUP, and are called NEW.MISCINFO. Backups are created by renaming or copying SYSTEM.MISCINFO with the Filer.

SYSTEM.MISCINFO contains three types of information:
- Miscellaneous data about the system,
- General information about the terminal, and
- Specific information about the terminal's various control keys.

Appendix D contains a sample session with SETUP. You might look this over before you actually run it.

1. RUNNING SETUP

SETUP is a utility program, and is run like any other compiled program: type X for eXecute, and then answer the prompt with "SETUP<carriage return>". It will display the word "INITIALIZING" followed by a string of dots, and then the prompt:

SETUP: C(HANGE T(EACH H(ELP Q(UIT [D1]

(The '[D1]' is the SETUP version number and may be different in your system.)

To invoke any command, just type the initial letter.

H(ELP gives you a clear description of the commands that are visible on any promptline where it appears.

T(EACH gives a detailed description of the use of SETUP. Most of it is concerned with input formats. They are mainly self-explanatory, but if this is your first time running SETUP, you should look through all of T(EACH.

C(HANGE gives you the option of going through a prompted menu of all the items, or changing one data item at a time. In either case, the current values are displayed, and you have the option of changing them. If this is your first time running SETUP, the values given are the system defaults. You will find that your particular terminal probably requires more sophisticated specifications.

Q(UIT has the following options:

H(ELP),
M(EMORY) UPDATE, which places the new values in main memory,
D(ISK) UPDATE, which creates NEW.MISCINFO on your disk for future use,
R(ETURN), which lets you go back into SETUP and make more changes, and
E(XIT), which ends the program and returns you to the Pascal Operating System.

Please note that if you have a NEW.MISCINFO already on your disk, D(ISK) UPDATE will write over it.

THE DATA ITEMS IN SYSTEM.MISCINFO contains a detailed description of this file (see below). An abbreviated list of all the data items, together with the system-supplied defaults, is at the end of this document, along with a list of sample settings for a variety of terminals (see Appendices A and B).

When you use SETUP to change your character set, don't underestimate the importance of using keys you can easily remember, and making dangerous keys like BREAK hard to hit.

Once you have run SETUP, you should always backup SYSTEM.MISCINFO under some other name (OLD.MISCINFO is one suggestion; you might want to name your backups according to different terminals, e.g., TTY.MISCINFO, IQ120.MISCINFO, VT52.MISCINFO, etc.), then change the name of NEW.MISCINFO to SYSTEM.MISCINFO and reboot or I(nitialize. It is indeed possible to update to memory alone, and go on using the system without rebooting, but the results may not always be what you wanted, and the backup security is more risky. In general, M(EMORY) UPDATE is a Q(UIT option that you will use only when experimenting. If you do get into a bind, remember that the current in-memory SYSTEM.MISCINFO can be saved by running SETUP and doing a D(ISK) UPDATE before you change any data items.

When you reboot or I(nitialize, the new SYSTEM.MISCINFO will be read into main memory and its data used by the system, provided it has been stored under that name on the system disk (the disk from which you boot).

The only thing SETUP will not arrange for you, as far as terminal handling goes, is telling the system how to do random addressing for your terminal's cursor. This is a feature that the Screen Oriented Editor requires. To learn how to support this capability, see the section on GOTOXY.

2. MISCELLANEOUS NOTES FOR SETUP

The STUDENT bit, one of SYSTEM.MISCINFO's data items, should always be set to FALSE while binding in GOTOXY (see below). In general, it should be FALSE unless your system is a Terak being used by inexperienced users, such as a group of students new to programming.

The HAS 8510A bit is always FALSE, unless you are using a Terak 8510a.

On the PDP-11, LSI-11, 8080, and Z-80 systems HAS WORD ORIENTED MACHINE is always FALSE; on 9900, 6800, and GA440 systems, this item is always TRUE.

HAS BYTE FLIPPED MACHINE is FALSE for all current systems except the GA440.

SETUP and the Manual refer to PREFIXED [DELETE CHARACTER]. This is misleading: it actually refers to the backspace function. Read it as PREFIXED [BACKSPACE]. On most terminals it will be FALSE.

Your terminal should be set to run in full duplex, with no auto-echo.

Don't use terminal functions that do a "Delete and close up" on lines or characters -- not all terminals have these functions, and so they are supplied through the Screen Oriented Editor's software.

If you have a DEC VT-52 and a backspace won't move the cursor on the console, this is because you have KEY TO DELETE CHARACTER set to '_', the "rubout character". This is a printing character, so the Operating System does not echo a cursor move; the contents of memory are updated correctly. One workaround is to use the V(erify key to display the actual file contents, but to fix this for good, use SETUP to change KEY TO DELETE CHARACTER to control-H or left-arrow -- BACKSPACE shoulld be set to the same character as well.

On the Hazeltine H1500, ERASE LINE should be set to NUL. Don't set it to the Hazeltine's "Delete line" sequence; this won't work, as cautioned in the paragraph above on "Delete and close up".

3. THE DATA ITEMS IN SYSTEM.MISCINFO

The information in this section is very specific, and you may skip it on first reading. If you have a question about a certain data item, look in this section. Default values are shown, and sometimes suggested values. When no suggested values are given, you should consult your own terminal documentation. The items are ordered according to SETUP's menu. (See Appendix A.)

If you are using a hardcopy terminal or a storage screen rather than a CRT, you can ignore all the data items that are only used by the Screen Oriented Editor and leave them set to their defaults.

Please note that SETUP frequently makes a distinction between a character which is a key on the keyboard, and a character which is sent to the screen from the Pascal system; on some terminals, the same function may be performed by two different characters. On some terminals, the key pressed and the character sent for a given function may be the same, but in any case, when you run SETUP you must be explicit and answer all questions, even if the information is redundant.

There are a few characters which you cannot change with SETUP. These are CARRIAGE RETURN (<ret>), LINE FEED (<lf>), ASCII DLE (control-P), and TAB (control-I). It is assumed that <ret>, <lf>, and TAB are consistent on all terminals. ASCII DLE (data link escape) is used as a blank compression character. When sent to an output file, it is always followed by a byte containing the number of blanks which the output device must insert. If you try to use control-P for any other function, you will run into trouble. More information on DLE is given in the sections below on GOTOXY and SCREENTEST.

BACKSPACE

When sent to the screen, this character should move the cursor one space to the left. Default: ASCII BS.

EDITOR ACCEPT KEY

This key is used by the Screen Oriented Editor. When pressed, it ends the action of a command, and accepts whatever actions were taken. Default: ASCII NUL. Suggested: ASCII ETX (control-C or "Home").

EDITOR ESCAPE KEY

This key is used by the Screen Oriented Editor. It is the opposite of the EDITOR ACCEPT KEY -- when pressed, it ends the action of a command, and ignores whatever actions were taken. Default and Suggested: ASCII ESC (control-[).

ERASE LINE

When sent to the screen, this character erases all the characters on the line that the cursor is on. Default: ASCII NUL.

ERASE SCREEN

When sent to the screen, this character erases the entire screen. Default: ASCII NUL.

ERASE TO END OF LINE

When sent to the screen, this character erases all characters from (and including) the current cursor position to the end of the same line. Default: ASCII NUL.

ERASE TO END OF SCREEN

When sent to the screen, this character erases all characters from (and including) the current cursor position to the end of the screen. Default: ASCII NUL.

HAS 8510A

May be TRUE or FALSE. Should be TRUE if and only if your hardware system is a Terak 8510a. Default: FALSE.

HAS BYTE FLIPPED MACHINE

May be TRUE or FALSE. On PDP-11, LSI-11, 8080, Z-80, 9900, 6800, and 6502 processors this bit is FALSE. On the GA440 system, it is TRUE. In general, it is TRUE only for implementations in which the IPC (Instruction Program Counter) is segment-relative. Default: FALSE.

HAS CLOCK

May be TRUE or FALSE. If the system has a line frequency (60 Hz) clock module, such as the DEC KW11, setting this bit TRUE will allow the Pascal system to optimize disk directory updates. It also allows you to use the TIME intrinsic -- see section 2.1.3 in the Manual. Default: FALSE.

HAS LOWER CASE

May be TRUE or FALSE. It should be TRUE if you do have lower case and want to use it. If you seem stuck in upper case even if this bit is TRUE, remember there is a soft alpha-lock: see KEY TO ALPHA LOCK. Default: FALSE.

HAS RANDOM CURSOR ADDRESSING

May be TRUE or FALSE. If your terminal is not a CRT, this should be FALSE. Default: FALSE.

HAS SLOW TERMINAL

May be TRUE or FALSE. When this bit is TRUE, the system's promptlines and messages are abbreviated. It is suggested that you leave this set at FALSE unless your terminal runs at 600 baud or slower. Default: FALSE.

HAS WORD ORIENTED MACHINE

May be TRUE or FALSE. If sequential addresses on your processor reference sequential 16 bit words, this should be TRUE. For PDP-11, LSI-11, 8080, Z-80, 9900, 6800, and 6502 systems, this should be FALSE. Default: FALSE.

KEY FOR BREAK

When this key is pressed while a program is running, the program will terminate immediately with a runtime error. Default: ASCII NUL. Suggested: a key that is difficult to hit accidentally.

KEY FOR FLUSH

This key may be pressed while the system is sending output (writing to the file OUTPUT). The first time it is pressed, output is no longer displayed, and will be ignored ("flushed") until FLUSH is pressed again. This can be done any number of times; FLUSH functions as a toggle. Note that processing continues while the output is ignored, so using FLUSH causes output to be lost. Default and suggested: ASCII ACK (control-F).

KEY FOR STOP

This key may be pressed while the system is writing to OUTPUT. Like FLUSH, it is a toggle. Pressing it once causes output and processing to stop, pressing it again causes output and processing to resume, and so on. No output is lost; STOP is useful for slowing down a program so the output can be read while it is being sent to the terminal. Default and suggested: ASCII DC3 (control-S).

KEY TO ALPHA LOCK

This character, when sent to the screen, locks the keyboard in upper case (alpha mode). It is usually a key on the keyboard as well. Default: ASCII DC2 (control-R).

KEY TO DELETE CHARACTER

Deletes the character where the cursor is, and moves cursor one character to the left. Default and suggested: ASCII BS (control-H or "Backspace").

KEY TO DELETE LINE

Deletes the line that the cursor is currently on. Default and suggested: ASCII DEL ("Rubout").

KEY TO END FILE

Sets the intrinsic Boolean function EOF to TRUE when pressed while reading from the system input files (either KEYBOARD or INPUT, which come from device CONSOLE:). Default and suggested: ASCII ETX (control-C or "Home").

KEY TO MOVE CURSOR DOWN
KEY TO MOVE CURSOR LEFT
KEY TO MOVE CURSOR RIGHT
KEY TO MOVE CURSOR UP

These keys are recognized by the Screen Oriented Editor, and are used when editing a document to move the cursor about the screen. If your keyboard has a vector pad, we suggest using those keys for these functions. If you have no vector pad, you might select four keys in the same pattern (such as, for example, '.','K',';', and 'O', in that order) and use them as your vector keys, prefixing them or using the corresponding ASCII control codes. Default (in order): ASCII LF, ASCII BS, ASCII FS, ASCII US.

LEAD IN FROM KEYBOARD

On some terminals, pressing certain keys generates a two-character sequence. The first character in these cases must always be a prefix, and must be the same for all such sequences. This data item specifies that prefix. Note that this character is only accepted as a lead in for characters where you have set PREFIXED[<itemname>] to TRUE. An example of this is in Appendix B below. Default: ASCII NUL.

LEAD IN TO SCREEN

Some terminals require a two-character sequence to activate certain functions. If the first character in all these sequences is the same, this data item can specify this prefix. This item is similar to the one above. The prefix is only generated as a lead in for characters where you have set PREFIXED[<itemname>] to TRUE. An example of this is in Appendix B below. Default: ASCII NUL.

MOVE CURSOR HOME

When sent to the terminal, moves the cursor to the upper left hand corner of the screen (position (0,0)). If your terminal doesn't have a character which does this, this data item must be set to CARRIAGE RETURN; you will not be able to use the Screen Oriented Editor. Default: ASCII CR ("Return").

MOVE CURSOR RIGHT

When sent to the terminal, moves the cursor nondestructively one space to the right. If your terminal doesn't have this function, you will not be able to use the Screen Oriented Editor. Default:

MOVE CURSOR UP

When sent to the terminal, moves the cursor vertically up one line. If your terminal doesn't have this function, you won't be able to use the Screen Oriented Editor. Default: ASCII NUL.

NON PRINTING CHARACTER

The character that will be displayed on the screen when a non-printing character is typed or sent to the terminal. Default and suggested: '?'.

PREFIXED[<itemname>]

If any two-character sequence must be generated by a key or sent to the screen, the system will recognize that if you set PREFIXED[<itemname>] to TRUE. See the explanations for LEAD IN FROM KEYBOARD and LEAD IN TO SCREEN. An example of the use of two-character sequences is given in Appendix B.

SCREEN HEIGHT

The number of lines in your display screen, starting from 1. If you are using a hardcopy terminal, this should be set to 0. Default: 24 (base ten).

SCREEN WIDTH

The number of characters in one line on your display, starting from 1. Default: 80 (base ten).

STUDENT

May be TRUE or FALSE. Should only be set to TRUE on Terak systems that are being used by inexperienced programmers. If STUDENT is true, an error detected while compiling sends the programmer into the Editor automatically (the programmer normally has the option to continue compiling). This bit must be FALSE when you run BINDER; see the next section. Default: FALSE.

VERTICAL MOVE DELAY

May be a decimal integer from 0 to 11. Many terminals require a delay after vertical cursor movements. This delay allows the movement to be completed before another character is sent. This data item specifies the number of nulls that the system sends to the terminal after every CARRIAGE RETURN, ERASE TO END OF LINE, ERASE TO END OF SCREEN, and MOVE CURSOR UP. Default: 5 (base ten).

SECTION 4.2.3
GOTOXY

When you have tailored SYSTEM.MISCINFO with SETUP, you should consider writing your own GOTOXY. GOTOXY is a Pascal procedure embedded in the Operating System. It provides random addressing for your terminal's cursor. There is a GOTOXY that is provided with the system we ship, (the source for this code, along with other examples, is in Appendix C below), but as it is a general routine for any terminal, it is not very fast. When you create your own GOTOXY, you will write a Pascal procedure, compile it, then bind it into the Operating System using the system utility BINDER.CODE.

If you are not ready to write your own GOTOXY, you should skip down to the last section of this document, which describes SCREENTEST.

If you intend to do all your work on a line-oriented terminal, you never need to use GOTOXY at all.

Before you write your own GOTOXY, you should understand the UCSD Pascal I/O intrinsics: these are described in Section 2.1.2 of the Manual. In Appendix C are a few sample versions of GOTOXY, including the source for the GOTOXY code which comes with the system, and the SAMPLEGOTO.TEXT that is also on your system disk. You should look this appendix over.

1. WRITING YOUR OWN GOTOXY

1.1 A DISCUSSION

You may write this procedure using either YALOE or the Screen Oriented Editor, whichever you find more convenient.

The purpose and the calling protocol of GOTOXY are quite simple. The procedure is given two parameters, X and Y. They must be in that order, and they must be of type INTEGER. The procedure should position the terminal's cursor at co-ordinates (X,Y), where (0,0) is home (the upper left hand corner of the screen). That is all it should do.

To get your GOTOXY to run at all, there are a few things that are required.

First, the name of your procedure cannot be GOTOXY, as it must be compiled at a level where this name is predeclared. Call it MYGOTOXY or something else obvious and unique.

Second, you must include the pseudo-comment {$U-,S+}. These options cause your procedure to be compiled at the system level, which is where it needs to be -- it will become part of the Operating System -- and allow swapping while you compile. This comment must be the first line of your source code.

Finally, your source must end with a BEGIN-END pair followed by a period. This dummy program body is necessary for the compiler only -- BINDER disregards it.

Your procedure should check that the values of X and Y are within bounds. If they are off the screen, change them to a value that is on the screen (such as the nearest location along the border -- this is what all the sample procedures do).

You will need to move the cursor by a WRITE to the terminal, a repeated set of WRITEs within a loop, or a UNITWRITE of a vector. Using UNITWRITE is recommended: it can speed up your terminal handling by about 10%.

To summarize, your GOTOXY should contain, in order:

1. The pseudo-comment {$U-,S+};
2. In the program body, a check to make sure that X and Y are on the screen,
3. A section that fills an array with all the characters you must send to the terminal, and
4. The actual write to the terminal, preferably with UNITWRITE.

Please note: some terminals take a bias on X and Y. That is, for example, sending (X+32,Y+32) actually positions the cursor at (X,Y). If your terminal is capable of this, you should include these offsets in your procedure. This will eliminate any problems you might run into with the ASCII DLE (control-P) character, which is ALWAYS interpreted as a blank-compression character. You don't want to send this value as a cursor control character. See the section below on SCREENTEST.

The following section contains a more detailed description of GOTOXY. Appendix C contains specific examples for a variety of terminals.

1.2 A RECIPE FOR GOTOXY

This section walks you through a sample GOTOXY. It demonstrates the best way of writing a GOTOXY. To see some more specific examples, see Appendix C.

The sample program here is commented like a Pascal program.

```
{$U-,S+}            { ALWAYS include this pseudo-comment. }
PROGRAM NEWGOTOXY;

PROCEDURE MYGOTOXY(X,Y: INTEGER);

{ Note that neither the program nor the procedure can
  actually be called GOTOXY -- that name is predeclared. }

CONST  TELL_LENGTH_MINUS_1 = 3,
       OFFSET = 32;
{ You may have to change these, depending on your terminal. }

VAR    TELL: PACKED ARRAY [0..TELL_LENGTH_MINUS_1]
             OF 0..255;

BEGIN
  IF X>79 THEN X:=79
    ELSE IF X<0 THEN X:=0;
  IF Y>23 THEN Y:=23
    ELSE IF Y<0 THEN Y:=0;
  {  This range-checking is necessary.  The actual
     screenwidth and height may be different for you. }

  { These first elements of TELL must contain  }
  { the characters which tell your terminal to }
  { position the cursor at (X,Y):              }
  { fill in the blanks...                      }
  TELL[0] := _____;
  TELL[1] := _____;
  ...
  { The actual X and Y values are usually the    }
  { last things in the array;                    }
  { the order may be different on your terminal. }
  TELL[TELL_LENGTH_MINUS_1 - 1] := Y+OFFSET;
  TELL[TELL_LENGTH_MINUS_1]     := X+OFFSET;

  UNITWRITE(1,TELL,TELL_LENGTH_MINUS_1 + 1)
END;

BEGIN
{ This is a dummy main program. }
END.
```

2. RUNNING BINDER

The first thing to do, once you have written your own GOTOXY, is to compile it to a codefile. Any filename will do, provided its suffix is .CODE. Choose a name you will remember.

You may run into some problems while compiling. The most common error while compiling your own GOTOXY is naming your procedure GOTOXY, which is a reserved name at the system level.

Once you have a valid codefile, type X to eXecute the BINDER utility, and respond to the prompt with "BINDER<ret>". It will prompt you for the name of the GOTOXY codefile, and you should respond with whatever name you have given that file, followed by <ret>.

When BINDER has finished running, you should always reboot. This causes the new procedure to actually be used. If you have been using YALOE for SYSTEM.EDITOR, now is the time to backup YALOE by going into the Filer, changing SYSTEM.EDITOR to YALOE.CODE, then changing your Screen Oriented Editor's codefile name to SYSTEM.EDITOR.

BINDER will put your new GOTOXY into the Operating System on the default disk. This is generally the disk you booted on, but if you have used the Filer's P(refix command, make sure you are altering the disk you want to alter.

You should also make sure that the STUDENT bit in SYSTEM.MISCINFO is set to FALSE -- otherwise GOTOXY binding will not work, and you will get the message "No proc in seg table" when you try to reboot.

Once BINDER has been run and the system rebooted, you should run SCREENTEST to make sure the Screen Oriented Editor interface will work. SCREENTEST is described immediately below.

Now that you have changed your SYSTEM.MISCINFO with SETUP (or your GOTOXY, or both), you will want to test the results. SCREENTEST is a utility which accomplishes that. Like SETUP, it is largely self-explanatory. SCREENTEST checks that the Interpreter and Operating System are sending and receiving characters correctly, that the control keys are set up correctly, and that the Screen Oriented Editor will interface to the terminal as it is supposed to.

When you run SCREENTEST, it will display patterns on the screen and ask you if they are correct. You will need to be seated at your terminal while SCREENTEST is running; it takes roughly five minutes.

SCREENTEST will also output a report of errors to any file you specify. If you do encounter problems, you will need this report to help track them down, especially if you call SofTech MicroSystems' Pascal Support.

1. RUNNING SCREENTEST

Type X for eXecute, and enter "SCREENTEST<ret>". It will respond by displaying a heading, telling you that all questions must be answered with either "Y" or "N" (either upper or lower case; all other characters are ignored), and will then prompt you for the name of an error log file.

If you hit RETURN instead of specifying a log file name, no error report will be generated. You may want to do this if you are running SCREENTEST for the first time and don't anticipate any problems. If you do have trouble, you can run it again, this time with a log. Sending the log to "PRINTER:" may suit your needs if you have a hardcopy device, otherwise you can save it on a disk file named "LOG.TEXT" or something similar. (The .TEXT suffix is necessary if you want to look at it with the Editor.)

If your terminal is set up correctly, you should be able to answer 'Y' to all of the yes/no questions that SCREENTEST asks. If there is any problem with the questions about individual characters, SCREENTEST will tell you immediately. The log file will also contain a record of all problems. A sample log is in Appendix E.

2. RESULTS OF SCREENTEST

SCREENTEST consists of twelve individual tests. Their names follow:

test_basic
test_clr_screen
test_gotoxy
test_clr_line
test_erase_eol
test_etoeos
test_home
test_single_vectors
test_scroll
test_DLE_expansion
test_keyboard
test_normal_keys

Each of these tests may generate error messages. While the text of each error message is fairly clear, some further explanation follows. The error messages are grouped by the nature of the problems -- what you must check in order to solve them. They are further grouped under the name of the test that generates them. This information is included in the error log. If you find yourself at a loss and decide to consult Pascal Support, you will need to refer to this log.

2.1 PROBLEMS THAT CAN BE FIXED BY CHANGING SETUP

If you get any of these error messages, check your SETUP values. To the right of each error message listed below is a suggestion as to which key or character value might be in error. These suggestions won't always pinpoint your problem, but they will tell you what you should check first. It may be the case that changing SETUP does not fix your problem. Some special cases are described at the end of this section. If these don't cover your particular problem, you should probably ask for help.

test_clr_screen:

```
screen not cleared    -> is ERASE SCREEN OK?
cursor not left at (0,0) afterwards
                      -> is MOVE CURSOR HOME OK?
```

test_clr_line:

```
didn't clear enough - (x,y)
   (where x and y are the cursor co-ordinates)
                      -> is ERASE LINE OK?
Clearing one line affected another
                      -> is ERASE LINE OK?
```

test_erase_eol:

```
     sc_erase_to_eol didn't work
                          -> is ERASE TO END OF LINE OK?
```

test_etoeos:

```
     sc_eras_eos didn't work
                          -> is ERASE TO END OF SCREEN OK?
```

test_home:

```
     cursor didn't go home
                          -> is MOVE CURSOR HOME OK?
```

test_single_vectors:

```
     sc_right didn't work -> is MOVE CURSOR RIGHT OK?
     sc_left didn't work  -> is BACKSPACE OK?
     sc_up didn't work    -> is MOVE CURSOR UP OK?
     sc_down didn't work  -> this shouldn't happen;
                             call Pascal Support!
```

test_keyboard:

```
     <key> not correct    -> is <key> OK?  <key> means one of
                             the following:
                             KEY TO MOVE CURSOR DOWN
                             KEY TO MOVE CURSOR LEFT
                             KEY TO MOVE CURSOR RIGHT
                             KEY TO MOVE CURSOR UP
                             BACKSPACE
                             EDITOR ACCEPT KEY
                             EDITOR ESCAPE KEY
                             KEY TO DELETE LINE
                             KEY TO END FILE
```

test_normal_keys:

```
     Can't type these - <list>
                          -> <list> means a list of any standard
                             printing characters;  this usually
                             means that a standard character is
                             being interpreted as a special key,
                             which usually happens when
                             HASPREFIX is incorrect -- it should
                             be FALSE for a key which needs
                             no prefix, or TRUE for a key which
                             does need one;  check your own
                             terminal manual;
```

2.2 PROBLEMS THAT CAN BE FIXED BY CHANGING GOTOXY

test_gotoxy:

```
     gotoxy(0,0) did not go home
     gotoxy(screenwidth-1,screenwidth) not ok
     box not correctly drawn
     exhaustive_gotoxy_check: first pass not ok
     exhaustive_gotoxy_check: top line not ok
                           -> all these problems relate to your
                              GOTOXY procedure;  if you find any
                              discrepancies, you will have to
                              change it;  refer to the previous
                              section in this document for a
                              description of using GOTOXY,
                              and to the first paragraph in
                              the miscellaneous notes below;
```

2.3 OTHER PROBLEMS

test_basic:

not all characters written out
-> there is a problem with the Pascal system intrinsic UNITWRITE, or, if you are using the adaptable system, with SBIOS. You should call Pascal Support; disregard the rest of SCREENTEST's results until this particular problem is cleared up;

test_scroll:

sc_down at bottom didn't scroll properly
-> there is a note below about scrolling;

test_DLE_expansion:

expansion not happening properly
-> there is a problem in your Interpreter's terminal handling; this may be hardware-related; it is still possible to run with improper DLE expansion -- you may encounter off-by-one errors and the like in your output and your editing; this is the case with Terak systems; DLE is an ASCII character used as a blank-compression code to save space in output strings;

3. MISCELLANEOUS NOTES ON SCREENTEST PROBLEMS

System II.0 interprets an ASCII DLE or chr(16) (base ten) as a blank compression code -- this is its standard use. It can lead to problems if GOTOXY ever writes out a chr(16) as an X or Y value. If you run into this problem, check whether your terminal can handle an offset on X and Y values, that is, whether sending it X+32 and Y+32 will position the cursor at (X,Y) (the value 32 is just an example). If so, this will fix your problem. If not, you will have to modify GOTOXY so it catches this situation; see above.

ERASE LINE will have difficulty if there are bugs in the screen emulator for memory mapped screens. This is applicable primarily to Terak systems. In particular, Teraks have trouble with blank-compression sequences (DLE-expansions) of 64 or longer.

Some terminals will not scroll at all, or scroll two lines at a time. The II.0 System's Screen Oriented Editor unfortunately cannot handle these terminals -- you must use YALOE for SYSTEM.EDITOR.

SECTION 4.2.5

Appendix A
SETUP MENU AND DEFAULTS

In the defaults shown below, 'T' means true and 'F' means false as per the input conventions in SETUP. The numbers shown are in base ten, literal characters are quoted, and ASCII abbreviations are used for nonprinting characters. When you use SETUP, these values are shown in several formats, so the meaning is clear.

BACKSPACE	BS
EDITOR ACCEPT KEY	NUL
EDITOR ESCAPE KEY	ESC
ERASE LINE	NUL
ERASE SCREEN	NUL
ERASE TO END OF LINE	NUL
ERASE TO END OF SCREEN	NUL
HAS 8510A	F
HAS BYTE FLIPPED MACHINE	F
HAS CLOCK	F
HAS LOWER CASE	F
HAS RANDOM CURSOR ADDRESSING	F
HAS SLOW TERMINAL	F
HAS WORD ORIENTED MACHINE	F
KEY FOR BREAK	NUL
KEY FOR FLUSH	ACK
KEY FOR STOP	DC3
KEY TO ALPHA LOCK	DC2
KEY TO DELETE CHARACTER	BS
KEY TO DELETE LINE	DEL
KEY TO END FILE	ETX
KEY TO MOVE CURSOR DOWN	LF
KEY TO MOVE CURSOR LEFT	BS
KEY TO MOVE CURSOR RIGHT	FS
KEY TO MOVE CURSOR UP	US
LEAD IN FROM KEYBOARD	NUL
LEAD IN TO SCREEN	NUL
MOVE CURSOR HOME	CR
MOVE CURSOR RIGHT	'!'
MOVE CURSOR UP	NUL
NON PRINTING CHARACTER	'?'
PREFIXED [DELETE CHARACTER]	F
PREFIXED [EDITOR ACCEPT KEY]	F
PREFIXED [EDITOR ESCAPE KEY]	F
PREFIXED [ERASE LINE]	F
PREFIXED [ERASE SCREEN]	F
PREFIXED [ERASE TO END OF LINE]	F
PREFIXED [ERASE TO END OF SCREEN]	F
PREFIXED [KEY TO DELETE CHARACTER]	F
PREFIXED [KEY TO DELETE LINE]	F
PREFIXED [KEY TO MOVE CURSOR DOWN]	F
PREFIXED [KEY TO MOVE CURSOR LEFT]	F
PREFIXED [KEY TO MOVE CURSOR RIGHT]	F
PREFIXED [KEY TO MOVE CURSOR UP]	F
PREFIXED [MOVE CURSOR HOME]	F
PREFIXED [MOVE CURSOR RIGHT]	F
PREFIXED [MOVE CURSOR UP]	F

PREFIXED [NON PRINTING CHARACTER]	F
SCREEN HEIGHT	24
SCREEN WIDTH	80
STUDENT	F
VERTICAL MOVE DELAY	5

Appendix B
SAMPLE SETUPS FOR SOME TERMINALS

Here is a list of SYSTEM.MISCINFO data items followed by some sample values for four popular terminals. Some items in the SETUP menu haven't been included; these are data items that refer to your processor configuration, not your terminal. Some sample items have been left blank -- we don't yet have complete specifications for these terminals.

These examples represent what we consider reasonable layouts for a few different keyboards, but we don't guarantee that they work for your particular hardware, or match your individual taste.

Terminals:	LSI ADM-3A	HAZELTINE 1500/1510	SOROC IQ120	HEATH H19
Data Items:				
BACKSPACE	left-arrow	backspace	ctrl-H	ctrl-H
EDITOR ACCEPT KEY	ctrl-C	ctrl-C	home	ctrl-C
EDITOR ESCAPE KEY	esc	esc	esc	ctrl-[
ERASE LINE	NUL		NUL	1
ERASE SCREEN	ctrl-Z		'*'	E
ERASE TO END OF LINE	NUL		T	K
ERASE TO END OF SCRN	NUL		Y	J
HAS LOWER CASE	TRUE	TRUE	TRUE	TRUE
HAS RAND CURS ADDR	TRUE	TRUE	TRUE	TRUE
HAS SLOW TERMINAL	FALSE	FALSE	FALSE	FALSE
KEY FOR BREAK	ctrl-B *	break **	break	break
KEY FOR FLUSH	ctrl-F	ctrl-F	ctrl-F	ctrl-F
KEY FOR STOP	ctrl-S	ctrl-S	ctrl-S	ctrl-S
KEY TO ALPHA LOCK	ctrl-R	ctrl-R	ctrl-R	ctrl-R
KEY TO DELETE CHAR	ctrl-H	backspace	l-arrow	ctrl-H
KEY TO DELETE LINE	rubout	shift-DEL	rubout	DEL
KEY TO END FILE	ctrl-C	ctrl-C	ctrl-C	ctrl-C
KEY TO MV CURS DOWN	ctrl-J	ctrl-K	d-arrow	B
KEY TO MV CURS LEFT	ctrl-H	backspace	l-arrow	D
KEY TO MV CURS RGHT	ctrl-L	ctrl-P	r-arrow	C
KEY TO MV CURS UP	ctrl-K	ctrl-L	u-arrow	A
LEAD IN FROM KEYBD	NUL		NUL	ESC
LEAD IN TO SCREEN	NUL		ESC	ESC
MOVE CURSOR HOME	ctrl-^		ctrl-^	H
MOVE CURSOR RIGHT	ctrl-L	ctrl-P	r-arrow	C
MOVE CURSOR UP	ctrl-K	ctrl-L	u-arrow	A
NON PRINTING CHAR	'?'	'?'	'?'	'?'
PREF [DELETE CHAR]	FALSE	FALSE	FALSE	FALSE
PREF [ED ACCEPT KEY]	FALSE	FALSE	FALSE	FALSE
PREF [ED ESCAPE KEY]	FALSE	FALSE	FALSE	TRUE
PREF [ERASE LINE]	FALSE		FALSE	TRUE
PREF [ERASE SCREEN]	FALSE		TRUE	TRUE
PREF [ERASE TO EOLN]	FALSE		TRUE	TRUE
PREF [ERSE TO EOSCN]	FALSE		TRUE	TRUE
PREF [KEY DEL CHAR]	FALSE	FALSE	FALSE	FALSE
PREF [KEY DEL LINE]	FALSE	FALSE	FALSE	TRUE
PREF [KEY MV CRS DN]	FALSE	FALSE	FALSE	TRUE
PREF [KEY MV CRS LT]	FALSE	FALSE	FALSE	TRUE

SAMPLE SETUPS FOR SOME TERMINALS

PREF [KEY MV CRS RT]	FALSE	FALSE	FALSE	TRUE
PREF [KEY MV CRS UP]	FALSE	FALSE	FALSE	TRUE
PREF [MOVE CRS HOME]	FALSE		FALSE	TRUE
PREF [MOVE CURS RT]	FALSE	FALSE	FALSE	TRUE
PREF [MOVE CURS UP]	FALSE	FALSE	FALSE	TRUE
PREF [NONPRINT CHAR]	FALSE	FALSE	FALSE	FALSE
SCREEN HEIGHT	24	24	24	24
SCREEN WIDTH	80	80	80	80
STUDENT	FALSE	FALSE	FALSE	FALSE
VERTICAL MOVE DELAY	5	5	10	10

* The BREAK key can also be used, but it's perilously close to RETURN.
** Break is also control-@ on Hazeltines.

Appendix C
GOTOXY SOURCE EXAMPLES

The following example is shipped on your system disk as SAMPLEGOTO.TEXT. It is about as simple a GOTOXY as can be written. It is NOT the code which is shipped in your Operating System: that is the next example, which on one hand is a much more general program, and on the other hand is also much longer. Since GOTOXY is a frequently used I/O routine, you want it to be efficient, and so it should be tailored to your particular terminal. This brief example works for a DEC VT-52. For an efficient example, see the Datamedia sample.

```
(*The following is a sample gotoxy procedure for the VT-52*)
(*$U-*)
PROGRAM DUMMY;
PROCEDURE FGOTOXY(X,Y:INTEGER);
BEGIN
  IF X<0 THEN X:=0;
  IF X>79 THEN X:=79;
  IF Y<0 THEN Y:=0;
  IF Y>23 THEN Y:=23;
  WRITE (CHR(27),'Y',CHR(Y+32),CHR(X+32));
END;
BEGIN
END.
```

GOTOXY SOURCE EXAMPLES

This is the source for the code that is actually in your Operating System when you receive your Pascal system. It is replaced when you run BINDER.CODE. This is a general GOTOXY for any terminal which has a cursor. It is not efficient, but it is worth looking over, especially to note the use of MOVELEFT, FILLCHAR, and UNITWRITE.

```
{$S+}
{$U-}
{$Iglobals.text}    {globals includes the SYSTEM.MISCINFO data.}
Procedure pan_gotoxy(x,y: Integer);
Const
  home = 4;
  ndfs = 1;
Var
  i: Integer;
  buffer: Packed Array[0..183] Of Char;
Begin
  With syscom^ Do    {syscom is in globals.}
    Begin            {All these operations depend on the}
      i := 0;        {SYSTEM.MISCINFO data.}
      If x < 0 Then
        x := 0
      Else If x > crtinfo.width Then
        x := crtinfo.width;
      If y < 0 Then
        y := 0
      Else If y > crtinfo.height Then
        y := crtinfo.height;
      If crtctrl.prefixed[home] Then
        Begin
          buffer[i] := crtctrl.escape;
          i := i + 1;
        End;
      buffer[i] := crtctrl.home;
      i := i + 1;
      If Not crtctrl.prefixed[ndfs] Then
        Begin
          If crtctrl.ndfs = Chr(0){NUL} Then
            Fillchar(buffer[i],x,' ')
          Else
            Fillchar(buffer[i],x,crtctrl.ndfs);
          i := i + x;
        End
      Else
        If x > 0 Then
          Begin
            buffer[i] := crtctrl.escape;
            buffer[i + 1] := crtctrl.ndfs;
            Moveleft(buffer[i],buffer[i+2],x+x-2);
            i := i + x + x;
          End;
      Fillchar(buffer[i],y,Chr(10){LF});
      i := i + y;
```

```
      Unitwrite(2,buffer[0],i);
    End;
End;

Begin {dummy}
End.
```

GOTOXY SOURCE EXAMPLES

This example appears in the Manual (Section 4.7). It works for a DEC VT-50. This version uses WRITEs embedded in WHILE loops, and is not fast.

```
{$U-,S+} {The pseudo-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 the 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 program is needed to keep the Pascal
    compiler happy about having complete programs to compile.
    Only the code for the above procedure is used by
    BINDER to add to SYSTEM.PASCAL.}
END.
```

This example is for a Datamedia 1520, and demonstrates the quickest form of GOTOXY: using a UNITWRITE to send one single command stream to the terminal. As mentioned above, this method can speed up your terminal I/O by as much as 10%; we recommend it.

```
{$U-,S+}
PROCEDURE ITSGOTOXY(X,Y: INTEGER);
VAR
  T: PACKED ARRAY[0..2] OF CHAR;
BEGIN
  T[0] := CHR(30);  {chr(30) is an ASCII RS, which is Datamedia's
                     absolute cursor address flag.}

   {Set appropriate character for x coordinate.}
  IF X < 0 THEN T[1] := CHR(32)     {Note the offset of 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.}
  UNITWRITE(1,T,3)     {1 is the device number of CONSOLE:}
END;

BEGIN  {Just a dummy program body.}
END.
```

GOTOXY SOURCE EXAMPLES

Here are two more examples using UNITWRITE. They are for a Soroc and a Hazeltine terminal, respectively.

```
(*$U-,S+*)
PROGRAM NEWGOTOXY;

PROCEDURE AGOTOXY(X,Y: INTEGER);

(* FOR A SOROC IQ 120 *)

VAR TELL: PACKED ARRAY [0..3] OF 0..255;

BEGIN
  IF X>79 THEN X:=79
    ELSE IF X<0 THEN X:=0;
  IF Y>23 THEN Y:=23
    ELSE IF Y<0 THEN Y:=0;
  TELL[0] := 27;              (* LEAD-IN FOR SOROCS *)
  TELL[1] := 17;              (* DC1 *)
  TELL[2] := 32+Y;            (* NOTE THE OFFSET *)
  TELL[3] := 32+X;
  UNITWRITE(1,TELL,4)
END;

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

```
{$U-,S+}
Program goxy(x,y: integer);

Procedure agotoxy(x,y: integer);

{gotoxy for the Hazeltine 1500 and 1510}

var tell: packed array [0..3] of 0..255;

Begin
   if x>79 then x:=79
      else if x<0 then x:=0;
   if y>23 then y:=23
      else if y<0 then y:=0;
   tell[0] := 126;            {the lead-in for a Hazeltine}
   tell[1] := ord('=');       {also a DC1}
   if x<30 then
      tell[2] := x+96         {different offset for these terminals}
   else
      tell[2] := x;
   tell[3] := y+96;
   unitwrite(1,tell,4)
End;

Begin   {dummy}   End.
```

Appendix D
SAMPLE SETUP SESSION WITH COMMENTS

The following is a sample of part of a session with SETUP. The data is being changed from the system defaults to the specifications for a Soroc terminal, as in Appendix B above. All text enclosed in square brackets [like this] is user input, and all text enclosed in curly brackets {like this} is commentary. Angle brackets <these> are used to enclose the names of non-printing characters {like <ret>}. All else is SETUP's output to the terminal.

```
{To begin, you must eXecute SETUP}

[XSETUP<ret>]
INITIALIZING.............................
SETUP: C(HANGE T(EACH H(ELP Q(UIT

{H(ELP tells you about the other commands, and T(EACH
 describes the use of SETUP.  Now is the most profitable
 time to use these commands.
 Suppose you have read H(ELP and T(EACH, and decide
 to change data items by going through the menu.
 You must hit C for C(HANGE.}

[C]   {Note: these single-character commands don't echo.}
CHANGE: S(INGLE) P(ROMPTED) R(ADIX)
    H(ELP) Q(UIT)

{H(ELP) describes the commands on this particular line,
 R(ADIX) allows you to change the base of the numbers
 you enter, and Q(UIT) returns you to the SETUP: prompt.
 What you want to do now is go through the prompted menu.}

[P]



FIELD NAME = BACKSPACE
 OCTAL  DECIMAL  HEXADECIMAL  ASCII  CONTROL
    10       8            8     BS      ^H
WANT TO CHANGE THIS VALUE? (Y,N,!)
[<ret>]
WANT TO CHANGE THIS VALUE? (Y,N,!)

{<ret> or <space> will cause this prompt to be repeated.
 ! causes an escape to the CHANGE: prompt.
 Since control-H (^H) is indeed the Soroc's backspace,
 you want to go on.}

[N]
```

SAMPLE SETUP SESSION WITH COMMENTS

```
FIELD NAME = EDITOR ACCEPT KEY
 OCTAL  DECIMAL  HEXADECIMAL  ASCII  CONTROL
     0        0            0  NUL      ^@
WANT TO CHANGE THIS VALUE? (Y,N,!)
[Y]
NEW VALUE: [<home>]

{When <home> or any other non-printing key
 is pressed, ? is displayed.}

 OCTAL  DECIMAL  HEXADECIMAL  ASCII  CONTROL
     3        3            3  ETX      ^C
WANT TO CHANGE THIS VALUE? (Y,N,!)
[N]



FIELD NAME = EDITOR ESCAPE KEY
 OCTAL  DECIMAL  HEXADECIMAL  ASCII  CONTROL
     0        0            0  NUL      ^@
WANT TO CHANGE THIS VALUE (Y,N,!)
[Y]
NEW VALUE: [<ret>]

{Any unexpected input here causes the
 relevant section of T(EACH to be output,
 followed by this:}
C(ONTINUE)
{All characters are ignored except C, and
 then the prompt is repeated.}

[C]
NEW VALUE: [<rubout>]  {Again, a ? is echoed.}
 OCTAL  DECIMAL  HEXADECIMAL  ASCII
   177      127           7F  DEL
WANT TO CHANGE THIS VALUE? (Y,N,!)

{(Note that there is no corresponding control key.)
 DEL is not the key you meant, so you must
 change it again.}

[Y]
NEW VALUE: [<esc>]  {? is echoed.}
 OCTAL  DECIMAL  HEXADECIMAL  ASCII  CONTROL
    33       27           1B  ESC      ^[
WANT TO CHANGE THIS VALUE? (Y,N,!)
[N]  {This is what it should be.}
```

```
{The menu continues in this way for the rest of
 the data items.  Suppose you have gone ahead and
 answered all of the questions according to the
 Soroc specifications.  After the last data item,
 you again get the menu:}

CHANGE: S(INGLE) P(ROMPTED) R(ADIX)
    H(ELP) Q(UIT)

{You realize that you left the prefix for
 ERASE LINE at FALSE, when it should be
 TRUE.  You want to change just this one
 data item.}

[S]  {For S(INGLE)}
NAME OF FIELD: [PREFIXED [ERASE]]
DIDN'T FIND PREFIXED [ERASE]   {Oops}
NAME OF FIELD: [PREFIXED [ERASE LINE]]

FIELD NAME = PREFIXED [ERASE LINE]
 CURRENT VALUE IS FALSE
WANT TO CHANGE THIS VALUE? (Y,N,!)
[Y]
NEW VALUE: [TRUE]  {T would also work.}
 CURRENT VALUE IS TRUE
WANT TO CHANGE THIS VALUE? (Y,N,!)
[N]

CHANGE: S(INGLE) P(ROMPTED) R(ADIX)
    H(ELP) Q(UIT)
[Q]
SETUP: C(HANGE T(EACH H(ELP Q(UIT [D2]
[Q]  {You're through changing data now.}
QUIT: D(ISK) OR M(EMORY) UPDATE,
          R(ETURN) H(ELP) E(XIT)

{You want to do a disk update to create
 NEW.MISCINFO on your disk for future use.}

[D]
QUIT: D(ISK) OR M(EMORY) UPDATE,
          R(ETURN) H(ELP) E(XIT)
[E]

{And now you're done.  The Pascal system prompt
 will appear.}
```

Appendix E
SAMPLE SCREENTEST LOG

This is a sample of a SCREENTEST log for a terminal that has some problems.

```
    1 test_DLE_expansion: expansion not happening properly
    2 test_DLE_expansion: expansion not happening properly
    3 test_DLE_expansion: expansion not happening properly
    4 test_DLE_expansion: expansion not happening properly
    5 test_DLE_expansion: expansion not happening properly
    6 test_DLE_expansion: expansion not happening properly
    7 test_DLE_expansion: expansion not happening properly
    8 test_DLE_expansion: expansion not happening properly
    9 test_keyboard: backspace key not correct
   10 test_keyboard: line feed key not correct

***** End Diagnostic; 10 errors encountered.
```

-- Notes --

BOOTSTRAP COPIER * Section 4.3

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 --

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)).

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 --

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 4.7 *
********** **********

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.

4.7.1 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.II and requires the file OPCODES.II0 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                       EB
      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.

4.7.2 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.

4.7.3 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 4.8 *
********************** ***************
```

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 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)

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

********** **********
* TABLE 1 * * EXECUTION ERRORS *
********** **********

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

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

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 (floppies , hard disks, etc)

-- Notes --

********** **********
* TABLE 4 * * RESERVED WORDS *
********** **********

STANDARD PASCAL RESERVED WORDS	UCSD RESERVED WORDS
AND	SEGMENT
ARRAY	SEPARATE
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 *

```
 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: Index type must not be integer.
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
407: Block0 overflow (codefile overflow)

```
********** ****************************
* TABLE 6 * * ASSEMBLER SYNTAX ERRORS *
********** ****************************
```

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 volume on-line
56: No file, no such file on volume
57: Duplicate file
58: Not closed, attempt to open an open file
59: Not open, attempt to access a closed file
60: Bad format, error in reading real or integer
61: Nested macro definitions not allowed
62: '=' or ' ' expected
63: May not EQU to undefined labels
64: Must declare .ABSOLUTE before first .PROC

Z-80 based machines
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

PDP-11 based machines
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: Unimplemented instruction
84: Must branch backwards to label

8080 based machines have no specific error messages.

6502 based machines
Specific error messages:

76: Index register required
77: "X" or "Y" expected
78: Zero-page address required
79: Illegal use of register
80: Index register expected
81: Ill-formed operand
82: "X" expected for indexed addressing
83: Must use "X" register

6800 based machines
Specific error messages:

76: "X" expected for indexed addressing
77: "A" or "B" expected

9900 based machines
Specific error messages:

76: Illegal immediate operand
77: Index must be WR
78: Close paren ")" expected
79: Indirect and autoincr must be WR
80: Autoincr must be WR indirect
81: Comma "," expected
82: No operand allowed
83: Illegal map file

— Notes —

********** **********
* TABLE 7 * * American Standard Code for Information Interchange *
********** **********

0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	`
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	68	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	89	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	¦
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

-- Notes --

********** **********
* TABLE 8 * * P-MACHINE OP-CODES *
********** **********

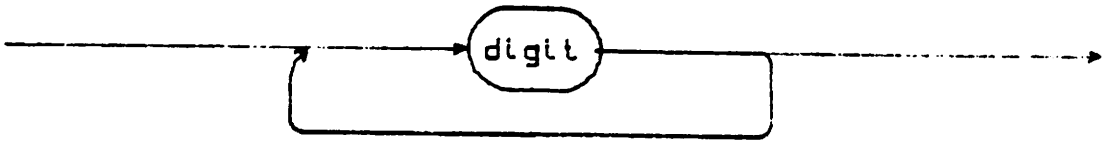
0	000	00	SLDC	0
1	001	01	SLDC	1
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
126	176	7E	SLDC	126
127	177	7F	SLDC	127

128	200	80	ABI
129	201	81	ABR
130	202	82	ADI
131	203	83	ADR
132	204	84	AND
133	205	85	DIF
134	206	86	DVI
135	207	87	DVR
136	210	88	CHK
137	211	89	FLO
138	212	8A	FLT
139	213	8B	INN
140	214	8C	INT
141	215	8D	IOR
142	216	8E	MOD
143	217	8F	MPI
144	220	90	MPR
145	221	91	NGI
146	222	92	NGR
147	223	93	NOT
148	224	94	SRS
149	225	95	SBI
150	226	96	SBR
151	227	97	SGS
152	230	98	SQI
153	231	99	SQR
154	232	9A	STO
155	233	9B	IXS
156	234	9C	UNI
157	235	9D	
158	236	9E	CSP
159	237	9F	LDCN
160	240	A0	ADJ
161	241	A1	FJP
162	242	A2	INC
163	243	A3	IND
164	244	A4	IXA
165	245	A5	LAO
166	246	A6	LSA
171	253	AB	SRO
172	254	AC	XJP
173	255	AD	RNP
174	256	AE	CIP
175	257	AF	EQU
176	260	B0	GEQ
177	261	B1	GRT
178	262	B2	LDA
179	263	B3	LDC
180	264	B4	LEQ
181	265	B5	LES
182	266	B6	LOD
183	267	B7	NEQ
184	270	B8	STR
185	271	B9	UJP
186	272	BA	LDP
187	273	BB	STP
188	274	BC	LDM
189	275	BD	STM
190	276	BE	LDB
191	277	BF	STB
192	300	C0	IXP
193	301	C1	RBP
194	302	C2	CBP
195	303	C3	EQUI
196	304	C4	GEQI
197	305	C5	GRTI
198	306	C6	LLA
199	307	C7	LDCI
200	310	C8	LEQI
201	311	C9	LESI
202	312	CA	LDL
203	313	CB	NEQI
204	314	CC	STL
205	315	CD	CXP
206	316	CE	CLP
207	317	CF	CGP
208	320	D0	LPA
209	321	D1	
214	326	D6	XIT
215	327	D7	NOP
216	330	D8	SLDL 1
217	331	D9	SLDL 2
218	332	DA	SLDL 3
219	333	DB	SLDL 4
220	334	DC	SLDL 5
221	335	DD	SLDL 6
222	336	DE	SLDL 7
223	337	DF	SLDL 8
224	340	E0	SLDL 9
225	341	E1	SLDL 10
226	342	E2	SLDL 11
227	343	E3	SLDL 12
228	344	E4	SLDL 13
229	345	E5	SLDL 14
230	346	E6	SLDL 15
231	347	E7	SLDL 16
232	350	E8	SLDO 1
233	351	E9	SLDO 2
234	352	EA	SLDO 3
235	353	EB	SLDO 4
236	354	EC	SLDO 5
237	355	ED	SLDO 6
238	356	EE	SLDO 7
239	357	EF	SLDO 8
240	360	F0	SLDO 9
241	361	F1	SLDO 10
242	362	F2	SLDO 11
243	363	F3	SLDO 12
244	364	F4	SLDO 13
245	365	F5	SLDO 14
246	366	F6	SLDO 15
247	367	F7	SLDO 16
248	370	F8	SIND 0
249	371	F9	SIND 1
250	372	FA	SIND 2
251	373	FB	SIND 3
252	374	FC	SIND 4

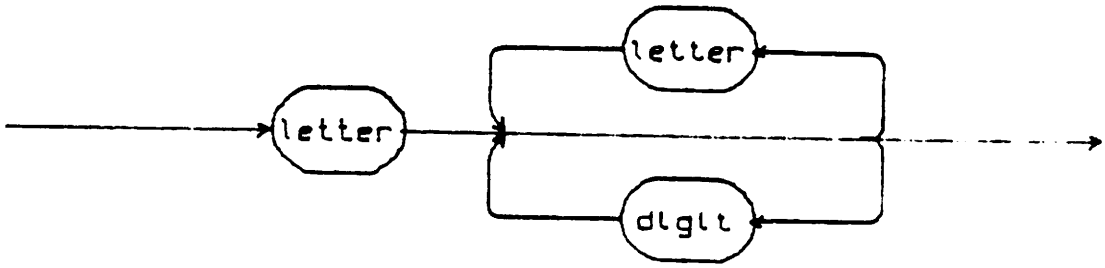
```
167 247 A7                210 322 D2                253 375 FD  SIND 5
168 250 A8  MOV           211 323 D3  EFJ           254 376 FE  SIND 6
169 251 A9  LDO           212 324 D4  NFJ           255 377 FF  SIND 7
170 252 AA  SAS           213 325 D5  BPT
        170 252 AA  SAS           213 325 D5  BPT
```

****
* Section 5.9 *
****

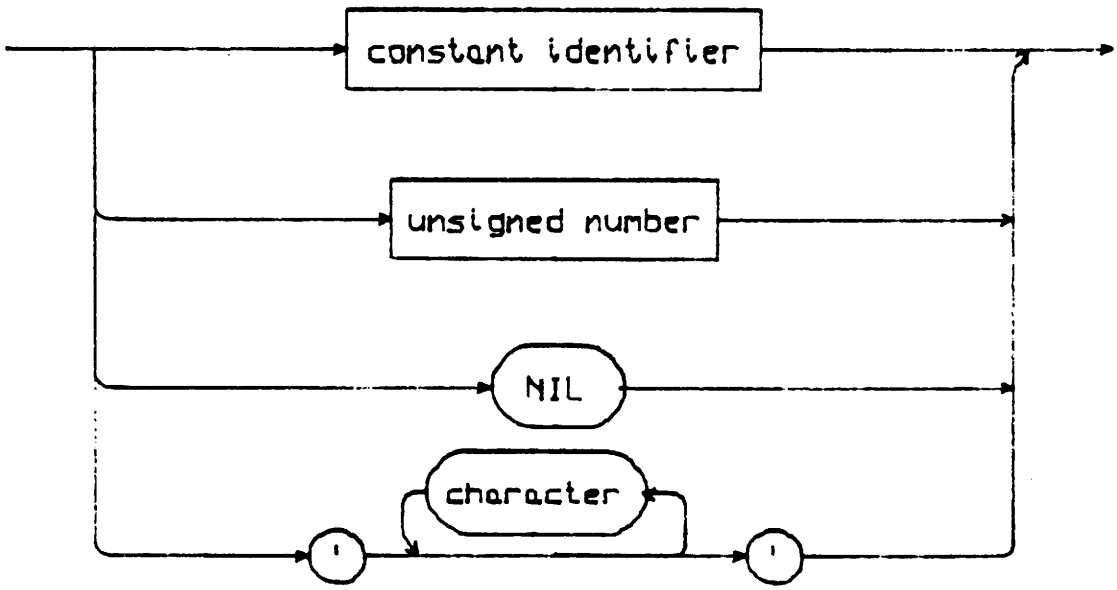
<unsigned integer>



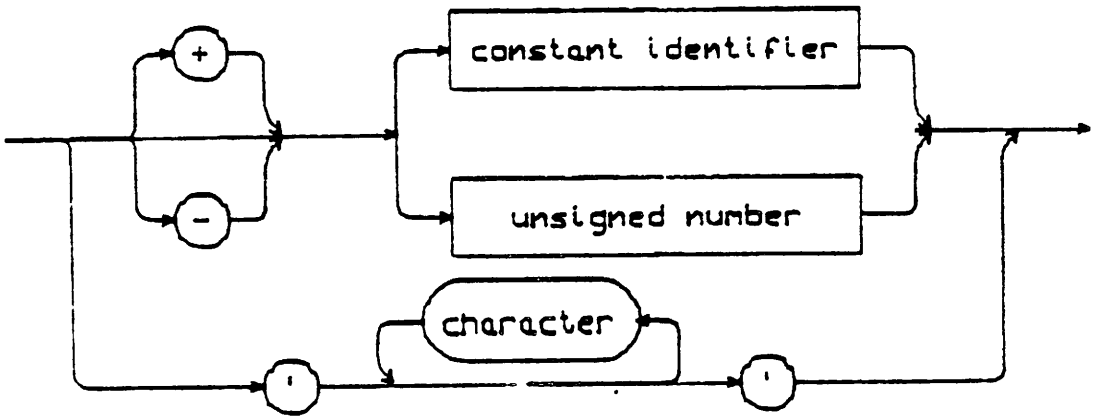
<identifier>



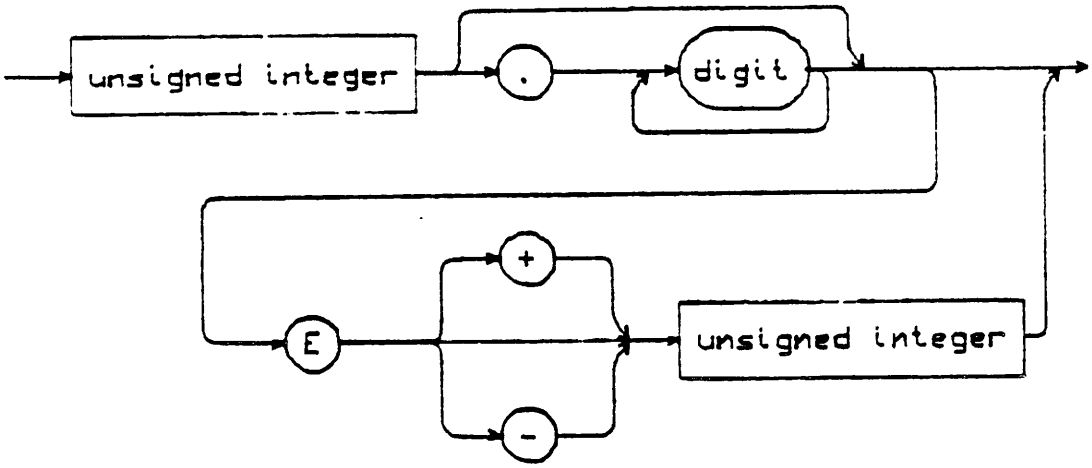
<unsigned constant>



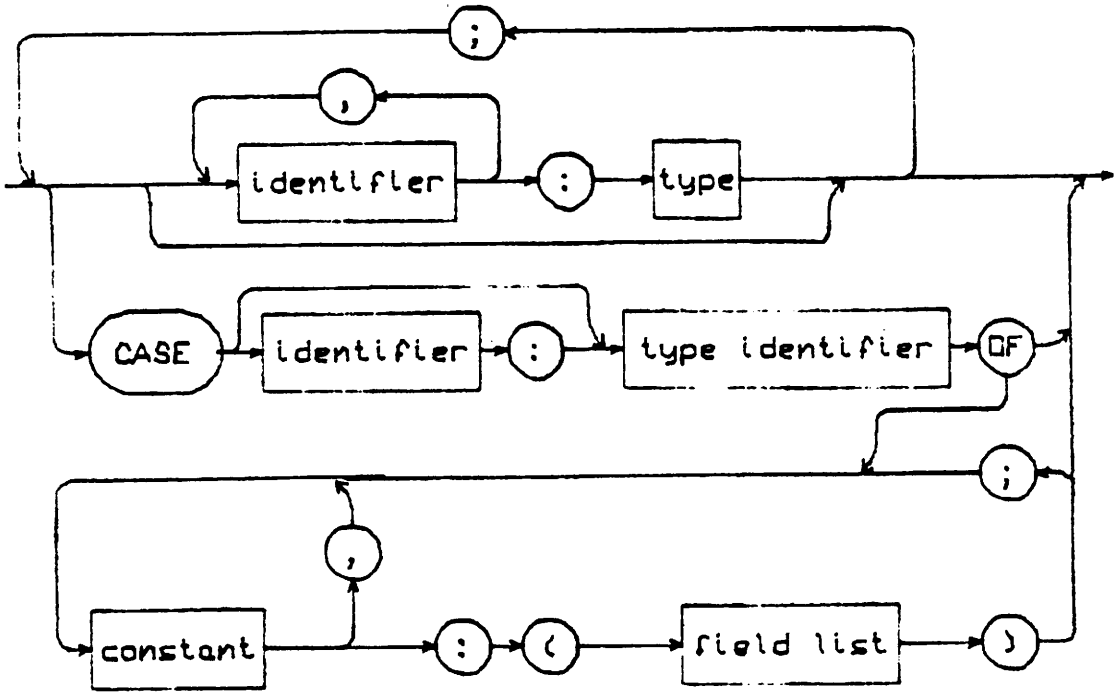
<constant>



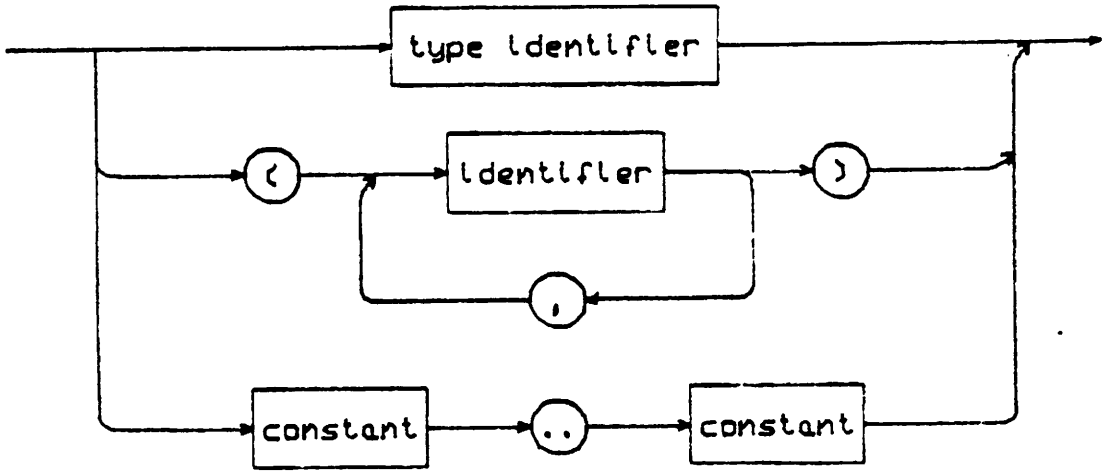
<unsigned number>



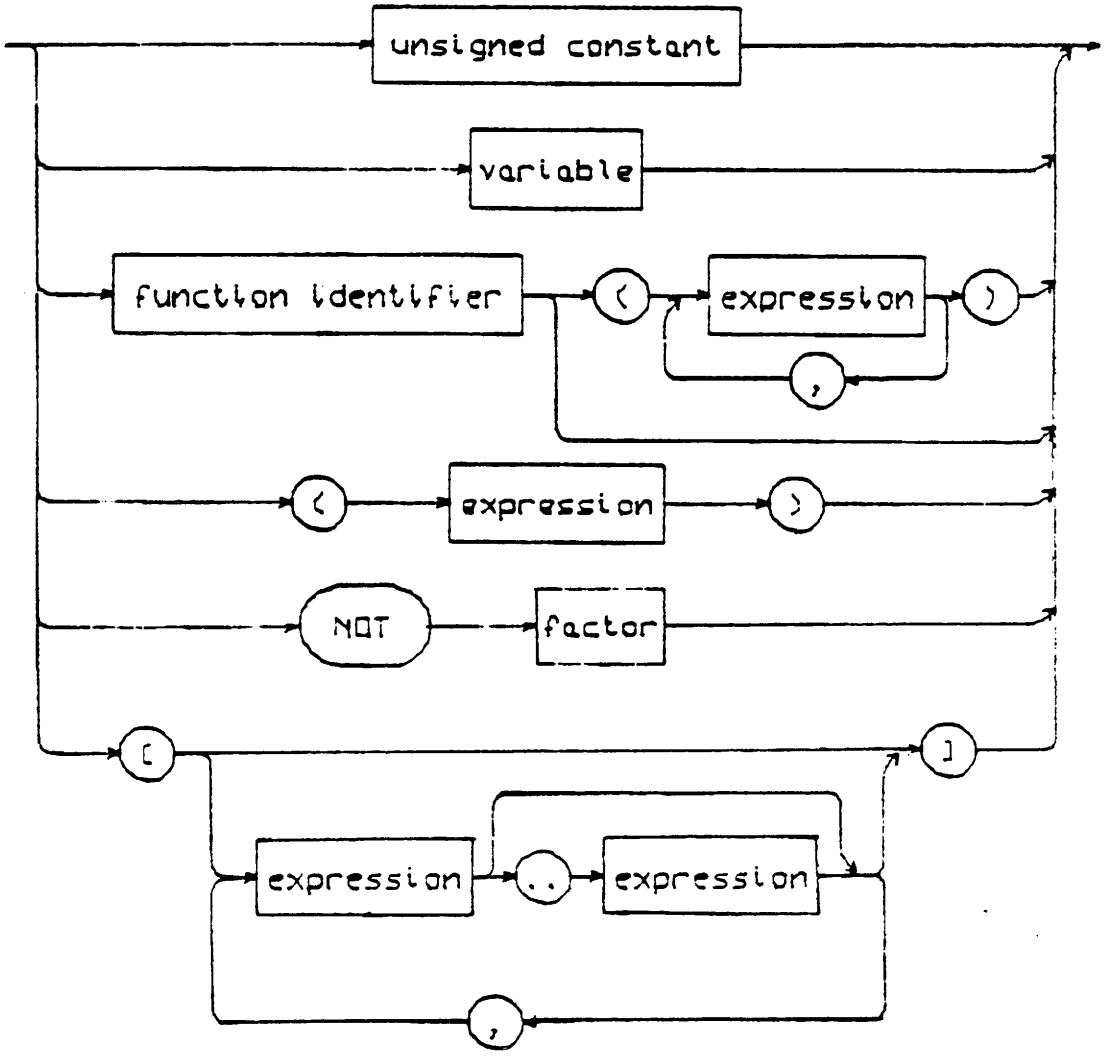
<field list>



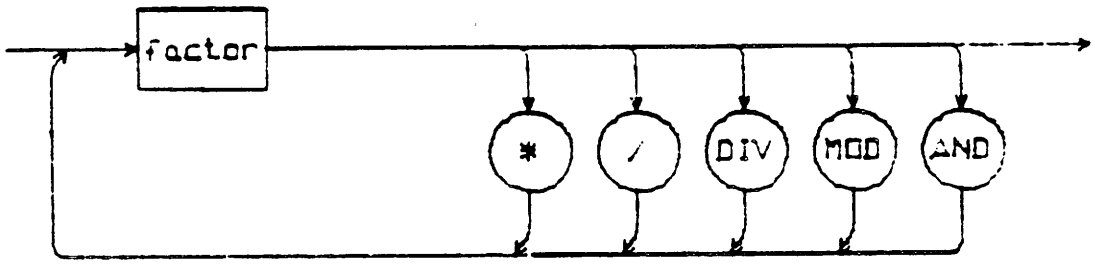
<simple type>



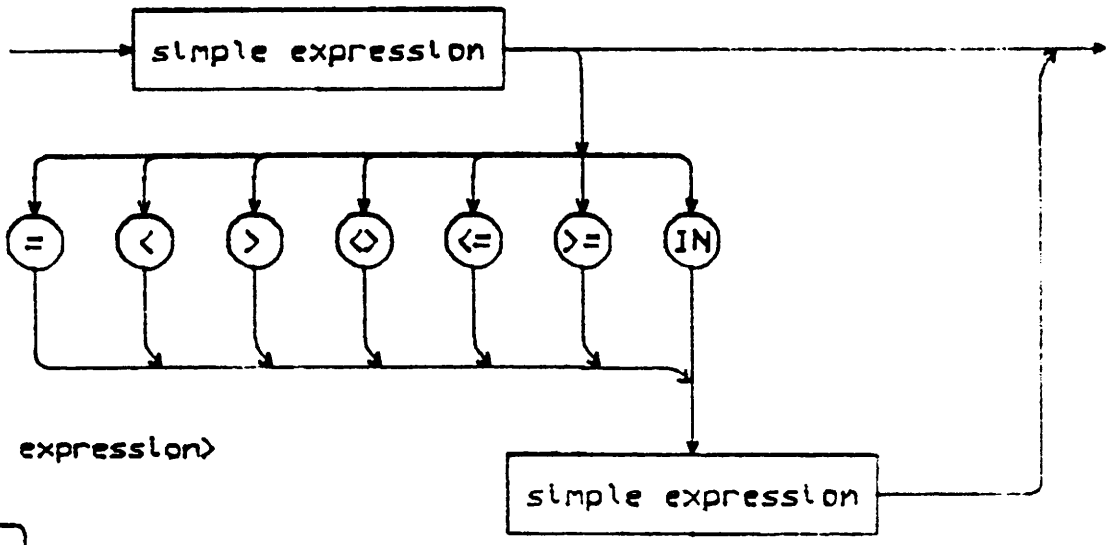
<factor>


unsigned constant
variable
function identifier
(
expression
)
,
(
expression
)
NOT
factor
[
]
expression
..
expression
,

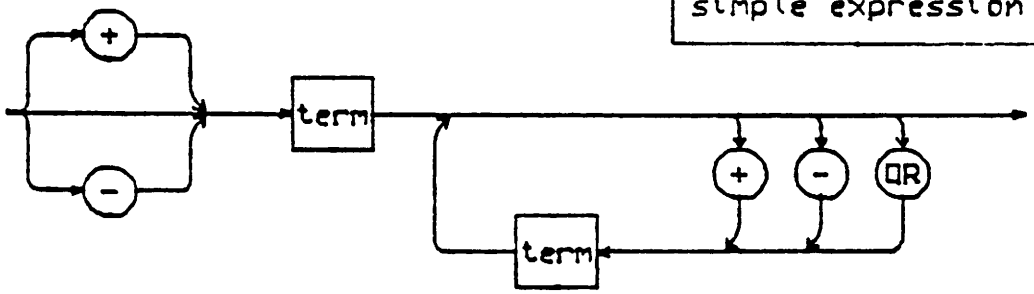
<term>


factor
*
/
DIV
MOD
AND

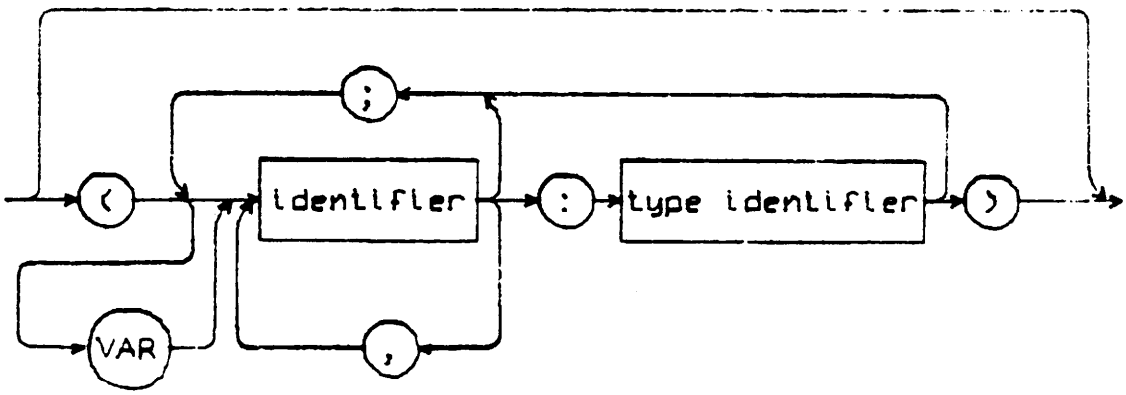
<expression>

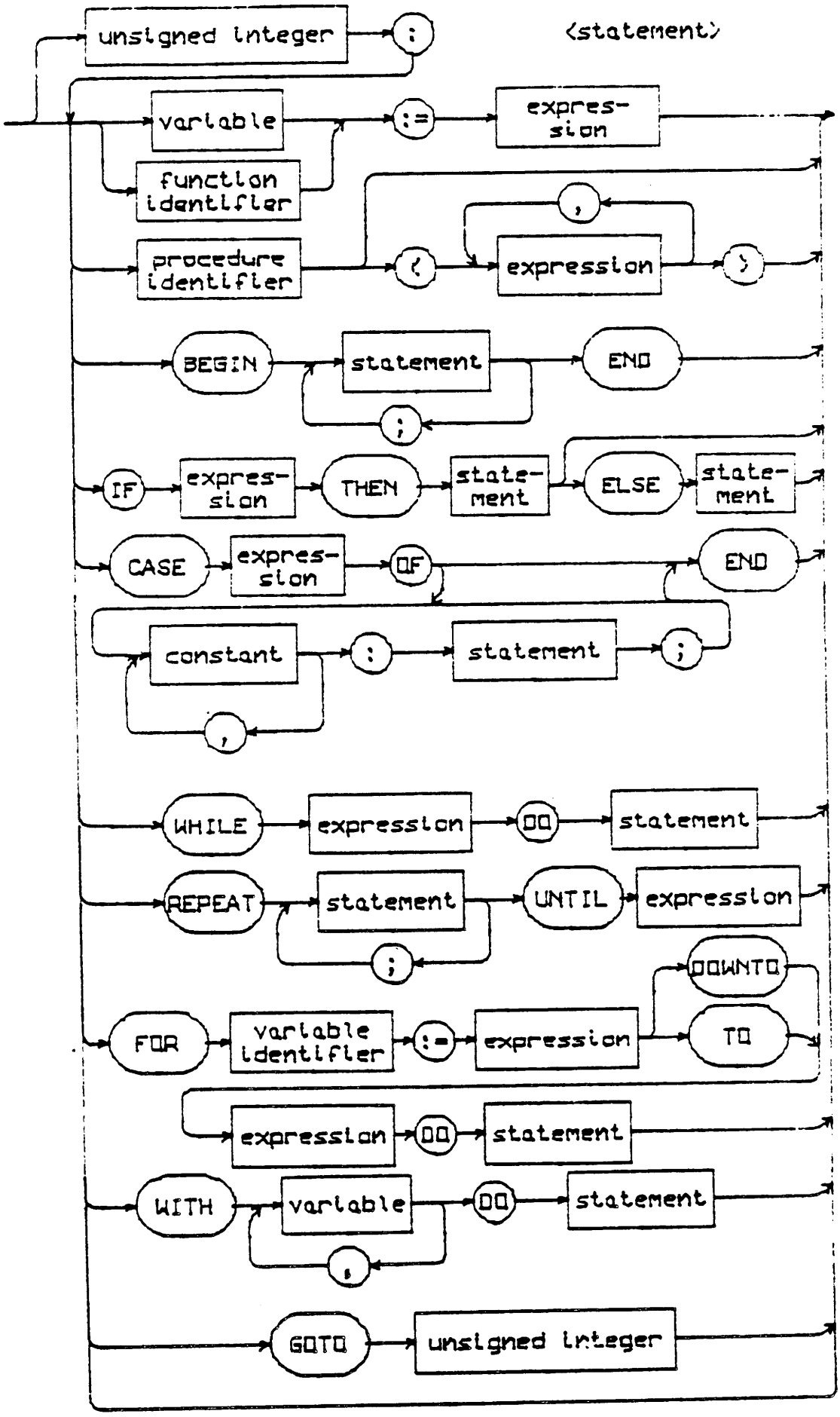


<simple expression>

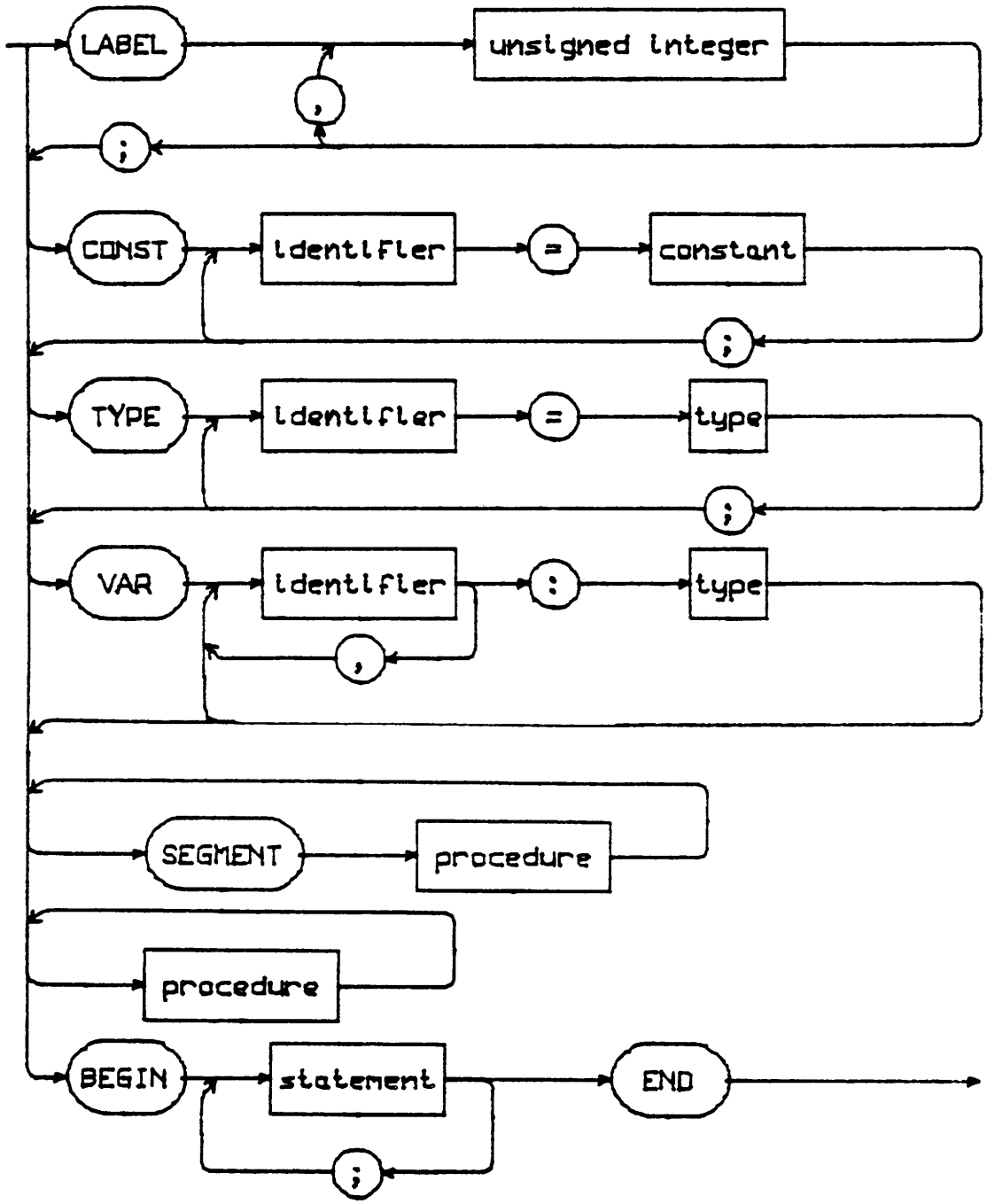


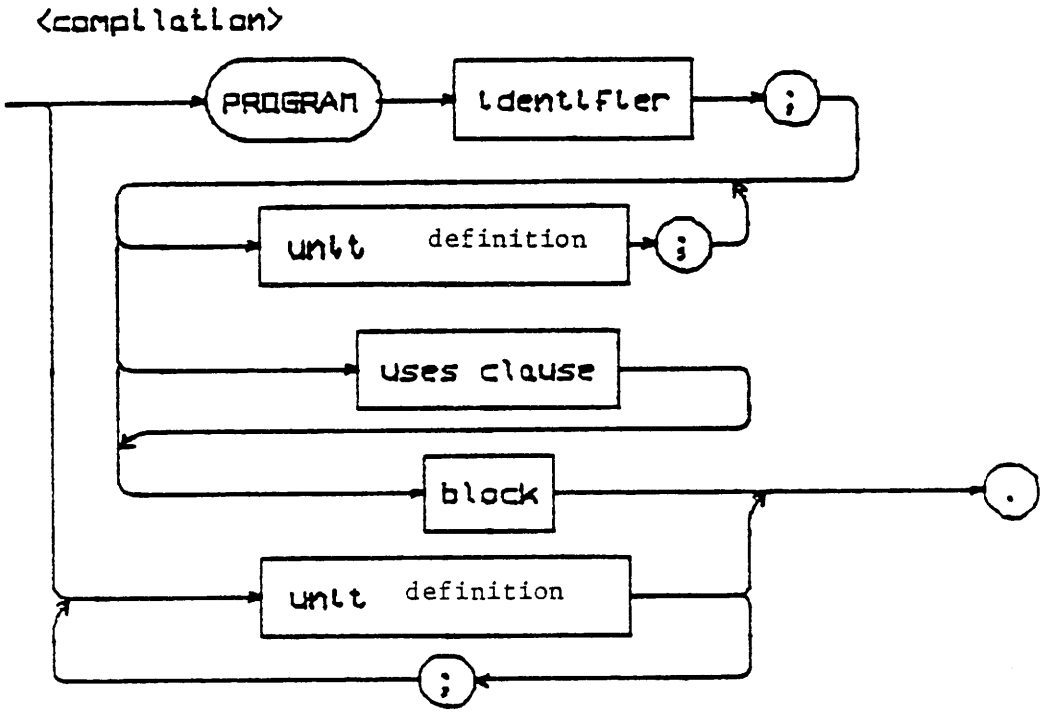
<parameter list>

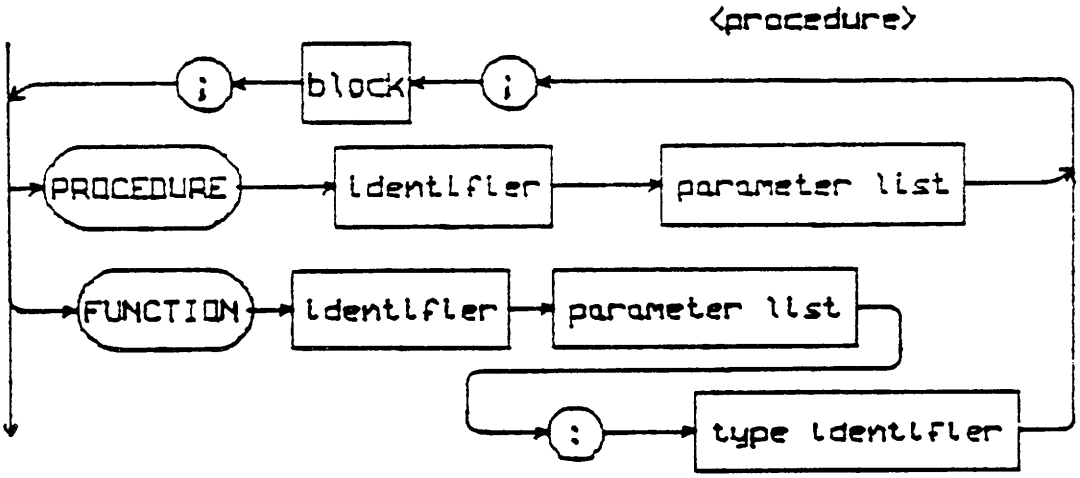



<statement>
unsigned integer
:
variable
:=
expres-
sion
function
identifier
procedure
identifier
(
,
expression
)
BEGIN
statement
END
;
IF
expres-
sion
THEN
state-
ment
ELSE
state-
ment
CASE
expres-
sion
OF
END
constant
:
statement
;
,
WHILE
expression
DO
statement
REPEAT
statement
UNTIL
expression
;
DOWNTO
FOR
variable
identifier
:=
expression
TO
expression
DO
statement
WITH
variable
DO
statement
,
GOTO
unsigned integer

<block>


LABEL
unsigned integer
,
;
CONST
identifier
=
constant
;
TYPE
identifier
=
type
;
VAR
identifier
:
type
,
SEGMENT
procedure
procedure
BEGIN
statement
END
;


<compilation>
PROGRAM
identifier
;
unit definition
;
uses clause
block
.
unit definition
;


<procedure>
;
block
;
PROCEDURE
identifier
parameter list
FUNCTION
identifier
parameter list
:
type identifier

- Notes -

- Notes -

* If you look up these items, you may also want to refer to the added sections which are bound into this manual immediately following this index.

SOFTECH MICROSYSTEMS

A SUBSIDIARY OF SOFTECH

UCSD PASCAL™

BOOTING UNDER CP/M®

UCSD Pascal is a trademark of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only, and any unauthorized use is contrary to the laws of the State of California.

CP/M is a registered trademark of the Digital Research Corporation.

Copyright©1980 by SofTech Microsystems, Inc.
All rights reserved. No part of this work may be reproduced in any form or by any means or used to make a derivative work (such as a translation, transformation, or adaptation) without the permission in writing of SofTech Microsystems, Inc.

9494 Black Mountain Rd., San Diego, CA 92126 (714) 578-6105 TWX: 910-335-1594

TABLE OF CONTENTS

I
BOOTING UNDER THE CP/M OPERATING SYSTEM

I
BOOTING UNDER THE CP/M OPERATING SYSTEM

1. Assessing the Situation

The three critical resources involved in bootstrapping UCSD PASCAL are RAM memory, floppy disk storage and I/O drivers.

1.1 Memory Configurations

It is possible to bootstrap UCSD PASCAL with 48K bytes of memory devoted exclusively to UCSD PASCAL.

1.2 Floppy Disk Requirements

It is necessary that the configuration of any machine that runs UCSD PASCAL have at least 175K bytes (350 PASCAL blocks) of floppy disk storage. This requirement arises from the fact that while it is possible to bootstrap with less space, it is virtually impossible to do anything of interest.

The UCSD PASCAL system is designed to work on any type of floppy medium. This includes mini-floppies, soft sectored floppies, hard sectored floppies, double density floppies, and double sided, double density floppies. The UCSD PASCAL Adaptable System disks are distributed on IBM 3740 soft sectored disks. If the target configuration does not include floppy drives capable of reading the bootstrap disk, a copy of the bootstrapping disk must be created on a disk (called the "target medium") that the available floppy drives can read. The UCSD PASCAL system will then bootstrap from that disk.

1.2.1 Format of the CP/M Adaptable System Disks

The UCSD PASCAL CP/M Adaptable System disks are logically divided into three UCSD PASCAL disk images of 25 tracks apiece. The first disk image is accessible to the UCSD PASCAL system upon initial bootstrapping. The second and third disk images are considered "packed" and must be "unpacked" before access is possible.

A UCSD PASCAL disk image has 25 tracks, logically numbered 0 through 24. Each track contains 26 sectors, numbered 1 through 26, with 128 bytes per sector.

Logical track 0 is reserved for the UCSD PASCAL bootstrap. Sectors 3 through 18 contain overlays (the Secondary bootstrap) called as bootstrapping progresses. Sectors 1, 2, and 19 through 26 are not used.

Logical track 1, sectors 9 through 26 are occupied by the UCSD PASCAL directory.

Logical tracks 2 through 24 are used by the UCSD PASCAL system for file storage.

1.2.2 Creating a UCSD PASCAL Disk on Another Medium

If the target medium is not an IBM 3740 floppy, it is necessary to transfer the Adaptable system floppies to the target medium. There are two commonly used methods: 1) find another computer that can support floppy drives that read the IBM 3740 disk and floppy drives that write the target floppy, or 2) find a computer that can read an IBM 3740 disk and transmit the contents over a serial line. The target computer must be capable of receiving data on the serial line and recording the data on the target disk. Any other method is acceptable as long as the end result is a UCSD PASCAL disk image on the target medium.

There is some data on the UCSD PASCAL disk image that must be transferred to equivalent locations on the target disk. The first 900 hex bytes (2304 bytes or 18 - 128 byte sectors) of track 0 must be transferred from track 0 of the UCSD PASCAL disk image disk to track 0 of the target disk. The bytes starting on the first sector of track 1 must be transferred to the target disk starting on the first sector of its track 1. All remaining data is transferred until the end of the UCSD PASCAL disk image is reached.

Note that for the purposes of initial bootstrapping, the information on a UCSD PASCAL disk image is recorded in contiguous sectors (ie: no sector interleaving is performed).

1.2.3 Disks Provided by SofTech Microsystems

The UCSD PASCAL CP/M Adaptable System release contains four disks. These disks are as follows:

CPMDISK - This disk is CP/M readable and contains the PASBOOT and SAMBOOT programs.

CPMADAP - This disk is used in booting to UCSD PASCAL. It contains three disk images starting at tracks 0, 25, and 50. The first disk image (tracks 0 through 24) contains a UCSD PASCAL Bootstrapping disk that boots under CP/M. The second disk image (tracks 25 through 49) contains a UCSD PASCAL Bootstrapping disk that boots under the SBOOT8 bootstrap (see the Adaptable System manual for more detail). The third disk image (tracks 50 through 74) contains the UCSD PASCAL Interpreter module files and the CPMBOOT program.

SYSTEM - This disk contains three disk images starting at tracks 0, 25, and 50. The first disk image (tracks 0 through 24) contains several system files, the SETUP utility, the duplicate directory utilities, and a information file that will be display when this disk is booted. The second disk image (tracks 25 through 49) contains the PASCAL compiler and the 8080 Assembler. The third disk image (tracks 50 through 74) contains several utilites including the BINDER, the linker, and the Z80 Assembler.

UTIL - This disk contains two disk images starting at tracks 0 and 25. The first disk image (tracks 0 through 24) contains the disassembler, the patch utility, and the screen tester. The second disk image (tracks 25 through 49) contains the IIs operating system (see the Adaptable System manual Appendix F for details).

1.3 I/O Drivers

The UCSD PASCAL CP/M Adaptable System assumes that a standard CBIOS as defined for CP/M version 1.4 is available for use by the UCSD PASCAL system. Only vectors defined within the CBIOS jump table are used. They are expected to be in exactly the same order as is defined for the 1.4 CBIOS. Also no RAM memory below the base of the CBIOS jump table should be used by the CBIOS as the UCSD PASCAL system uses all of this memory. Except for interrupt vectors in memory locations 0 through 80 hex, the CBIOS should be wholly contained within itself.

2. Bootstrapping the UCSD PASCAL System

A CP/M compatible diskette is included in the CP/M release. It contains the source and code of the PASBOOT program. (Note: if the target medium is not IBM 3740 disks, separate arrangements must be made to download the contents of this disk to the CP/M system on the target medium.) To bootstrap the UCSD PASCAL system perform the following steps:

1) Boot a standard CP/M version 1.4 (or a proper superset) operating system on the target machine.

2) Insert the CP/M compatible diskette containing the PASBOOT program into drive A.

3) Type 'PASBOOT' followed by a <carriage return>. The PASBOOT program will execute and print the following message:

```
UCSD PASCAL (II.0) BOOTER VERSION [xx]
INSERT PASCAL DISK INTO DRIVE A, THEN TYPE <RETURN>
```

   The UCSD PASCAL CP/M Adaptable System disk should be used as the system disk. Insert the system disk into drive A then hit the <carriage return> key.

The PASBOOT program will then begin reading the Secondary Bootstrap, and will print the following message:

```
READING SECONDARY BOOTSTRAP
```

If the Secondary Bootstrap is read correctly, the program will print the following message and enter the Secondary Bootstrap:

```
BOOTING TO UCSD PASCAL
```

The Secondary Bootstrap will then read the PASCAL directory and search for the UCSD PASCAL Interpreter file on the disk. If this fails it will print the following message and halt:

```
Can't find SYSTEM.INTERP
```

The Secondary Bootstrap will then read the interpreter and enter the Tertiary Bootstrap. The Tertiary Bootstrap will once again read the PASCAL directory, this time searching for the operating system file. If it cannot be found the following message is printed and the system will halt:

```
Can't find SYSTEM.PASCAL
```

The Tertiary Bootstrap will then initialize the operating system and the interpreter. The operating system prompt line should be printed soon thereafter.

This entire bootstraping process should take less than two minutes. A one disk drive UCSD PASCAL System is initially booted.

3. Checking the UCSD PASCAL System

At this point, the UCSD PASCAL system should have bootstrapped. There are a few simple tests to perform that check the interaction between the CBIOS and the UCSD PASCAL system.

The first is observation of the output on the console. There should be a welcoming message followed by the system version number and the date on which the UCSD PASCAL Bootstrap disk was created. Following that, the standard system prompt line should appear (refer to the UCSD PASCAL user's manual for the exact form of this). If these outputs do not appear, almost anything could be wrong. Check the values passed to the UCSD PASCAL bootstrap on the stack by the PASBOOT program. In addition, either the disk read routines or the console output routines may be nonfunctional.

The next test is to hit the 'F' key on the console. this should invoke the file manager. The system floppy drive should read several sectors. Another prompt line should appear. If these actions do not occur, the disk read routines or the console input/output routines may not work.

The final test is to hit the 'D' key on the console. Next, type the current date (eg. 12-JAN-79) followed by the <return> key. Finally, type 'D' again. If the correct date is not displayed, the disk write routines may be at fault.

3.1 Devices Available

The CP/M version of the UCSD PASCAL system can communicate with all standard CP/M devices. In addition to the console and disk drives, the printer is available using UCSD PASCAL unit 6 (PRINTER:); the tape reader is available using UCSD PASCAL unit 7 (REMIN:); and the tape punch is available using UCSD PASCAL unit 8 (REMOUT:).

The UCSD PASCAL system has a limited form of keyboard queuing. A character typed when the PASCAL system is not reading from the keyboard will be queued for subsequent processing if there is some other I/O (eg. disk reads/writes, console writes) in progress. The system supports the STOP/START, FLUSH, and BREAK functions in this manner (See the UCSD PASCAL User's manual for further details.)

3.2 Two Drive Systems

It is possible to install a UCSD PASCAL interpreter that communicates with two floppy drives by entering the F)iler and using the C)hange command to change the files SYSTEM.INTERP to CPM1.INTERP and CPM2.INTERP to SYSTEM.INTERP. This operation saves the one disk version of the UCSD PASCAL interpreter in the file CPM1.INTERP. The two disk version of the UCSD PASCAL interpreter may then be booted in a like manner to the one disk version. The only difference is that a diskette must be in both drives in order to prevent the system from hanging on a read to the second drive.

3.3 Utility Programs on the Bootstrapping Disk

A listing of the directory of the UCSD PASCAL Bootstrapping disk can be obtained by entering the F)iler and typing 'L' followed by ':' and a <carriage return>. The following files are provided:

FILE NAME	DESCRIPTION
SYSTEM.INTERP	UCSD PASCAL Interpreter 1 disk drive
SYSTEM.PASCAL	UCSD PASCAL Operating System
SYSTEM.FILER	UCSD PASCAL File Handler
SYSTEM.MISCINFO	Terminal description file
SYSTEM.LIBRARY	Long Integer and Real Number library
FINDPARAMS.CODE	Program to determine optimal disk recording format
DISKCHANGE.CODE	Program to reformat a UCSD PASCAL disk
DISKSIZE.CODE	Program to change the size of a UCSD PASCAL disk
BOOTER.CODE	Program to transfer bootstraps from disk to disk
CPM2.INTERP	UCSD PASCAL Interpreter 2 disk drives

3.4 Disk number mapping

The UCSD PASCAL system accesses floppy disk drives by number. The mapping between CBIOS unit numbers (as passed to the SETDISK routine) and UCSD PASCAL unit numbers is:

CBIOS UNIT	UCSD PASCAL UNIT
0	4
1	5
2	9
3	10
4	11
5	12

When referring to a floppy disk while under control of the UCSD PASCAL system, the UCSD PASCAL unit number should be used.

3.5 Preparing Release Disks for Use

The DISKCHANGE program can be used to extract a given floppy image. This is done by X)ecuting the DISKCHANGE program. It will ask for the disk drive numbers (4 = CBIOS disk 0, 5 = CBIOS disk 1, 9 = CBIOS disk 2, etc.) involved in the transfer. A transfer interleaving factor is also requested. This may be small (eg. 2) for a fast floppy drive or large (eg. 7) for a slower drive. This factor is relevent to the speed of the transfer, not to the format of the data.

Information is requested about the source recording format then about the destination recording format. The source interleaving and skew factors must be 1 and 0, respectively. The first interleaved track will be 0, 25, or 50, respectively, for each of the disk images. The destination interleaving and skew factors will be 1 and 0, respectively, and the first interleaved track will be 0.

To unpack an entire release disk, X)ecute the DISKCHANGE program to transfer the first floppy image to a floppy. Repeat the process, transferring the second and third floppy images to a second and third floppy. The release disk should be kept as a backup.

3.6 Accessing the UCSD PASCAL System Programs

The sole purpose of UCSD PASCAL Bootstrapping disk is to aid in the developement of bootstraps. There is no need for most UCSD PASCAL system programs in this process. Hence, they are provided on the UCSD PASCAL System disk rather than on the UCSD PASCAL Bootstrapping disk.

The System disk contains three disk images and may be unpacked as described above. The first disk image, SYSTEM1, contains a UCSD PASCAL system that may be booted once a bootstrap and a UCSD PASCAL interpreter are transferred to it. The bootstrap is transferred by X)ecuting the BOOTER program on the UCSD PASCAL Bootstrapping disk to copy the bootstrap on the UCSD PASCAL Bootstrapping disk to the SYSTEM1 disk. The UCSD PASCAL interpreter may be transferred by using the F)iler's T)ransfer command to transfer the SYSTEM.INTERP file on the UCSD PASCAL Bootstrapping disk to the SYSTEM1 disk. The SYSTEM1 disk may be booted in the same manner as the UCSD PASCAL Bootstrapping disk. Other system programs are recorded on the second and third disk images of the UCSD PASCAL System disk. It may be the case that there is not enough room on the SYSTEM1 disk to install the Interpreter. If this is the case a disk can be made, again using the T(ransfer command, that contains: SYSTEM.PASCAL, SYSTEM.INTERP, SYSTEM.MISCINFO, SYSTEM.STARTUP, SETUP.CODE and BINDER.CODE (as a bare minimum). This disk can have the bootstrap written to it, as described above, and then booted.

Once the SYSTEM1 disk is booted, a systems program may be executed by typing the associated letter when the system prompt line appears (eg. "E" for editor, "C" for compiler). Refer to the UCSD PASCAL User's Manual for full details. A system program invoked in this manner must reside on some floppy that is on line, not necessarily the SYSTEM1 floppy.

The SYSTEM.PASCAL file and the SYSTEM.INTERP file MUST be on any disk that is booted directly. The SYSTEM.SYNTAX, SYSTEM.MISCINFO, and SYSTEM.LIBRARY files must also be on the booted disk if the UCSD PASCAL system is to make use of them.

Note that in order to use an assembler, the assembler must be named SYSTEM.ASSMBLER. It may be necessary to enter the F)iler and use the C)hange command to change the name of an assembler not so named. Any assembler information files (eg. Z80.OPCODES) must reside on the same disk as the assembler code file.

3.7 Backing Up the Bootstrapping Disk

The UCSD PASCAL Bootstrapping disk and the SYSTEM1 disk should be backed up at this point. To do so, enter the F)iler and do a volume to volume T)ransfer. (See the UCSD PASCAL User's Manual for details). This will copy the UCSD PASCAL system-areas to a backup disk. Use the BOOTER program on the UCSD PASCAL Bootstrapping disk to copy the bootstrap to the backup disk.

3.8 Customizing a UCSD PASCAL Disk Image

All UCSD PASCAL disk images are configured to contain 153 PASCAL blocks (512 bytes per block). Thus, disk space is wasted on floppies that can hold more than 153 PASCAL blocks. The DISKSIZE program is provided to alter the configuration of a UCSD PASCAL disk image so it utilizes an entire floppy disk. When X)ecuted, it asks for the drive number (4 = CBIOS disk 0, 5 = CBIOS disk 1, 9 = CBIOS disk 2, etc) that contains the disk to be reconfigured. It then asks for the number of blocks the disk may hold. This can be calculated by the following formula:

```
(# tracks per disk - first PASCAL track)
* (# of sectors per track) / (512 / # of bytes per sector)
```

4. Improvements

Once the UCSD PASCAL system has been bootstrapped, it is possible to speed up disk accesses and to provide an automatic bootstrap. Disk access speed may be improved by choosing a disk recording format suited to the floppy drive being used. An automatic bootstrap may be written that executes without the presence of the CP/M operating system.

4.1 The PASBOOT Program

The PASBOOT program is an assembly language program that runs under the CP/M operating system and bootstraps the UCSD PASCAL system. PASBOOT reads the UCSD PASCAL Secondary bootstrap from track 0, sectors 3 through 18 of the UCSD PASCAL Bootstrap disk into memory starting at 8200 hex. It also puts parameters describing the target machine configuration onto the processor stack then jumps to 8200 hex.

Several equates in the source may be modified to reflect the CP/M operating system environment or the UCSD PASCAL system environment. These are:

DDT - This flag is normally set to FALSE, but when set to TRUE allows the UCSD PASCAL system to be followed and debugged using DDT.

BOOT - This is the address of the JMP WBOOT for the CP/M operating system. It is used to set the top of available RAM memory. (The DDT flag must be set to FALSE.) The default is the standard location 0000H.

BDOS - This is the address of the JMP BDOS for the CP/M operating system. It is used to set the top of available RAM memory, when debugging with DDT in memory. (The DDT flag must be set to TRUE.) The default is the standard location 0005H.

TPA - This is the address of the start of a user program when assembled under the CP/M operating system. The default is the standard address 0100H.

INTERP$BASE - This is the starting address for the UCSD PASCAL Interpreter and is normally set to be equal to TPA. It must, however, be on a page boundary, i.e., XX00H. The default is 0100H.

LOW$MEMORY - This is the lowest available RAM address. It must be the base of a contiguous block of at least 48K of RAM, must be greater than or equal to INTERP$BASE and must

be on a page boundary. The default is 0100H.

TRACKS - This is the number of tracks on the booting disk. The default is for IBM standard 8" disks, which is 77.

SECTORS - This is the number of sectors on the booting disk. The default is for IBM standard 8" disks, which is 26.

BYTES - This is the number of bytes per sector on the booting disk. The default is for IBM standard 8" disks, which is 128.

INTERLEAVE - This is the disk interleaving ratio, given as INTERLEAVE:1. It is determined by the FINDPARAMS program. The default is for the UCSD PASCAL CP/M Adaptable disks, which is 1:1, i.e., un-interleaved.

FIRST$TRACK - This is the track on which UCSD PASCAL block 0 starts. It is usually set to one track above any bootstrap-containing tracks. The default is for the UCSD PASCAL CP/M Adaptable disks, which is 1.

SKEW - This is the track-to-track offset to the next sector. It is determined by the FINDPARAMS program. The default is for the UCSD PASCAL CP/M Adaptable disks, which is 0.

MAX$SECTORS - This is the maximum number of sectors on any of the disks to be on-line, e.g., if a single-density and a double-density drive are to be on-line, and the single drive has 26 sectors per track and the double has 52 sectors per track, MAX$SECTORS must be set to 52. The default is the same as SECTORS.

4.2 Disks Not On Line

The definition of the CP/M BIOS disk handler requires that whenever an unloaded floppy drive is accessed, the disk handler emits error messages until the drive is loaded. The UCSD PASCAL system will run without all floppy drives loaded and online if the following changes are made. The disk routines should return an error result of 9 in the A register when an unloaded floppy drive is accessed. An error result of 1 in the A register should be returned on any other floppy error. Successful floppy operations should return a 0 in the A register. Finally, no error messages should be printed under any circumstances.

4.3 Speeding Up the UCSD PASCAL System

All UCSD PASCAL disks distributed as part of the CP/M Adaptable System package are recorded with 1 to 1 interleaving and no skew factors. (The sector interleaving factor is a function of the ratio of the processor speed to the floppy disk revolution speed. The track-to-track skew is a function of the ratio of the floppy disk revolution speed to the floppy drive seek time. The particular recording scheme chosen for the UCSD PASCAL Bootstrapping disk makes no assumptions about either of these ratios.) It may be possible to significantly decrease the average disk access time by using recording parameters better suited to the target configuration.

The recording format of the UCSD PASCAL Bootstrapping disk may be changed as follows: (Note: it is advisable that a backup copy be made of any floppy undergoing this procedure before continuing. Refer to the section on the T)ransfer command of the F)iler in the UCSD PASCAL User's Manual for instructions on disk-to-disk transfers.)

1) eX)ecute the FINDPARAMS program found on the UCSD PASCAL Bootstrapping disk. It will aid in determining the optimal interleaving and skew factors.

2) eX)ecute the DISKCHANGE program found on the UCSD PASCAL Bootstrapping disk. It reformats the UCSD PASCAL Bootstrapping disk according to new sector interleaving, track-to-track skew, and first PASCAL track parameters. (For this exercise, the first PASCAL track should be 1).

3) Change the sector interleaving and the track-to-track skew parameters passed on the stack in PASBOOT to match the new disk format and reassemble PASBOOT with the CP/M Assembler.

4) Reboot the UCSD PASCAL Bootstrapping disk.

Note: all floppies intended to be read by the UCSD PASCAL system must have the same disk recording parameters. Thus, if the format of the UCSD PASCAL Bootstrapping disk is changed, the formats of all other floppies intended for use with the UCSD PASCAL system must also be changed. Use the DISKCHANGE program. Be warned, however, the DISKCHANGE program destroys all disk images recorded on tracks after the end of the desired floppy image. If the preservation of other disk images is important, extract them as described above BEFORE the DISKCHANGE program is executed.

There are many soft sectored 8" UCSD PASCAL floppies in the field whose disk format is 2 to 1 interleaving, 6 sector skew, and first PASCAL track of 1. This format is suggested for 8" soft sectored floppies if compatibility with other users' floppies is important. However, the DISKCHANGE program can be used to

convert any UCSD PASCAL disk to a compatible format.

4.4 Creating an Automatic Bootstrap

It is possible to create a UCSD PASCAL system disk that boots without the aid of the CP/M operating system. The target configuration must include a facility that reads a primary bootstrap recorded on track 0, sector 1 of the UCSD PASCAL system disk into memory starting at a predetermined location when the boot-button is pressed. A new primary bootstrap must be written (under the CP/M operating system) that will read a secondary bootstrap and a CBIOS into memory. A UCSD PASCAL program is run to transfer a copy of the primary bootstrap and a copy of the CBIOS onto track 0 of the UCSD PASCAL Bootstrapping disk. The UCSD PASCAL Bootstrapping disk is then booted in the same way as the CP/M operating system disk.

4.4.1 Writing the Primary Bootstrap

The primary bootstrap is similar to the PASBOOT program with the exception that the CBIOS is not assumed to be available in memory, but must be read from the UCSD PASCAL disk. The bootstrap must:

1) Read the UCSD PASCAL Secondary bootstrap from track 0, sectors 3 through 18 into memory starting at 8200 hex.

2) Read a copy of the CBIOS from track 0, sectors 19 through 26 into memory starting at the location at which the CBIOS is assembled to execute.

3) Push the parameters that describe the target configuration onto the processor stack. An example of this is found in the PASBOOT source file.

The primary bootstrap must be origined to run wherever the boot-button hardware for the target machine reads it.

A sample primary bootstrap program is provided in the SAMBOOT file on the CP/M compatible disk.

4.4.2 Running the CPMBOOT Transfer Program

When the primary bootstrap is debugged, a copy of it and the CBIOS must be transferred to track 0 of the UCSD PASCAL Bootstrapping disk. This is accomplished by bootstrapping the UCSD PASCAL system as described above and X)ecuting the CPMBOOT program (found on the third disk image on the CPMADAP release disk). This program copies the primary bootstrap HEX file from a CP/M disk to track 0, sector 1 of the UCSD PASCAL Bootstrapping disk. It also copies the CBIOS from a CP/M disk to track 0, sectors 19 through 26 of the UCSD PASCAL Bootstrapping disk. The CBIOS can come either from the bootstrap area of the CP/M disk or from a CBIOS.HEX file on the CP/M disk. This results in the creation of a stand-alone UCSD PASCAL system. Note: CPMBOOT assumes the a standard 8" disk is being written to. For disk drives with less than 26 sectors on track 0, an alternative method for creating a stand-alone bootstrap will have to be found. Refer to the Adaptable System manual for more details.

4.5 Changing the UCSD PASCAL Interpreter

The UCSD PASCAL Interpreter (SYSTEM.INTERP) is configured to handle 1 floppy disk drive and polled console type-ahead with stop/start, flush, and break capabilities. It has no floating point number capabilities.

The characteristics of the UCSD PASCAL interpreter may be changed by re-linking the UCSD PASCAL interpreter. Refer to the Adaptable System manual for detailed instructions. The files INTER.CODE, INTER.X.CODE, BIOS.CR.CODE, and BIOS.CRP.CODE should NOT be used. In their place, substitute the files INTER.CPM1.CODE, INTER.CPM2.CODE, and INTER.CPM4.CODE found on the CP/M UCSD PASCAL Bootstrapping disk. These modules interface with CBIOSes that handle 1, 2, and 4 disks, respectively.

4.6 Using the Full Adaptable System

The capabilities offered in the CP/M compatible version of the UCSD PASCAL system are a subset of those found in the UCSD PASCAL Adaptable System. The Adaptable System offers the ability to support more diverse types of disk drives, user defined devices, a system clock, and remote and printer type-ahead.

BOOTING UNDER THE CP/M OPERATING SYSTEM

An image of the UCSD PASCAL Bootstrapping disk compatible with the UCSD PASCAL Adaptable System manual is available on the second floppy image (tracks 25 through 49) of the UCSD PASCAL Adaptable System disk.

SOFTECH MICROSYSTEMS

A SUBSIDIARY OF SOFTECH

UCSD PASCAL™

ADAPTABLE SYSTEM MANUAL


```
************************************************************************
* Copyright 1979©  by SofTech Microsystems, Inc.                       *
*   All rights reserved.  No part of this work may be reproduced in    *
*   any form or by any means or used to make a derivative work (such   *
*   as a translation, transformation or adaptation) without the        *
*   permission in writing of SofTech Microsystems, Inc.                *
*                                                                      *
* UCSD Pascal is a trademark of the Regents of the University of       *
*   California.  Use thereof in conjunction with any goods or          *
*   services is authorized by specific license only, and any           *
*   unauthorized use is contrary to the laws of the state of           *
*   California.                                                        *
************************************************************************
```


9494 Black Mountain Rd., San Diego, CA 92126 (714) 578-6105 TWX: 910-335-1594

TABLE OF CONTENTS

Chapter I
THE CAPABILITIES OF THE UCSD PASCAL ADAPTABLE SYSTEM

Chapter II
ASSESSING THE SITUATION

Chapter III
BOOTSTRAPPING UCSD PASCAL

TABLE OF CONTENTS

Chapter IV
ADDING CAPABILITIES

Appendix A
VECTOR LISTS AND REGISTER ASSIGNMENTS

Appendix B
SAMPLE BOOTSTRAP LOADERS

Appendix C
GENERAL NOTES

Appendix D
RECONFIGURING THE UCSD PASCAL INTERPRETER

Chapter I
THE CAPABILITIES OF THE UCSD PASCAL ADAPTABLE SYSTEM

The UCSD Pascal Adaptable System is intended to assist in the implementation of UCSD Pascal on almost ANY configuration with minimal effort. With this system, it is possible to boot UCSD Pascal on machines that have at least 48K bytes of RAM of which at least 36K are contiguous. The minimum disk storage capacity necessary is 175K bytes (350 Pascal blocks). The actual floppies may be of any type (single density, double density, mini, hard sectored, etc.). The system configuration must also include a teletype or crt display that can both send and receive ASCII characters.

In addition to a UCSD Pascal system, the Adaptable disk contains special bootstrapping and testing utilities that enable UCSD Pascal to bootstrap quickly and reliably. They are set up to execute under a SofTech Microsystems defined standard memory configuration. If the target system is not so configured, an alternate bootstrap is provided that should execute on the target configuration. An I/O module that communicates with the floppy disks and console must be provided for the target configuration.

Once the UCSD Pascal system has been bootstrapped, additional facilities may be provided to communicate with a printer, a remote device, user defined devices, and a system clock. The I/O configuration may be extended to provide access to different types of floppy drives on line at the same time.

A special version of the Adaptable System is distributed for configurations running the CP/M operating system. It uses a standard CP/M BIOS for I/O support and is bootstrapped with much less initial effort than the standard Adaptable System. This version is described in a self-contained section which is included in this volume — you should refer to that section.

The disks that come with the CP/M Adaptable System are described in detail in that section. The other Adaptable Systems come on four disks.

Each disk in the Adaptable System is an 8" soft sectored, single density floppy which contains the virtual images of three UCSD Pascal floppy disks, in uninterleaved format. Each of these virtual disks is small enough to fit on one 5¼" minifloppy. The four 8" disks include a SYSTEM disk, which contains UCSD Pascal system software, one UTIL disk containing some miscellaneous utility programs, and two Adaptable disks, which are the disks used in booting your configuration for the first time.

Each Adaptable disk contains two Bootstrap disks, and one Interpreter disk. Which Bootstrap disk you need to use depends on your memory configuration: see Section II.1.

Of the two Adaptable disks shipped, you will probably need to use only one. If you have a Z-80/8080 system, the ADAP8 applies to 8080 configurations, and the ADAPZ applies to Z-80 configurations. For 6800 or 6502 configurations, the two Adaptable disks are called HI PAGE or LO PAGE — the one you will use depends on your memory configuration: see Section C.2.1 or C.3.1.

---

1. UCSD Pascal is a trademark of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only, and any unauthorized use is contrary to the laws of the State of California.
2. CP/M is a registered trademark of the Digital Research Corporation.

We recommend you read through all of Chapter II before attempting to bootstrap your system. We also highly advise that a few copies of the Adaptable disk be made BEFORE proceeding to work with it, as the types of operations described in this document lend themselves to accidental destruction of floppy disk information.

Chapter II
ASSESSING THE SITUATION

The three critical resources involved in bootstrapping UCSD PASCAL are RAM memory, floppy disk storage, and I/O drivers.

1. Memory Configurations

It is possible to bootstrap UCSD PASCAL with 48K bytes of memory devoted exclusively to UCSD PASCAL. A minimum of 36K contiguous bytes must be available for user data space. There are three software modules that must coexist in RAM. They include the SBIOS, the interpreter, and the bootstrap.

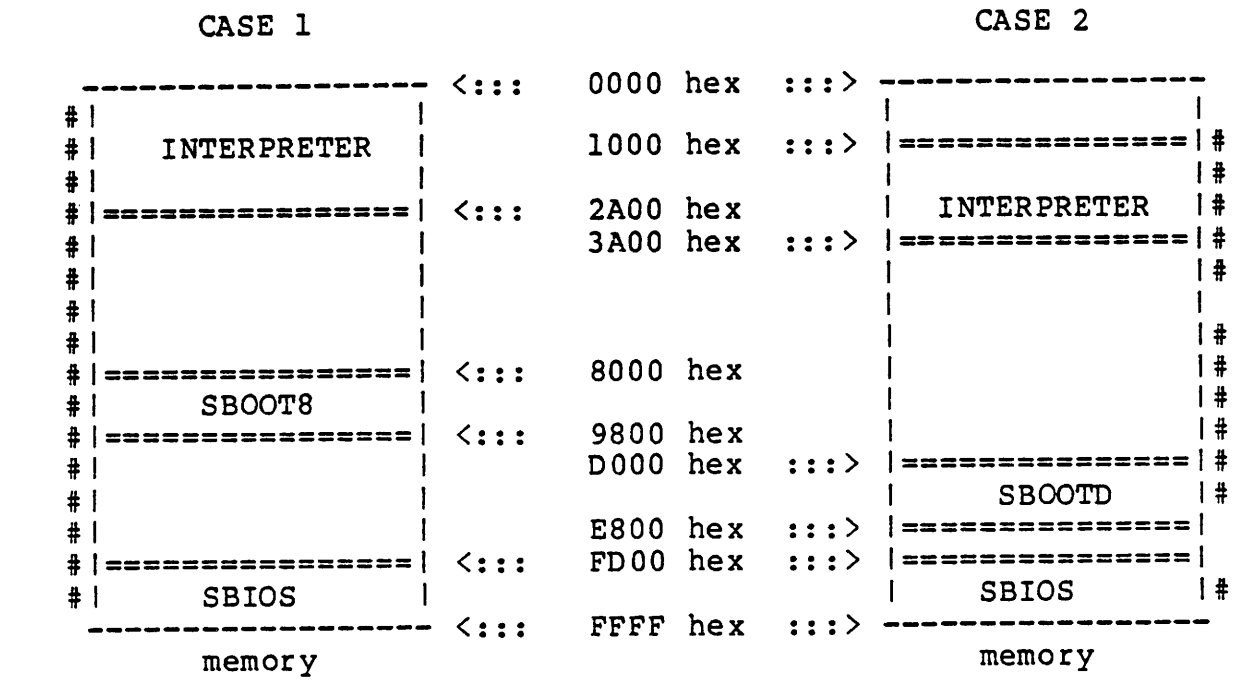
The SBIOS is an I/O module customized to the target configuration (see section II.3). It can be located wherever it fits best in RAM. From the UCSD PASCAL system's point of view, it is best situated in a small, isolated RAM space of its own. If one is not available, the SBIOS may occupy as much of the end of the large contiguous RAM as is necessary.

The bootstrap module is 1800 hex bytes long and executes in the large contiguous RAM space. If the large contiguous memory space starts at or before location 3000 hex, the default bootstrap may be used. It runs at 8000 hex and is called SBOOT8. If the large contiguous RAM starts after 3000 hex, an alternate copy of the bootstrap must be used. It runs at D000 hex and is called SBOOTD. Once it is finished running, the memory space occupied by the bootstrap is returned to the large contiguous memory pool.

Finally, the interpreter module is approximately 12K bytes long and runs either at the start of the large contiguous memory space (see Appendix C for further details peculiar to the target CPU) or in a separate RAM space. If there is a separate RAM space large enough to accommodate the interpreter, it should be used rather than the large contiguous space. (The exact size of the interpreter can be obtained by examining the first word of the SYSTEM.INTERP file once the PASCAL system is bootstrapped.)

1.1 Sample Configurations

To illustrate these requirements, take two cases. The first case is the simple one of a configuration that has 64K of RAM available. In the second case, the configuration provides a 16K RAM between 1000 hex and 5000 hex, a 36K byte RAM between 6000 hex and F000 hex, and a 300 hex bytes RAM between FD00 hex and FFFF hex. In both cases, assume the SBIOS is 300 hex bytes long, the bootstrap is 1800 hex bytes long, and the interpreter is 2A00 hex bytes long.



('#' indicates the presence of memory)

In the first case, there is no separate RAM space for the interpreter. Therefore, it must start where the large contiguous RAM space starts. Since there is RAM at 8000 and the large contiguous RAM starts before 3000 hex, the SBOOT8 is used. Finally, the SBIOS is located so as to break up as little of the large contiguous memory space as possible.

In the second case, there is a separate RAM space that is large enough to hold the interpreter. The interpreter is therefore located there. Moreover, there is also a miscellaneous RAM space large enough to hold the SBIOS. The SBIOS is therefore located there. Finally, since the large contiguous space does not start before 3000 hex, the SBOOTD copy of the bootstrap must be used at D000 hex.

2. Floppy Disk Requirements

It is necessary that the configuration of any machine that runs UCSD PASCAL have at least 175K bytes (350 PASCAL blocks) of floppy disk storage. This requirement arises from the fact that while it is possible to bootstrap with less space, it is virtually impossible to do anything of interest.

The UCSD PASCAL system is designed to work on any type of floppy medium. This includes mini-floppies, soft sectored floppies, hard sectored floppies, double density floppies, and double sided, double density floppies. The UCSD PASCAL Adaptable disk is distributed on an IBM 3740 soft sectored disk. If the target configuration does not include floppy drives capable of reading the bootstrap disk, a copy of the bootstrapping disk must be created on a disk (called the "target

medium") that the available floppy drives can read. The UCSD PASCAL system will then bootstrap from that disk.

2.1 Format of the UCSD PASCAL Adaptable System Disk

The UCSD PASCAL Adaptable disk is logically divided into three UCSD PASCAL disk images of 25 tracks apiece. The first disk image (tracks 0 through 24) contains a UCSD PASCAL Bootstrapping disk that boots under the SBOOT8 bootstrap (section II.1). The second disk image (tracks 25 through 49) contains a UCSD PASCAL Bootstrapping disk that boots under the SBOOTD bootstrap. The third disk image (tracks 50 through 74) contains the UCSD PASCAL Interpreter disk. The first disk image is accessible to the UCSD PASCAL system upon initial bootstrapping. For this reason, the SBOOT8 bootstrap is considered the default bootstrap. The second and third disk images are considered "packed".

A UCSD PASCAL disk image has 25 tracks, logically numbered 0 through 24. Each track contains 26 sectors, numbered 1 through 26, with 128 bytes per sector.

Logical track 0 is reserved for the UCSD PASCAL bootstrap. Sectors 1 and 2 contain the Primary bootstrap. Sectors 3 through 18 contain overlays (the Secondary bootstrap) called as bootstrapping progresses. Sectors 19 through 26 are not used.

Logical track 1, sectors 1 through 8 are reserved for the SBIOS tester (see sections II.3 and III.2) associated with the UCSD PASCAL bootstrap. Sectors 9 through 26 are occupied by the UCSD PASCAL directory.

Logical tracks 2 through 24 are used by the UCSD PASCAL system to store files.

Refer to Appendix C for any details specific to the target processor.

2.2 Preparing the UCSD PASCAL Bootstrapping Disk

The operations necessary to prepare a UCSD PASCAL Bootstrapping disk for use depend on the disk configuration and memory configuration of the target system. The UCSD PASCAL Bootstrapping disk image appropriate for use with the target configuration must occupy tracks 0 through 24 of the target medium.

If the target medium is an 8" soft sectored floppy and it is necessary (according to section II.1 and Appendix C) to bootstrap under the SBOOT8 bootstrap there is no work necessary. The SBOOT8 UCSD PASCAL Bootstrapping disk image is already on tracks 0 through 24 of the Adaptable System disk. If, instead, it is necessary to bootstrap

under the SBOOTD bootstrap, the contents of second disk image (tracks 25 through 49) must be copied onto tracks 0 through 24 of the Adaptable disk. Any means may be used to effect the transfer (eg. a native operating system, a ROM monitor, an assembly language program). The Adaptable disk can thus serve as the UCSD PASCAL Bootstrapping Disk.

If the target medium is not an 8" soft sectored floppy, it is necessary to transfer the appropriate (according to section II.1 and Appendix C) UCSD PASCAL Bootstrapping disk image from the Adaptable disk to the target medium. If it is necessary to bootstrap under the SBOOT8 bootstrap, the first disk image (tracks 0 through 24) must be transferred. If it is necessary to bootstrap under the SBOOTD bootstrap, the second disk image (tracks 25 through 49) must be transferred.

2.2.1 Creating a UCSD PASCAL Disk on Another Medium

In creating a UCSD PASCAL disk image on another floppy medium, the obvious task is to get the information on the UCSD PASCAL disk image onto the target medium. There are two commonly used methods: 1) find another computer that can support floppy drives that read the IBM 3740 disk and floppy drives that write the target floppy, or 2) find a computer that can read an IBM 3740 disk and transmit the contents over a serial line. The target computer must be capable of receiving data on the serial line and recording the data on the target disk. Any other method is acceptable as long as the end result is a UCSD PASCAL disk image on the target medium.

There is some data on the UCSD PASCAL disk image that must be transferred to equivalent locations on the target disk. The first 900 hex bytes (2304 bytes or 18 - 128 byte sectors) of track 0 must be transferred from track 0 of the UCSD PASCAL disk image disk to track 0 of the target disk. The bytes starting on the first sector of track 1 must be transferred to the target disk starting on the first sector of its track 1. All remaining data is transferred until the end of the UCSD PASCAL disk image is reached.

Note that for the purposes of initial bootstrapping, the information on a UCSD PASCAL disk image is recorded in contiguous sectors (ie: no sector interleaving is performed).

3. Providing I/O Routines - the SBIOS

The next requirement for bootstrapping the UCSD PASCAL system is the set of I/O routines needed to communicate with the disk and console. This is called the SBIOS (Simplified Basic Input/Output Subsystem). In most cases, these routines are derivations of routines that are already available in other programs or in monitor ROMs.

There are 15 entry points into the SBIOS:

ROUTINE NAME	VECTOR NUMBER	DESCRIPTION
SYSINIT	0	initialize machine
SYSHALT	1	exit UCSD PASCAL
CONINIT	2	console initialize
CONSTAT	3	console status
CONREAD	4	console input
CONWRIT	5	console output
SETDISK	6	set disk number
SETTRAK	7	set track number
SETSECT	8	set sector number
SETBUFR	9	set buffer address
DSKREAD	10	read sector from disk
DSKWRIT	11	write sector to disk
DSKINIT	12	reset disk
DSKSTRT	13	activate disk
DSKSTOP	14	de-activate disk

An SBIOS routine is called through a jump vector. The jump vector is a set of jumps contiguous in memory. In the case of the SBIOS there are 15 jumps, each one jumping to the start of a different SBIOS routine. The jumps are arranged in vector number order. To call a routine, a program must call the jump to the routine. This scheme allows a program to use the routines in the SBIOS without being dependent on the length or location of any particular routine.

(Note: The general algorithm used by a program calling an SBIOS routine is :
STEP 1: Multiply the vector number of the SBIOS routine by the number of bytes required to represent a jump instruction in the jump vector. This yields the position of the desired jump relative to the start of the jump vector.
STEP 2: Add the actual address of the start of the jump vector to the result from step 1. This gives the address of the jump to call.
STEP 3: Call the jump instruction. The instruction will immediately jump into the desired routine. Note that a program calling the SBIOS uses a subroutine call to call the desired jump instruction in the jump vector. Each SBIOS routine should take care to return to the calling routine.)

The functions and parameter protocols of each of the SBIOS routines are described in detail below. For an enumeration of the various CPUs and the registers used in parameter passing, refer to Appendix A. See Appendix C for restrictions on the use of interrupt vectors.

Note: the disk handling routines operate on a set of values defined to be "current". The values are

NAME	MEANING
CURDISK	Current floppy drive number
CURTRAK	Current track on CURDISK
CURSECT	Current sector on CURTRAK
CURBUFR	Current memory buffer address

3.1 SYSINIT

The SYSINIT routine is the first routine called when a system is booted. A pointer to the UCSD PASCAL interpreter jump table is passed to SYSINIT. This jump table is not used in initial bootstrapping attempts and should be ignored during such attempts. It is described in section IV.2.1.

Actions that are necessary to condition the system hardware are performed in SYSINIT. This includes setting up interrupt vectors, enabling RAM memories, and turning off any I/O devices that won't be used. Any other initialization necessary to make the machine run correctly should be done here.

3.2 SYSHALT

The SYSHALT routine is the last routine called by a PASCAL system. This routine is responsible for shutting all devices down in an orderly manner. It may also start any host operating system available.

3.3 CONINIT

The CONINIT routine is called both to initialize the console port and to report the status of the connection. Initialization includes preparation of the console hardware to send and receive characters. If the baud rate and parity bits can be set by software, CONINIT should configure the console to operate as quickly as possible and with no parity translation. Any interrupt vectors associated with console operation should be set in the SYSINIT routine (section II.3.1).

If the console can be detected to be off line (disconnected), a 9 should be returned as the status of the console connection. Otherwise, CONINIT returns a 0.

3.4 CONSTAT

The CONSTAT routine is called for a console status report. It returns two statuses. The first is the state of the console connection. If the console can be detected to be off line (disconnected), CONSTAT returns 9. Otherwise it returns 0. The second status is the state of the console input channel. If it can be determined that a character has been typed on the keyboard, CONSTAT returns FF hex; otherwise 0 is returned. (Note: a pending character is NOT read, its presence is merely reported).

3.5 CONREAD

The CONREAD routine is called both to receive a character from the keyboard and to report the status of the console connection. The status of the console connection is reported as with CONSTAT (0 for no error, 9 for off line). If the console appears to be on line, it is checked to see if a character is available from the keyboard; if not, CONREAD continues to check until there is. Once a character is available, it is read from the keyboard channel. If it can be determined that there was an error in transmission the character is returned, but 1 is returned as the status of the console connection. Note that the character must be returned exactly as read from the keyboard port. It is not desired that CONREAD turn the high order bit off.

3.6 CONWRIT

The CONWRIT routine is called both to write a character to the console and to report the status of the console connection. The status of the console connection is reported as with CONSTAT (0 for no error, 9 for off line). If the console appears to be on line, the character is transmitted when the console is ready to receive it. If there is an error in the transmission, the character is assumed lost and 1 is returned as the status of the console connection. It is not desired that CONWRIT pre-process the output character in any way other than what is necessary to correctly display the character on the console (for example, don't strip parity bits unless the terminal will not function properly with them set).

3.7 SETDISK

The SETDISK routine is called to set CURDISK. The disk number is passed solely to be recorded by the SBIOS for use with the DSKREAD, DSKWRIT, DSKINIT, DSKSTRT, and DSKSTOP routines. The disk numbers start at 0 and continue through 5. No hardware disk select is done and no status is returned.

3.8 SETTRAK

The SETTRAK routine is called to set CURTRAK. The track number is passed solely to be recorded by the SBIOS for use with the DSKREAD and DSKWRIT routines. Track numbers start at 0 and continue to one less than the highest numbered track on the disk. No hardware disk seek is done and no status is returned.

3.9 SETSECT

The SETSECT routine is called to set CURSECT. The sector number is passed solely to be recorded by the SBIOS for use with the DSKREAD and DSKWRIT routines. Sector numbers start at 1 and continue to the highest numbered sector on a track. No hardware disk operations are done and no status is returned.

3.10 SETBUFR

The SETBUFR routine is called to set CURBUFR. The buffer address is passed solely to be recorded by the SBIOS for use with the DSKREAD and DSKWRIT routines. No hardware disk operations are done and no status is returned.

3.11 DSKREAD

The DSKREAD routine is called to read a sector from a floppy disk and return a status for the operation. DSKREAD is responsible for ensuring that the CURDISK is selected and that the recording head is positioned above the CURTRAK. The CURSECT is read into memory starting at the CURBUFR. CURDISK, CURTRAK, CURSECT, and CURBUFR are assumed to be set before the call to DSKREAD by calls to SETDISK, SETTRAK, SETSECT, and SETBUFR. They are not changed by this routine.

If the read finishes without error, DSKREAD returns a status of 0. If the disk is determined not to be on line or not available, the returned status is 9. All other errors cause the returned status to be 1. This routine should always return with an error status rather than retry or hang on an error.

Before DSKREAD is called, it is assumed that DSKINIT has been called at least once for the CURDISK since the system was started and that DSKSTRT has been called for the CURDISK more recently than DSKSTOP.

3.12 DSKWRIT

The DSKWRIT routine is called to write a sector to a floppy disk and return a status for the operation. DSKWRIT is responsible for ensuring that the CURDISK is selected and that the recording head is positioned above the CURTRAK. The CURSECT is written from memory starting at the CURBUFR. CURDISK, CURTRAK, CURSECT, and CURBUFR are assumed to be set before the call to DSKWRIT by calls to SETDISK, SETTRAK, SETSECT, and SETBUFR. These values are not changed by this routine.

If the write finishes without error, DSKWRIT returns a status of 0. If the disk is determined not to be on line or not available, the returned status is 9. All other errors cause the returned status to be 16. This routine should always return with an error status rather than retry or hang on an error.

Before DSKWRIT is called, it is assumed that DSKINIT has been called at least once for the CURDSK since the system was started and that DSKSTRT has been called for the CURDSK more recently than DSKSTOP. Read after write checking is discouraged.

3.13 DSKINIT

The DSKINIT routine is called both to reset a disk and to return a status for the operation. The SETDISK and DSKSTRT routines are assumed to have been called to activate the CURDISK before DSKINIT is called. The recording head for the CURDISK is moved to track 0. The floppy drive is reset to its power-up state if possible and prepared for reading and writing. If the CURDISK is on line (the drive exists and there is a floppy loaded) and the DSKINIT is successful, the status of the operation is returned 0; otherwise, the status is returned 9.

This routine should always return with an error status rather than hang on an error. CURDISK, CURTRAK, CURSECT, and CURBUFR are not changed by this routine.

3.14 DSKSTRT

The DSKSTRT routine is called in preparation for a series of disk read, write, or init operations. The SETDISK routine is assumed to have been called to set the CURDISK before DSKSTRT is called. DSKSTRT operates on the CURDISK and performs any motor starting and head loading operations not done automatically as consequences of read, write, and init operations. No status is returned on the result of this operation. This routine is most useful when dealing with mini-floppy drives. It is expected that most 8" floppy drives will not need any action here.

3.15 DSKSTOP

The DSKSTOP routine is called at the end of a series of disk read, write, or init operations. The SETDISK routine is assumed to have been called to set the CURDISK before DSKSTOP is called. DSKSTOP operates on the CURDISK and performs any motor stopping and head unloading operations not done automatically after read, write, and init operations. No status is returned on the result of this operation. This routine is most useful when dealing with mini-floppy drives. It is expected that most 8" floppy drives will not need any action here.

4. Where to Get the SBIOS Routines

The SBIOS routines are very basic routines of the type found in the lowest levels of most operating systems or ROM monitors. If the target machine does not have these routines or similar routines in ROM, they should be available either from parts of other existing programs or from the manufacturer of the hardware involved.

5. What To Do With SBIOS Routines

The SBIOS is edited and assembled on whatever operating system or other computer is available. It may also be assembled by hand.

When the SBIOS is complete, the individual routines should be checked against the specifications in section II.3. Special attention should be given to the values returned describing the result of the requested I/O.

5.1 Organization of the SBIOS

The SBIOS should be organized with the jump vector at the beginning of the SBIOS, followed by data space and code. A sample SBIOS might look like:

```
SBIOS                               ; Beginning of the SBIOS
        JUMP    SYSINIT             ; Jump to SYSINIT routine
        JUMP    SYSHALT             ; Jump to SYSHALT routine
        JUMP    CONSTAT             ; Jump to CONSTAT routine
        JUMP    CONREAD             ; Jump to CONREAD routine
             .
             .
             .
        JUMP    DSKSTRT             ; Jump to DSKSTRT routine
        JUMP    DSKSTOP             ; Jump to DSKSTOP routine

CURDISK .WORD                       ; Temporary area
CURTRAK .WORD

SYSINIT
             .
             .
             .
        RET                         ; Make sure to return to caller

SYSHALT
        HALT                        ; Croaking on this machine is simple

             .
             .
             .
```

Chapter III
BOOTSTRAPPING UCSD PASCAL

Once an SBIOS and a UCSD PASCAL Bootstrapping disk appropriate for the target configuration exist , either the SBIOS test program or the UCSD PASCAL bootstrap may be executed. The appropriate code must be loaded into memory and parameters must be loaded onto the processor stack.

1. Loading the SBIOS into Memory

As mentioned in section II.1, the SBIOS should be loaded either into a small, isolated RAM or into the end of the large, contiguous RAM space (the isolated RAM is preferred). An existing operating system or ROM monitor must used to load the SBIOS into a suitable memory space.

2. Testing the SBIOS

It is advisable that the SBIOS be tested before any attempts to bootstrap UCSD PASCAL are made.

The SBIOS tester is a utility program that resides on the UCSD PASCAL Bootstrapping disk and tests each SBIOS routine. It is passed a set of parameters that describes the configuration of the target system (see section III.2.2). A lengthy test will be performed on each floppy drive numbered 0 through the highest floppy drive number. Each track and sector on each disk is written and read. There are also random disk seek tests and console read/write tests.

The chances of bootstrapping the UCSD PASCAL system are quite good if the SBIOS passes these tests.

The SBIOS tester must be loaded into memory at the same location as the UCSD PASCAL bootstrap (either 8000 hex or D000 hex; see section II.1). The SBIOS tester is recorded on the first *1024* bytes of Track 1 on the UCSD PASCAL Bootstrapping disk (track 1, sectors 1 through 8 on the IBM 3740 eight inch disk). Any available method may be used to load this code into the appropriate memory space. Possible methods include using a manufacturer-supplied operating system or a small assembly language program to read the code.

2.1 Parameters to the SBIOS Tester

A number of parameters must be passed on the processor stack to the SBIOS tester. These describe the configuration of the target machine, including bootstrapping disk characteristics, memory configurations, and other miscellaneous information. Some of them are not used directly by the SBIOS tester. They are defined here for completeness, as they will be referenced by other sections.

The stack pointer for a given CPU must be initialized according to the processor-specific instructions in Appendix C. Several 16-bit word parameters are pushed onto the stack in the following order:

```
    top of stack ---> highest numbered floppy drive to test
                      desired address of interpreter
                      address of SBIOS in memory
                      address of lowest word of contiguous memory
                      address of highest word of contiguous memory
                      number of tracks per disk
                      number of sectors per track
                      number of bytes per sector
                      interleaving factor
                      first PASCAL track
                      track-to-track skew
                      maximum number of sectors per track for all disks
```

2.1.1 Highest Numbered Floppy Drive to Test

The SBIOS tester will attempt to test all disks between the system floppy drive (numbered 0) and the specified floppy drive number. For example, if this parameter is 1, the SBIOS will attempt to test drives 1 and 0.

2.1.2 Address of the Interpreter

The next parameter is the desired address of the interpreter module. This is the value calculated for the start of the interpreter using the formulae in section II.1.

2.1.3 Address of the SBIOS

The next parameter is the location in memory of the SBIOS. The memory location of the SBIOS is determined by formulae found in section II.1. The SBIOS is loaded into memory starting at this location in section III.1.

2.1.4 Bounds of the Large Contiguous RAM

The next two parameters are the addresses of the first and last WORDS in the large (minimum 36K bytes), contiguous RAM space. The UCSD PASCAL system assumes it can use all words within this range. Therefore the SBIOS must not be located within this range, nor should there be any ROM or lack of memory. (Note: If, as in Case 1 of section II.1.1, the SBIOS occupies the end of the large, contiguous RAM space, the "last word" of contiguous RAM should be reported to occur immediately before the start of the SBIOS). If the interpreter module is located within this space, it must be wholly contained and start at the beginning of the space. See Appendix C for details specific to each processor.

2.1.5 Tracks per Disk

The number of tracks on disk is the number of tracks on the physical floppy medium.

2.1.6 Sectors per Track

The number of sectors on disk is the number of physical sectors on a physical track of a floppy.

2.1.7 Bytes per Sector

The number of bytes per sector is the number of bytes in a physical sector recorded on the floppy disk. It is assumed to be the number of bytes SBIOS DSKREAD and DSKWRIT routines will transfer at one time. Allowable values are 128, 256, and 512.

2.1.8 Miscellaneous Parameters

The next four parameters are necessary for configuration after UCSD PASCAL has been bootstrapped once (see section IV). They are involved in alternate disk recording/formatting strategies aimed at optimizing floppy drive performance and thus the overall performance of the UCSD PASCAL system. Initial values are provided for the first bootstrapping attempts.

The initial value for the interleaving factor is 1. The initial value for the first PASCAL track is 1. The initial value for the track-to-track skew is 0. The initial value for the maximum number of sectors per track for all disks is the value supplied in section III.2.1.6.

2.1.9 Sample Configurations

To illustrate valid parameter values, consider the two cases presented as examples in section II.1.1. Assume in case 1 there are 2 eight inch, single density, IBM 3740 floppy drives on line, with 77 tracks/disk, 26 sectors/track, and 128 bytes/sector. For case 2, assume there are 6 mini-floppy drives on line, with 35 tracks/disk, 10 sectors/track, and 256 bytes/sector.

The parameter stack appears as follows:

Case 1			Case 2
top of stack --->			
0001	++	highest numbered floppy drive to test	++ 0005
0000 hex	++	desired address of interpreter	++ 1000 hex
FD00 hex	++	address of SBIOS in memory	++ FD00 hex
0000 hex	++	address of low word of contiguous RAM	++ 6000 hex
FCFE hex	++	address of high word of contiguous RAM	++ F000 hex
004D hex	++	number of tracks per disk	++ 0023 hex
001A hex	++	number of sectors per track	++ 000A hex
0080 hex	++	number of bytes per sector	++ 0100 hex
0001	++	interleaving factor	++ 0001
0001	++	first PASCAL track	++ 0001
0000	++	track-to-track skew	++ 0000
001A hex	++	maximum number of sectors per track	++ 000A hex

2.2 Executing the SBIOS Tester

Once the SBIOS tester is loaded into memory, and the set of parameters has been pushed onto the stack, the SBIOS tester must be entered (via a jump instruction, not a call instruction; the stack must appear exactly as in section III.2.1) at the location into which it is loaded. It operates as follows:

Step 1: The SYSINIT routine is called to initialize the SBIOS. The CONINIT routine is called to initialize the console.

Step 2: If step 1 is successful, the prompt message is printed on the console:

Insert blank disks into all drives then hit the <return> key.
WARNING!! The contents of ALL disks in the tested drives will be destroyed by the SBIOS read/write tests!

Step 3: If the CONWRIT routine prints the prompt in Step 2 correctly and the CONREAD returns a carriage return when the <return> key is hit, the highest numbered drive is declared the "current disk" by a call to SETDISK.

Step 4: The DSKSTRT routine is called to start the "current disk", and the DSKINIT routine is called to initialize the floppy drive hardware. If the status of the DSKINIT request is returned non-zero, the message:

Disk x is not on line

is printed on the console and the test continues at step 9. If the status of the DSKINIT request is returned zero, the message:

Testing disk x

is printed on the console. If neither of these messages appear on the console, either the CONWRIT routine is not functioning, the DSKSTRT routine is not functioning, or the DSKINIT routine is not functioning.

Step 5: If Step 4 succeeds, a unique data pattern is written on each sector of each track using the SETTRAK, SETSECT, and DSKWRIT routines. Before the write occurs, the numbers of the track and sector are printed on the console:

Writing track xx, sector yy .

The track and sector number are printed in hex (base 16). If there is an error attempting the write operation, an error message is printed on the console:

Bad data transfer on track xx, sector yy .

Again, the track and sector number are printed in hex.

Step 6: If Step 5 executes correctly, each sector on each track is read and checked to make sure the data recorded in Step 5 is actually on each sector. The SETTRAK, SETSECT, and DSKREAD routines are used for this. As each sector is checked, its location is printed on the console:

Reading track xx, sector yy

If a pattern is incorrectly recorded or there is an I/O error on the read, the error message of Step 5 is printed on the console.

Step 7: If Step 6 executes correctly, a read/write head seek test is initiated. A unique data pattern is written on one sector of each track using the same SBIOS routines employed in Step 5. The tracks are accessed starting in the middle of the disk and oscillating on either side of the middle track until the outer tracks are reached. As each sector is recorded, its location is printed on the console as

in Step 5. If there is an error attempting the write operation, the error message of Step 5 is printed on the console.
Step 8: If Step 7 executes correctly, the content of each sector written in step 7 is checked for correctness. The same SBIOS routines used in Step 6 are used here. As each sector is read, its location is printed on the console as in Step 6. If there is an error on the read or an error in the contents, the error message of Step 5 is printed on the console.
Step 9: If Step 8 completes successfully, the DSKSTOP routine is called to de-activate the "current disk". The "current disk" number is decremented. If the new "current disk" number is non-negative, the test resumes at STEP 4 with the new floppy drive.
Step 10: If all steps terminate successfully, a termination prompt is printed on the console:

Test complete

The SYSHALT routine is called and the test will terminate.

2.3 After the SBIOS Tester

If the SBIOS tester successfully completes Steps 1 through 10, an attempt may be made at bootstrapping the UCSD PASCAL system. If, however, any one of the SBIOS tester tests fail, the cause of the failure must be found and eliminated before a successful attempt at bootstrapping the UCSD PASCAL system can be made.

3. Loading the UCSD PASCAL Bootstrap

If the SBIOS is not already in memory, it must be loaded according to section III.1. The UCSD PASCAL bootstrap must then be loaded into memory at either 8000 hex or D000 hex, depending on the machine's memory configuration (see section II.1 for details). The bootstrap is recorded on the first 256 bytes of Track 0 on the UCSD PASCAL Bootstrapping disk (track 0, sectors 1 and 2 on the IBM 3740 eight inch disk). Any available method may be used to load this code into the appropriate memory space. Possible methods include using a manufacturer-supplied operating system or a small assembly language program that calls the already-resident SBIOS to read the code. An example of such a program for each type of CPU is provided in Appendix B.

4. Executing the UCSD PASCAL bootstrap

The UCSD PASCAL bootstrap requires that the parameters described in section III.2.2 be on the top of the processor stack (the number of disks to test must be zero). Once the configuration parameters are on the stack, the SBIOS is in memory, and the UCSD PASCAL bootstrap is in memory, the UCSD PASCAL bootstrap is ready to execute. This is done by executing a jump instruction to the location into which the UCSD PASCAL bootstrap is loaded in section III.3. Notice that the UCSD PASCAL bootstrap is not "call"ed; the stack must appear exactly as in section III.2.2 when the UCSD PASCAL bootstrap executes.

The bootstrapping process may take as long as two or three minutes. This is not a permanent condition. It may be remedied in section IV.

5. Checking the UCSD PASCAL System

At this point, the UCSD PASCAL system should have bootstrapped. There are a few simple tests to perform that check the interaction between the SBIOS and the UCSD PASCAL system.

The first is observation of the output on the console. There should be a welcoming message followed by the system version number and the date on which the UCSD PASCAL Bootstrap disk was created. Following that, the standard system prompt line should appear (refer to the UCSD PASCAL user's manual for the exact form of this). If these outputs do not appear, almost anything could be wrong. Check the values passed to the UCSD PASCAL bootstrap on the stack. In addition, either the disk read routines or the console output routines may be nonfunctional.

The next test is to hit the 'F' key on the console. this should invoke the file manager. The system floppy drive should read several sectors. Another prompt line should appear. If these actions do not occur, the disk read routines or the console input/output routines may not work.

The final test is to hit the 'D' key on the console. Next, type the current date (eg. 12-JAN-79) followed by the <return> key. Finally, type 'D' again. If the correct date is not displayed, the disk write routines may be at fault.

6. Utility Programs on the Bootstrapping Disk

A listing of the directory of the UCSD PASCAL Bootstrapping disk can be obtained by entering the F)iler and typing 'L' followed by ':' and a <carriage return>. The following files are provided:

FILE NAME	DESCRIPTION
SYSTEM.INTERP	UCSD PASCAL Interpreter
SYSTEM.PASCAL	UCSD PASCAL Operating System
SYSTEM.FILER	UCSD PASCAL File Handler
SYSTEM.MISCINFO	Terminal description file
SYSTEM.LIBRARY	Long Integer and Real Number library
FINDPARAMS.CODE	Program to determine optimal disk recording format
DISKCHANGE.CODE	Program to reformat a UCSD PASCAL disk
DISKSIZE.CODE	Program to change the size of a UCSD PASCAL disk
BOOTER.CODE	Program to transfer bootstraps from disk to disk

7. Disk number mapping

The UCSD PASCAL system accesses floppy disk drives by number. The mapping between SBIOS unit numbers (as passed to the SETDISK routine) and UCSD PASCAL unit numbers is:

SBIOS UNIT	UCSD PASCAL UNIT
0	4
1	5
2	9
3	10
4	11
5	12

When referring to a floppy disk while under control of the UCSD PASCAL system, the UCSD PASCAL unit number should be used.

8. Preparing Release Disks for Use

All SofTech Microsystems software compatible with the Adaptable System is released in the same format as the Adaptable System disk (section II.2.1). There are three UCSD PASCAL disk images per 8" soft sectored floppy disk.

If the target configuration does not include a floppy drive capable of reading a release disk, the disk images can be extracted for use by the UCSD PASCAL system by transferring the appropriate set of tracks to the target medium as in section II.2.2.

If the target configuration contains a floppy drive capable of reading a release disk, the DISKCHANGE program can be used to extract a given floppy image. This is done by X)ecuting the DISKCHANGE program. It will ask for the disk drive numbers (4 = SBIOS disk 0, 5 = SBIOS disk 1, 9 = SBIOS disk 2, etc.) involved in the transfer. A transfer interleaving factor is also requested. This may be small (eg. 2) for a fast floppy drive or large (eg. 7) for a slower drive. This factor is relevent to the speed of the transfer, not to the format of the data.

Information is requested about the source recording format then about the destination recording format. The source interleaving and skew factors must be 1 and 0, respectively. The first interleaved track is the physical track number associated with logical track 1 of the desired disk image (either track 1, 26, or 51). The destination interleaving factor, skew factor, and first interleaving track are 1, 0, and 1, respectively, unless action is taken according to section IV.1.

To unpack an entire release disk, X)ecute the DISKCHANGE program to transfer the first floppy image to a floppy. Repeat the process, transferring the second and third floppy images to a second and third floppy. The release disk should be kept as a backup.

Note that if the DISKCHANGE program is used to extract a disk that is intended to be booted directly, the destination disk must be formatted with 1 to 1 interleaving, zero sector skew, and a first PASCAL track of 0. If further interleaving is necessary (according to section IV.1), the DISKCHANGE program must be executed again and will operate starting at PASCAL track 1 or 2. (This procedure is necessary to assure that the interleaving of track 0 is 1 to 1.)

9. Accessing the UCSD PASCAL System Programs

The sole purpose of UCSD PASCAL Bootstrapping disk is to aid in the developement of bootstraps. There is no need for most UCSD PASCAL system programs in this process. Hence, they are provided on the UCSD PASCAL System disk rather than on the UCSD PASCAL Bootstrapping disk.

The System disk contains three disk images and may be unpacked as described in section III.7. The first disk image, SYSTEM1, contains a UCSD PASCAL system that may be booted once a bootstrap and a UCSD PASCAL interpreter are transferred to it. The bootstrap is transferred by X)ecuting the BOOTER program on the UCSD PASCAL Bootstrapping disk to copy the bootstrap on the UCSD PASCAL Bootstrapping disk to the SYSTEM1 disk. The UCSD PASCAL interpreter may be transferred by using the F)iler T)ransfer command to transfer the SYSTEM.INTERP file on the UCSD PASCAL Bootstrapping disk to the SYSTEM1 disk. The SYSTEM1 disk may be booted in the same manner as the UCSD PASCAL Bootstrapping disk. Other system programs are recorded on the second and third disk images of the UCSD PASCAL System disk.

Once the SYSTEM1 disk is booted, a systems program may be executed by typing the associated letter when the system prompt line appears (eg. "E" for editor, "C" for compiler). Refer to the UCSD PASCAL User's Manual for full details. A system program invoked in this manner must reside on some floppy that is on line, not necessarily the SYSTEM1 floppy.

The SYSTEM.PASCAL file and the SYSTEM.INTERP file MUST be on any disk that is booted directly. The SYSTEM.SYNTAX, SYSTEM.MISCINFO, and SYSTEM.LIBRARY files must also be on the booted disk if the UCSD PASCAL system is to make use of them.

Note that in order to use an assembler, the assembler must be named SYSTEM.ASSMBLER. It may be necessary to enter the F)iler and use the C)hange command to change the name of an assembler not so named. Any assembler information files (eg. Z80.OPCODES) must reside on the same disk as the assembler code file.

10. Backing Up the Bootstrapping Disk

The UCSD PASCAL Bootstrapping disk and the SYSTEM1 disk should be backed up at this point. To do so, enter the F)iler and do a volume to volume T)ransfer. (See the UCSD PASCAL User's Manual for details). This will copy the UCSD PASCAL system-areas to a backup disk. Use the BOOTER program on the UCSD PASCAL Bootstrapping disk to copy the bootstrap to the backup disk.

11. Customizing a UCSD PASCAL Disk Image

All UCSD PASCAL disk images are configured to contain 153 PASCAL blocks (512 bytes per block). Thus, disk space is wasted on floppies that can hold more than 153 PASCAL blocks. The DISKSIZE program is provided to alter the configuration of a UCSD PASCAL disk image so it utilizes an entire floppy disk. When X)ecuted, it asks for the drive number (4 = SBIOS disk 0, 5 = SBIOS disk 1, 9 = SBIOS disk 2, etc) that contains the disk to be reconfigured. It then asks for the number of blocks the disk may hold. This can be calculated by the following formula using the parameters described in section III.2.1:

```
(# tracks per disk - first PASCAL track)
* (# of sectors per track) / (512 / # of bytes per sector)
```

Chapter IV
ADDING CAPABILITIES

The UCSD PASCAL system bootstrapped in sections I through III is intended as a minimal configuration. It does not have the capability of communicating with a printer, a serial line, a system clock, or disks formatted differently than the system disks. It also lacks the capability of a turn-key bootstrap, and disk accesses are not optimized for speed. This section describes how to add these features, assuming the target machine will support them.

1. Quick Improvements

There are simple improvements that can be made to the minimal configuration that can dramatically improve system performance. Changing the disk recording format can yield a significant speed improvement. Depending on the target machine configuration, it may also be possible to simplify the bootstrapping procedure.

1.1 Changing the Disk Recording Format

The UCSD PASCAL Bootstrapping disk is formatted with 1 to 1 sector interleaving and zero sector track-to-track skew. (The sector interleaving factor is a function of the ratio of the processor speed to the floppy disk revolution speed. The track-to-track skew is a function of the ratio of the floppy disk revolution speed to the floppy drive seek time. The particular recording scheme chosen for the UCSD PASCAL Bootstrapping disk makes no assumptions about either of these ratios.) It may be possible to significantly decrease the average disk access time by using recording parameters better suited to the target configuration.

The recording format of the UCSD PASCAL Bootstrapping disk may be changed as follows (note: it is advisable that a backup copy be made of any floppy undergoing this procedure before continuing. Refer to the section on the T)ransfer command of the F)iler in the UCSD PASCAL User's Manual for instructions on disk-to-disk transfers.):

1) eX)ecute the FINDPARAMS program found on the UCSD PASCAL Bootstrapping disk. It will aid in determining the optimal interleaving and skew factors.
2) eX)ecute the DISKCHANGE program found on the UCSD PASCAL Bootstrapping disk. It reformats the UCSD PASCAL Bootstrapping disk according to new sector interleaving, track-to-track skew, and first PASCAL track parameters. (For this exercise, the first PASCAL track should be 1 unless action is taken according to section IV.1.2.1 or IV.1.2.2.)
3) Change the sector interleaving and the track-to-track skew parameters passed to the UCSD PASCAL bootstrap in section III.4 to match the new disk format.
4) Reboot the UCSD PASCAL Bootstrapping disk.

Note that if the UCSD PASCAL Bootstrapping disk is not interleaved 1 to 1, the SBIOS tester can NOT be used. Note, also, that all floppies intended to be read by the UCSD PASCAL system must have the same disk recording parameters. Thus, in changing the format of the UCSD PASCAL Bootstrapping disk, the formats of all other floppies intended for use with the UCSD PASCAL system must be changed (use the DISKCHANGE program; be warned, however, the DISKCHANGE program destroys all disk images recorded on tracks after the end of the desired floppy image. If the preservation of other disk images is important, extract them according to section III.8 BEFORE the DISKCHANGE program is executed).

There are many soft sectored 8" UCSD PASCAL floppies in the field whose disk format is 2 to 1 interleaving, 6 sector skew, and first PASCAL track of 1. This format is suggested for 8" soft sectored floppies if compatibility with other users' floppies is important. However, the DISKCHANGE program can be used to convert any UCSD PASCAL disk to a compatible format.

1.2 Making a Simpler Bootstrap

In order to produce a turnkey UCSD PASCAL system, the target configuration must have some mechanism that, when invoked, reads the contents of a pre-defined area of disk into a pre-defined area in memory. This typically comes in the form of a boot-button that transfers control to a boot-rom that reads the contents of a sector into memory and executes it.

If this feature is available in the target configuration, a Primary Bootstrap may be written that loads the appropriate configuration parameters onto the stack, loads the SBIOS into memory from a pre-defined location on the UCSD PASCAL disk, loads the UCSD PASCAL Secondary bootstrap, and executes it. The location of the Primary bootstrap on the floppy disk is defined to be the sector the boot-rom reads.

If this bootstrapping approach is to be adopted, room must be found on the UCSD PASCAL Bootstrapping disk to store the Primary Bootstrap and the SBIOS without overwriting the UCSD PASCAL system. The areas on the UCSD PASCAL disk that are not used by the UCSD PASCAL system and on which the Primary Bootstrap and the SBIOS may be stored are:
1) Track 0, sectors 1 and 2
2) Track 0, sectors 19 through the end of track 0
3) Track 1, sectors 1 through 8 (the sectors to which they are mapped by the interleaving scheme)

Note that if there is an interleaving factor other than 1 to 1, the sectors into which Track 1, sectors 1 through 8 are mapped are available rather than sectors 1 through 8 themselves.

The new Primary Bootstrap must:
1) Read the SBIOS from the UCSD PASCAL Bootstrapping disk (from wherever it is recorded) into the memory space in which it is intended to execute.
2) Load the Secondary Bootstrap from the UCSD PASCAL Bootstrapping disk starting at Track 0, sector 3 into the memory space in which it is intended to execute. If SBOOT8 is chosen as the bootstrap in section II.1, the Secondary Bootstrap must be loaded starting at 8200 hex. IF SBOOTD is chosen as the bootstrap in section II.1, the Secondary Bootstrap must be loaded starting at D200 hex.
3) Load the configuration parameters onto the stack (sections III.2.2 and III.4).
4) Jump to the beginning of the Secondary Bootstrap (either 8200 hex or D200 hex according to step 2).

1.2.1 Alternate Floppy Locations for the SBIOS

If the free area does not provide ample storage space for the SBIOS, the UCSD PASCAL system can be reconfigured to start on track 2 instead of track 1. This leaves track 1 un-interleaved (1 to 1) and available for storage of the SBIOS. The reconfiguration procedure is:
1) eX)ecute the DISKCHANGE program on the UCSD PASCAL Bootstrapping disk. It will reformat the UCSD PASCAL Bootstrapping disk according to the new first PASCAL track (track 2). The sector interleaving and track-to-track skew factors should remain unchanged.
2) Change the first PASCAL track parameter passed to the UCSD PASCAL bootstrap in section III.4 to reflect the new disk format.

1.2.2 Alternate Locations for the Primary Bootstrap

If the boot-rom is defined to read a Primary bootstrap that must be recorded on parts of Track 0, sectors 3 through 18, the Secondary Bootstrap must be moved to a different area on the floppy. It may be moved elsewhere on track 0 or onto tracks 1 or 2. If it is moved to tracks 1 or 2, the UCSD PASCAL Bootstrapping disk must be reformatted as in section IV.1.2.1. In any case, the Primary Bootstrap must read the Secondary Bootstrap into memory from WHEREVER it is recorded on the floppy disk.

1.2.3 Backing Up the Bootstrapping Disk

The UCSD PASCAL Bootstrapping disk should be backed up at this point. To do so, enter the F)iler and do a volume to volume T)ransfer. (See the UCSD PASCAL User's Manual for details.) This will copy the UCSD PASCAL system-areas of the UCSD PASCAL Bootstrapping disk to a backup disk. Use the BOOTER program on the UCSD PASCAL Bootstrapping disk to copy the bootstrap to the backup disk.

2. Extending the I/O Capabilities

The UCSD PASCAL system is designed to interface to a remote port (serial line), a printer, a real time clock, user-defined devices, and floppy disks of possibly dissimilar formats. These devices may be added to the UCSD PASCAL system by enhancing the SBIOS and by providing an Extended SBIOS and then reconfiguring the UCSD PASCAL interpreter.

2.1 The UCSD PASCAL Interpreter Jump Vector

As mentioned in section II.3.1, a pointer to the UCSD PASCAL interpreter jump vector is passed as a parameter to the SYSINIT routine. This jump vector can be used for certain extensions to the I/O system that require handshaking with the UCSD PASCAL interpreter. SBIOS routines call the UCSD PASCAL interpreter routines in the same manner as UCSD interpreter routines call SBIOS routines (see section II.3 for details). The routines provided in this manner are:

ROUTINE NAME	VECTOR NUMBER	DESCRIPTION
POLLUNITS	0	polls character-oriented devices
DSKCHNG	1	change parameters of CURDISK

The functions and parameter protocols of these routines are described in detail below. For an enumeration of the various CPU's and the registers used in parameter passing, refer to Appendix A.

Note that if an SBIOS that uses the interpreter jump vector is called by user-written software not running under the UCSD PASCAL system (eg. a bootloader routine written for section III.3), care must be taken that the user-written software provide a substitute interpreter jump vector. A pointer to the substitute vector must be provided by the user-written software to the SBIOS SYSINIT routine. The routines accessed through the jump vector need only return to the caller.

2.1.1 POLLUNITS

The POLLUNITS routine may be called by the SBIOS DSKINIT, DSKREAD, and DSKWRIT routines to check the console, remote, and printer input drivers for available data. Any available data is read from the appropriate device and saved in that device's input queue. No registers are altered by this routine.

2.1.2 DSKCHNG

The UCSD PASCAL interpreter communicates with disks that are assumed to be formatted according to a "current format", CURFORM. The CURFORM is initialized by the UCSD PASCAL bootstrap according to the values passed to it on the stack (see section III.2.2). This format is assumed to describe the system disk.

The DSKCHNG routine may be called by the SBIOS SETDISK routine to change CURFORM to conform to the recording format of the floppy disk on which SETDISK is called. A pointer to a disk information block is passed to the DSKCHNG routine. This information block contains six 16-bit words:

WORD	DEFINITION
0	number of tracks per disk
1	number of sectors per track
2	number of bytes per sector
3	interleaving factor
4	first PASCAL track
5	track-to-track skew

The semantics of these words are consistent with the definitions presented in sections III.2.2.5-III.2.2.7, IV.1.1, IV.1.2.1, and IV.1.2.2.

Note that this routine must be called to change CURFORM if the current call to SETDISK is to set CURDISK to a disk whose format is not CURFORM. Also note that all processor registers (except the stack pointer) are assumed destroyed by this routine.

2.2 Enhancing the Floppy Disk Drivers

It is possible to configure a UCSD PASCAL system to access floppy disks that are not formatted the same as the system disk. For example, a UCSD PASCAL system may be configured to communicate with a double density system disk, a single density disk, and two mini-floppies. It is also possible to allow polling of the character-oriented devices when such polling does not interfere with disk driver timing.

2.2.1 Allowing Multiple Floppy Disk Formats

The SBIOS SETDISK routine may be modified so as to call the UCSD PASCAL interpreter DSKCHNG routine whenever the format of the new CURDISK is different to that of the prior CURDISK.

Under this scheme, if any accessible floppy drive has more sectors per track than the system disk, the maximum sectors per track parameter that is passed to the UCSD PASCAL bootstrap (section III.2) must be changed to the number of sectors per track of that floppy format.

2.2.2 Polling During Disk Accesses

Polling of the console, remote, and printer input channels may be done during the execution of the SBIOS DSKINIT, DSKREAD, and DSKWRIT routines. This is accomplished by calling the UCSD PASCAL interpreter routine POLLUNITS.

The POLLUNITS could be repeatedly called, for instance, while waiting for a seek to terminate. The integrity of each device's type-ahead queue is enhanced by making this call; it increases the frequency with which these channels are sampled.

2.3 The Extended SBIOS

In section II.3, fifteen SBIOS vectors are defined. The Extended SBIOS allows communication with a printer, a remote, a clock, and user-defined devices. It contains an additional 13 SBIOS vectors:

ROUTINE NAME	VECTOR NUMBER	DESCRIPTION
PRNINIT	15	printer initialize
PRNSTAT	16	printer status
PRNREAD	17	printer read
PRNWRIT	18	printer write
REMINIT	19	remote initialize
REMSTAT	20	remote status
REMREAD	21	remote read
REMWRIT	22	remote write
USRINIT	23	user devices initialize
USRSTAT	24	user devices status
USRREAD	25	user devices read
USRWRIT	26	user devices write
CLKREAD	27	system clock read

The vectors corresponding to vector numbers 0 through 14 are exactly as defined in section II.3.

The printer and remote vectors duplicate those for the console except that they communicate through the printer and remote ports instead of the console port.

The USR vectors are provided to allow a PASCAL program to communicate with devices other than those pre-defined (for instance, a grain elevator). A PASCAL program accesses these devices through the UNITREAD, UNITWRITE, UNITCLEAR, and UNITSTATUS intrinsics with unit numbers 128 through 255. See the UCSD PASCAL User's Manual for further information on these intrinsics.

The functions and parameter protocols of each of the Extended SBIOS routines are described in detail below. For an enumeration of the various CPUs and the registers used in parameter passing, please refer to Appendix A. See Appendix C for restrictions on the use of interrupt vectors.

2.3.1 PRNINIT

The PRNINIT routine is called both to initialize the printer port and to report the status of the connection. Initialization includes preparation of the printer hardware to send and receive characters. If the baud rate and parity bits can be set by software, PRNINIT should configure the printer to operate as quickly as possible and with no parity translation. Any interrupt vectors associated with printer operation should be set in the SYSINIT routine (section II.3.1).

If the printer can be detected to be off line (disconnected) or there is no printer driver, a 9 should be returned as the status of the printer connection. Otherwise, PRNINIT returns a 0. Note that it is not desired that a form feed be sent to the printer from PRNINIT.

2.3.2 PRNSTAT

The PRNSTAT routine is called for a printer input status report. It returns two statuses. The first is the state of the printer connection. If the printer can be detected to be off line (disconnected) or there is no printer driver, PRNSTAT returns 9. Otherwise it returns 0. The second status is the state of the printer input channel. If it can be determined that a character is available on the printer input channel (if there is one), PRNSTAT returns FF hex; otherwise 0 is returned. (Note: a pending character is NOT read, its presence is merely reported).

2.3.3 PRNREAD

The PRNREAD routine is called both to receive a character from the printer input channel and to report the status of the printer connection. The status of the printer connection is reported as with PRNSTAT (0 for no error, 9 for off line). If the printer appears to be on line, it is checked to see if a character is available on the input channel; if not, PRNREAD continues to check until there is. Note that if there is no printer input channel, the wait is indefinite. Once a character is available, it is read from the printer keyboard channel. If it can be determined that there was an error in transmission the character is returned, but 1 is returned as the status of the printer connection. It is also necessary that PRNREAD return the character exactly as read from the input port (without turning the high order bit off).

2.3.4 PRNWRIT

The PRNWRIT routine is called both to write a character to the printer and to report the status of the printer connection. The status of the printer connection is reported as with PRNSTAT (0 for no error, 9 for off line). If the printer appears to be on line, the character is transmitted when the printer is ready to receive it. If there is an error in the transmission, the character is assumed lost and 1 is returned as the status of the printer connection. It is not desired that PRNWRIT pre-process the output character in any way other than what is necessary to correctly display the character on the printer (for example, don't strip parity bits unless the printer will not function properly with them set).

2.3.5 REMINIT

The REMINIT routine is called both to initialize the remote (extra serial) port and to report the status of the connection. Initialization includes preparation of the remote hardware to send and receive characters. If the baud rate and parity bits can be set by software, REMINIT should configure the remote to operate as quickly as possible and with no parity translation. Any interrupt vectors associated with remote operation should be set in the SYSINIT routine (section II.3.1).

If the remote can be detected to be off line (disconnected) or there is no remote driver, a 9 should be returned as the status of the remote connection. Otherwise, REMINIT returns a 0.

2.3.6 REMSTAT

The REMSTAT routine is called for a remote status report. It returns two statuses. The first is the state of the remote connection. If the remote can be detected to be off line (disconnected) or there is no remote driver, REMSTAT returns 9. Otherwise it returns 0. The second status is the state of the remote input channel. If it can be determined that a character has been received on the remote input channel, REMSTAT returns FF hex; otherwise 0 is returned. (Note: a pending character is NOT read, its presence is merely reported).

2.3.7 REMREAD

The REMREAD routine is called both to receive a character from the remote input channel and to report the status of the remote connection. The status of the remote connection is reported as with REMSTAT (that is, 0 for no error, 9 for off line). If the remote appears to be on line, it is checked to see if a character is available from the remote input channel; if not, REMREAD continues to check until there is. Once a character is available, it is read from the remote input channel. If it can be determined that there was an error in transmission the character is returned, but 1 is returned as the status of the remote connection. It is also necessary that REMREAD return the character exactly as read from the keyboard port (without turning the high order bit off).

2.3.8 REMWRIT

The REMWRIT routine is called both to write a character to the remote and to report the status of the remote connection. The status of the remote connection is reported as with REMSTAT (that is, 0 for no error, 9 for off line). If the remote appears to be on line, the character is transmitted when the remote output channel is ready to receive it. If there is an error in the transmission, the character is assumed lost and 1 is returned as the status of the remote connection. It is not desired that REMWRIT pre-process the output character in any way (for example, don't strip parity bits unless the remote will not function properly with them set).

2.3.9 USRINIT

The USRINIT routine is called both to initialize a user-defined device and to return the status of the device. The device number is passed as a parameter to USRINIT. If the corresponding device can be determined not to be on line, USRINIT returns a 9 for the device status. Otherwise the device is reset to its power up condition and the device status is returned 0. Any interrupt vectors should be initialized in the SYSINIT routine (section II.3.1).

If the device statuses 0 and 9 do not provide adequate information about the initialization of the device, alternate device statuses may be defined in the range of 100 through 255.

2.3.10 USRSTAT

The USRSTAT routine is called to get both a simple status and a more detailed status on a user-defined device. The device number, a pointer to a status record, and a direction indicator are passed to USRSTAT. If the corresponding device can be determined not to be on line, USRSTAT returns a 9 for the simple device status; otherwise a 0 is returned. If the device statuses 0 and 9 do not provide adequate information about the device status, alternate device statuses may be defined in the range of 100 through 255.

A pointer is provided to a 30 word status record. Its format and contents is dictated by the needs of the device being checked. If the low order bit of the direction indicator is 0, USRSTAT records the condition of the output channel in the status record. If the low order bit of the direction indicator is 1, USRSTAT records the condition of the input channel of the device in the status record. Additional status request information may be passed in the high order three bits of the direction indicator.

2.3.11 USRREAD

The USRREAD routine is called both to get the status of a user-defined device and to perform a read operation on it. USRREAD is passed the device number, a pointer to a buffer, and three extra parameters. A read operation is performed on the specified device into the specified buffer. The extra parameters may be defined as the device requires.

The result of the read operation on the specified device is returned as the status of the device (as with USRINIT: 0 for ok, 9 for off line, 100-255 for other).

2.3.12 USRWRIT

The USRWRIT routine is called both to get the status of a user-defined device and to perform a write operation on it. USRWRIT is passed the device number, a pointer to a buffer, and three extra parameters. A write operation is performed on the specified device from the specified buffer. The extra parameters may be defined as the device requires.

The result of the write operation on the specified device is returned as the status of the device (as with USRINIT: 0 for ok, 9 for off line, 100-255 for other).

2.3.13 CLKREAD

The CLKREAD routine is called both to get the status of the real-time clock and to get the current time. The status is returned 0 if a real-time clock exists and is enabled; otherwise 9 is returned. The current time is returned as a 32 bit integer representing 1/60ths of a second since midnight. If this is not determinable, it should be the number of 1/60ths of a second since the last call to the SYSINIT routine. This implies the system clock should be started by the SYSINIT routine of the SBIOS if it is not already operating when the system is bootstrapped. If there is no system clock on line, the time returned should be zero.

If the real-time clock does not keep time in 1/60ths of a second, CLKREAD must transform the clock time to approximate 1/60ths of a second.

2.4 Testing the Extended SBIOS

SofTech Microsystems does not provide test routines for the Extended SBIOS as are available for the SBIOS (see section III.2). The Extended SBIOS routines may, however, be tested individually under the control of a host operating system or under the UCSD PASCAL system (for a description of assembly language procedures, see the UCSD PASCAL User Manual).

2.5 Bootstrapping with the Extended SBIOS

The UCSD PASCAL interpreter must be reconfigured before the SBIOS extensions can be used. Interpreter reconfiguration instructions specific to each type of CPU are provided in Appendix D. Once the interpreter is reconfigured, the UCSD PASCAL system may be bootstrapped under the Extended SBIOS by substituting the Extended SBIOS for the current SBIOS in the bootstrapping sequence (see section III).

Appendix A
VECTOR LISTS AND REGISTER ASSIGNMENTS

1. Z80/8080 Processor

1.1 Z80/8080 SBIOS

All registers except the stack pointer are assumed destroyed by the SBIOS. When it is specified that an SBIOS routine must return an I/O status, the A register is defined to carry it. The stack pointer may be modified only insofar as necessary to remove parameters and return addresses from the stack. The Vector Offsets are the numbers added to the start of the SBIOS in order to find the "jump" to the respective SBIOS routine.

ROUTINE NAME	VECTOR OFFSET	PARAMETERS
SYSINIT	00 hex	Given BC = pointer to jump table
SYSHALT	03 hex	<none>
CONINIT	06 hex	Returns A = 0 if keyboard on line = 9 if keyboard off line
CONSTAT	09 hex	Returns A = 0 if keyboard on line = 9 if keyboard off line C = 0 if no char = FF if char available
CONREAD	0C hex	Returns A = 0 if ok = 1 if error = 9 if off line C = input character
CONWRIT	0F hex	Given C = output character Returns A = 0 if ok = 9 if off line
SETDISK	12 hex	Given C = disk number
SETTRAK	15 hex	Given C = track number
SETSECT	18 hex	Given C = sector number
SETBUFR	1B hex	Given BC = buffer address
DSKREAD	1E hex	Returns A = 0 if ok = 1 if error = 9 if off line
DSKWRIT	21 hex	Returns A = 0 if ok = 16 if error = 9 if off line
DSKINIT	24 hex	Returns A = 0 if ok = 9 if off line
DSKSTRT	27 hex	<none>
DSKSTOP	2A hex	<none>

1.2 Z80/8080 Extended SBIOS

The register assignment rules are the same as those described in section A.1.1. The following vectors are appended to the jump vector after the DSKSTOP entry. All references to SP indicate the contents of the processor stack. Parameters mentioned on the SP line are 16-bit word quantities in the order they are popped off the stack. Extra parameters 1, 2, and 3 passed to the USR routines are the byte count, block number, and control word, respectively, in the UNITREAD or UNITWRITE call (section IV.2.3).

ROUTINE NAME	VECTOR OFFSET	PARAMETERS
PRNINIT	2D hex	Returns A = 0 if printer on line 9 if printer off line
PRNSTAT	30 hex	Returns A = 0 if printer on line = 9 if printer off line C = 0 if no char = FF if char available
PRNREAD	33 hex	Returns A = 0 if ok = 1 if error = 9 if off line C = input character
PRNWRIT	36 hex	Given C = output character Returns A = 0 if ok = 9 if off line
REMINIT	39 hex	Returns A = 0 if remote on line 9 if remote off line
REMSTAT	3C hex	Returns A = 0 if remote on line = 9 if remote off line C = 0 if no char = FF if char available
REMREAD	3F hex	Returns A = 0 if ok = 1 if error = 9 if off line C = input character
REMWRIT	42 hex	Given C = output character Returns A = 0 if ok = 9 if off line
USRINIT	45 hex	Given C = device number Returns A = 0 if device on line = 9 if device off line
USRSTAT	48 hex	Given SP = return address I/O direction status record pointer device number Returns A = 0 if device on line = 9 if device off line
USRREAD	4B hex	Given SP = return address extra parameter 2 extra parameter 1 pointer to buffer device number extra parameter 3 Returns A = 0 if device on line = 9 if device off line
USRWRIT	4E hex	Given SP = return address extra parameter 2 extra parameter 1 pointer to buffer device number extra parameter 3 Returns A = 0 if device on line = 9 if device off line
CLKREAD	51 hex	Returns A = 0 if clock on line

= 9 if clock off line
DE = least significant word
HL = most significant word

1.3 Z80/8080 Interpreter

The Vector Offsets are the numbers added to the start of the UCSD PASCAL interpreter jump table in order to find the "jump" to the respective routine.

ROUTINE NAME	VECTOR OFFSET	PARAMETERS
POLLUNITS	0	<none>
DSKCHNG	3	Given BC = pointer to format

2. 6502 Processor

2.1 6502 SBIOS

All registers except the stack pointer are assumed destroyed by the SBIOS. When it is specified that an SBIOS routine must return an I/O status, the X register is defined to carry it. The Vector Offsets are the numbers added to the start of the SBIOS in order to find the "jump" to the respective SBIOS routine. The XA register pair is a 16-bit quantity where the X register contains the high order byte and the A register contains the low order byte.

ROUTINE NAME	VECTOR OFFSET	PARAMETERS
SYSINIT	00 hex	Given XA = pointer to jump table
SYSHALT	03 hex	<none>
CONINIT	06 hex	Returns X = 0 if keyboard on line = 9 if keyboard off line
CONSTAT	09 hex	Returns X = 0 if keyboard on line = 9 if keyboard off line A = 0 if no char = FF if char available
CONREAD	0C hex	Returns X = 0 if ok = 1 if error = 9 if off line A = input character
CONWRIT	0F hex	Given A = output character Returns X = 0 if ok = 9 if off line
SETDISK	12 hex	Given A = disk number
SETTRAK	15 hex	Given A = track number
SETSECT	18 hex	Given A = sector number
SETBUFR	1B hex	Given XA = buffer address
DSKREAD	1E hex	Returns X = 0 if ok = 1 if error = 9 if off line
DSKWRIT	21 hex	Returns X = 0 if ok = 16 if error = 9 if off line
DSKINIT	24 hex	Returns X = 0 if ok = 9 if off line
DSKSTRT	27 hex	<none>
DSKSTOP	2A hex	<none>

2.2 6502 Extended SBIOS

The register assignment rules are the same as those described in section A.2.1. The following vectors are appended to the jump vector after the DSKSTOP entry. All references to SP indicate the contents of the processor stack. Parameters mentioned on the SP line are 16-bit word quantities in the order they are popped off the stack. The least significant part of a 16-bit word is popped off the stack first. Extra parameters 1, 2, and 3 passed to the USR routines are the byte count, block number, and control word, respectively, in the UNITREAD or UNITWRITE call (section IV.2.3).

VECTOR LISTS AND REGISTER ASSIGNMENTS

ROUTINE NAME	VECTOR OFFSET	PARAMETERS
PRNINIT	2D hex	Returns X = 0 if printer on line 9 if printer off line
PRNSTAT	30 hex	Returns X = 0 if printer on line = 9 if printer off line A = 0 if no char = FF if char available
PRNREAD	33 hex	Returns X = 0 if ok = 1 if error = 9 if off line A = input character
PRNWRIT	36 hex	Given A = output character Returns X = 0 if ok = 9 if off line
REMINIT	39 hex	Returns X = 0 if remote on line 9 if remote off line
REMSTAT	3C hex	Returns X = 0 if remote on line = 9 if remote off line A = 0 if no char = FF if char available
REMREAD	3F hex	Returns X = 0 if ok = 1 if error = 9 if off line A = input character
REMWRIT	42 hex	Given A = output character Returns X = 0 if ok = 9 if off line
USRINIT	45 hex	Given A = device number Returns X = 0 if device on line = 9 if device off line
USRSTAT	48 hex	Given SP = return address I/O direction status record pointer device number Return X = 0 if device on line = 9 if device off line
USRREAD	4B hex	Given SP = return address extra parameter 2 extra parameter 1 pointer to buffer device number extra parameter 3 Return X = 0 if device on line = 9 if device off line
USRWRIT	4E hex	Given SP = return address extra parameter 2 extra parameter 1 pointer to buffer device number extra parameter 3 Return X = 0 if device on line = 9 if device off line
CLKREAD	51 hex	Return X = 0 if clock on line

= 9 if clock off line
SP = least significant word
most significant word

2.3 6502 Interpreter

The Vector Offsets are the numbers added to the start of the UCSD PASCAL interpreter jump table in order to find the "jump" to the respective routine. The XA register pair is a 16-bit quantity where the X register is the most significant byte.

ROUTINE NAME	VECTOR OFFSET	PARAMETERS
POLLUNITS	0	<none>
DSKCHNG	3	Given XA = pointer to format

3. 6800 Processor

3.1 6800 SBIOS

All registers except the stack pointer are assumed destroyed by the SBIOS. When it is specified that an SBIOS routine must return an I/O status, the B register is defined to carry it. The Vector Offsets are the numbers added to the start of the SBIOS in order to find the "jump" to the respective SBIOS routine. The AB register pair is a 16-bit quantity where the A register is the most significant byte.

VECTOR LISTS AND REGISTER ASSIGNMENTS

ROUTINE NAME	VECTOR OFFSET	PARAMETERS
SYSINIT	00 hex	Given AB = pointer to jump table
SYSHALT	03 hex	<none>
CONINIT	06 hex	Returns B = 0 if keyboard on line = 9 if keyboard off line
CONSTAT	09 hex	Returns B = 0 if keyboard on line = 9 if keyboard off line A = 0 if no char = FF if char available
CONREAD	0C hex	Returns B = 0 if ok = 1 if error = 9 if off line A = input character
CONWRIT	0F hex	Given A = output character Returns B = 0 if ok = 9 if off line
SETDISK	12 hex	Given A = disk number
SETTRAK	15 hex	Given A = track number
SETSECT	18 hex	Given A = sector number
SETBUFR	1B hex	Given AB = buffer address
DSKREAD	1E hex	Returns B = 0 if ok = 1 if error = 9 if off line
DSKWRIT	21 hex	Returns B = 0 if ok = 16 if error = 9 if off line
DSKINIT	24 hex	Returns B = 0 if ok = 9 if off line
DSKSTRT	27 hex	<none>
DSKSTOP	2A hex	<none>

3.2 6800 Extended SBIOS

The register assignment rules are the same as those described in section A.3.1. The following vectors are appended to the jump vector after the DSKSTOP entry. All references to SP indicate the contents of the processor stack. Parameters mentioned on the SP line are 16-bit word quantities in the order they are popped off the stack. The most significant part of a 16-bit word is popped off the stack first. Extra parameters 1, 2, and 3 passed to the USR routines are the byte count, block number, and control word, respectively, in the UNITREAD or UNITWRITE call (section IV.2.3).

ROUTINE NAME	VECTOR OFFSET	PARAMETERS
PRNINIT	2D hex	Returns B = 0 if printer on line 9 if printer off line
PRNSTAT	30 hex	Returns B = 0 if printer on line = 9 if printer off line A = 0 if no char = FF if char available
PRNREAD	33 hex	Returns B = 0 if ok = 1 if error = 9 if off line A = input character
PRNWRIT	36 hex	Given A = output character Returns B = 0 if ok = 9 if off line
REMINIT	39 hex	Returns B = 0 if remote on line 9 if remote off line
REMSTAT	3C hex	Returns B = 0 if remote on line = 9 if remote off line A = 0 if no char = FF if char available
REMREAD	3F hex	Returns B = 0 if ok = 1 if error = 9 if off line A = input character
REMWRIT	42 hex	Given A = output character Returns B = 0 if ok = 9 if off line
USRINIT	45 hex	Given A = device number Returns B = 0 if device on line = 9 if device off line
USRSTAT	48 hex	Given SP = return address I/O direction status record pointer device number Return B = 0 if device on line = 9 if device off line
USRREAD	4B hex	Given SP = return address extra parameter 2 extra parameter 1 pointer to buffer device number extra parameter 3 Return B = 0 if device on line = 9 if device off line
USRWRIT	4E hex	Given SP = return address extra parameter 2 extra parameter 1 pointer to buffer device number extra parameter 3 Return B = 0 if device on line = 9 if device off line
CLKREAD	51 hex	Return B = 0 if clock on line

= 9 if clock off line
SP = least significant word
most significant word

3.3 6800 Interpreter

The Vector Offsets are the numbers added to the start of the UCSD PASCAL interpreter jump table in order to find the "jump" to the respective routine. The AB register pair is a 16-bit quantity where the A register is the most significant byte.

ROUTINE NAME	VECTOR OFFSET	PARAMETERS
POLLUNITS	0	<none>
DSKCHNG	3	Given AB = pointer to format

Appendix B
SAMPLE BOOTSTRAP LOADERS

1. Z80 Sample Bootstrap Loader

```
; This routine loads the Primary bootstrap part of SBOOT8 from track 0,
; sectors 1 and 2 of the UCSD PASCAL Bootstrap disk.  The I/O routines
; in the SBIOS (assumed resident before this program is executed) are
; called to access the floppy.  If there is any problem reading the
; bootstrap, the SYSHALT routine is called.  Notice that the jump vector
; offset are as per the formula given in section II.3 with 3 bytes as the
; length of a jump instruction.

        .PROC   LOAD

BIOSJP  .EQU    0FD00H          ; Assume the SBIOS is loaded here
BOOTAD  .EQU    8000H           ; The address of SBOOT8
SECSIZE .EQU    80H             ; Number of bytes in a sector

SYSINIT .EQU    00H             ; Offset into jump vector of SYSINIT jump
SYSHALT .EQU    03H             ; Offset into jump vector of SYSHALT jump
SETDISK .EQU    12H             ; Offset into jump vector of SETDISK jump
SETTRAK .EQU    15H             ; Offset into jump vector of SETTRAK jump
SETSECT .EQU    18H             ; Offset into jump vector of SETSECT jump
SETBUFR .EQU    1BH             ; Offset into jump vector of SETBUFR jump
DSKREAD .EQU    1EH             ; Offset into jump vector of DSKREAD jump
DSKINIT .EQU    24H             ; Offset into jump vector of DSKINIT jump
DSKSTRT .EQU    27H             ; Offset into jump vector of DSKSTRT jump
DSKSTOP .EQU    2AH             ; Offset into jump vector of DSKSTOP jump

        .MACRO  SBIOS           ; Call an SBIOS routine
        CALL    BIOSJP + %1
        .ENDM

LOADR                           ; Bootstrap loader
        SBIOS   SYSINIT         ; Initialize the SBIOS
        LD      C,0             ; Set to bootstrap disk
        SBIOS   SETDISK
        SBIOS   DSKSTRT         ; Start the disk
        SBIOS   DSKINIT         ; Reset the disk - ready for action
        AND     A               ; Was the operation successful ?
        JP      NZ,CALLHLT
        LD      C,0             ; So far, so good, set to track 0
        SBIOS   SETTRAK

        LD      BC,BOOTAD       ; Set up to read into RAM code buffer
        SBIOS   SETBUFR
        LD      C,1             ; Set to read sector 1
        SBIOS   SETSECT
        SBIOS   DSKREAD         ; Actually read the sector
        AND     A               ; Was the read successful ?
        JP      NZ,CALLHLT

        LD      BC,BOOTAD+SECSZ ; Set up to read into RAM code buffer
        SBIOS   SETBUFR
```

```
        LD      C,2             ; Set to read sector 2
        SBIOS   SETSECT
        SBIOS   DSKREAD         ; Actually read the sector
        AND     A               ; Was the read successful ?
        JP      NZ,CALLHLT

        SBIOS   DSKSTOP         ; De-activate the disk
        RET                     ; Return so the user can set up the
                                ; parameter stack

CALLHLT                         ; Call the SBIOS SYSHALT routine to die
        SBIOS   SYSHALT
        JP      CALLHLT         ; Just in case SYSHALT messes up and returns

        .END
```

2. 6502 Sample Bootstrap Loader

```
; This routine loads the Primary bootstrap part of SBOOT8 from track 0,
; sectors 1 and 2 of the UCSD PASCAL Bootstrap disk.  The I/O routines
; in the SBIOS (assumed resident before this program is executed) are
; called to access the floppy.  If there is any problem reading the
; bootstrap, the SYSHALT routine is called.  Notice that the jump vector
; offset are as per the formula given in section II.3 with 3 bytes as the
; length of a jump instruction.

        .PROC   LOAD

BIOSJP  .EQU    0FD00H          ; Assume the SBIOS is loaded here
BOOTADR .EQU    80H             ; High byte of the address of SBOOT8
SECSIZE .EQU    80H             ; Number of bytes in a sector

SYSINIT .EQU    00H             ; Offset into jump vector of SYSINIT jump
SYSHALT .EQU    03H             ; Offset into jump vector of SYSHALT jump
SETDISK .EQU    12H             ; Offset into jump vector of SETDISK jump
SETTRAK .EQU    15H             ; Offset into jump vector of SETTRAK jump
SETSECT .EQU    18H             ; Offset into jump vector of SETSECT jump
SETBUFR .EQU    1BH             ; Offset into jump vector of SETBUFR jump
DSKREAD .EQU    1EH             ; Offset into jump vector of DSKREAD jump
DSKINIT .EQU    24H             ; Offset into jump vector of DSKINIT jump
DSKSTRT .EQU    27H             ; Offset into jump vector of DSKSTRT jump
DSKSTOP .EQU    2AH             ; Offset into jump vector of DSKSTOP jump

        .MACRO  SBIOS           ; Call an SBIOS routine
        JSR     BIOSJP + %1
        .ENDM

LOADR                           ; Bootstrap loader
        SBIOS   SYSINIT         ; Initialize the SBIOS
        LDA     #0              ; Set to bootstrap disk
        SBIOS   SETDISK
        SBIOS   DSKSTRT         ; Start the disk
        SBIOS   DSKINIT         ; Reset the disk - ready for action
```

```
        TXA                         ; Was the operation successful ?
        BNE     CALLHLT
        LDA     #0                  ; So far, so good, set to track 0
        SBIOS   SETTRAK

        LDA     #0
        LDX     #BOOTADR            ; Set up to read into RAM code buffer
        SBIOS   SETBUFR
        LDA     #1                  ; Set to read sector 1
        SBIOS   SETSECT
        SBIOS   DSKREAD             ; Actually read the sector
        TXA                         ; Was the read successful ?
        BNE     CALLHLT

        LDA     #SECSZ % 100H       ; This is SECSZ modulo 100H
        LDX     #BOOTADR + <SECSZ / 100H>
        SBIOS   SETBUFR             ; Set up to read into next RAM code buffer
        LDA     #2                  ; Set to read sector 2
        SBIOS   SETSECT
        SBIOS   DSKREAD             ; Actually read the sector
        TXA                         ; Was the read successful ?
        BNE     CALLHLT

        SBIOS   DSKSTOP             ; De-activate the disk
        RTS                         ; Return so the user can set up the
                                    ; parameter stack

CALLHLT                             ; Call the SBIOS SYSHALT routine to die
        SBIOS   SYSHALT
        JMP     CALLHLT             ; Just in case SYSHALT messes up and returns

        .END
```

3. 6800 Sample Bootstrap Loader

```
; This routine loads the Primary bootstrap part of SBOOT8 from track 0,
; sectors 1 and 2 of the UCSD PASCAL Bootstrap disk.  The I/O routines
; in the SBIOS (assumed resident before this program is executed) are
; called to access the floppy.  If there is any problem reading the
; bootstrap, the SYSHALT routine is called.  Notice that the jump vector
; offset are as per the formula given in section II.3 with 3 bytes as the
; length of a jump instruction.

        .PROC   LOAD

BIOSJP  .EQU    0FD00H              ; Assume the SBIOS is loaded here
BOOTADR .EQU    80H                 ; High byte of the address of SBOOT8
SECSIZE .EQU    80H                 ; Number of bytes in a sector

SYSINIT .EQU    00H                 ; Offset into jump vector of SYSINIT jump
SYSHALT .EQU    03H                 ; Offset into jump vector of SYSHALT jump
SETDISK .EQU    12H                 ; Offset into jump vector of SETDISK jump
SETTRAK .EQU    15H                 ; Offset into jump vector of SETTRAK jump
```

SAMPLE BOOTSTRAP LOADERS

```
SETSECT .EQU    18H             ; Offset into jump vector of SETSECT jump
SETBUFR .EQU    1BH             ; Offset into jump vector of SETBUFR jump
DSKREAD .EQU    1EH             ; Offset into jump vector of DSKREAD jump
DSKINIT .EQU    24H             ; Offset into jump vector of DSKINIT jump
DSKSTRT .EQU    27H             ; Offset into jump vector of DSKSTRT jump
DSKSTOP .EQU    2AH             ; Offset into jump vector of DSKSTOP jump

        .MACRO  SBIOS           ; Call an SBIOS routine
        JSR     BIOSJP + %1
        .ENDM

LOADR                           ; Bootstrap loader
        SBIOS   SYSINIT         ; Initialize the SBIOS
        CLR     A               ; Set to bootstrap disk
        SBIOS   SETDISK
        SBIOS   DSKSTRT         ; Start the disk
        SBIOS   DSKINIT         ; Reset the disk - ready for action
        TST     B               ; Was the operation successful ?
        BNE     CALLHLT
        CLR     A               ; So far, so good, set to track 0
        SBIOS   SETTRAK

        CLR     B
        LDA     A,#BOOTADR      ; Set up to read into RAM code buffer
        SBIOS   SETBUFR
        LDA     A,#1            ; Set to read sector 1
        SBIOS   SETSECT
        SBIOS   DSKREAD         ; Actually read the sector
        TST     B               ; Was the read successful ?
        BNE     CALLHLT

        LDA     B,#SECSZ % 100H ; This is SECSZ modulo 100H
        LDA     A,#BOOTADR + <SECSZ / 100H>
        SBIOS   SETBUFR         ; Set up to read into next RAM code buffer
        LDA     A,#2            ; Set to read sector 2
        SBIOS   SETSECT
        SBIOS   DSKREAD         ; Actually read the sector
        TST     B               ; Was the read successful ?
        BNE     CALLHLT

        SBIOS   DSKSTOP         ; De-activate the disk
        RTS                     ; Return so the user can set up the
                                ; parameter stack

CALLHLT                         ; Call the SBIOS SYSHALT routine to die
        SBIOS   SYSHALT
        JMP     CALLHLT         ; Just in case SYSHALT messes up and returns

        .END
```

Appendix C
GENERAL NOTES

1. Z80 Notes

1.1 Memory Configuration Constraints

All parameters in section III.2.2 referring to areas in memory are WORD pointers (the least significant byte of the address must be even). For example, the highest word address in memory is specified as FFFE hex instead of FFFF hex.

The address of the interpreter (section III.2.1.2) must be a page boundary. That is, the least significant byte of the address must be 0. If the interpreter starts on the beginning of the large contiguous RAM space, the lower bound of the large contiguous RAM space must be a page boundary as well.

The SBIOS may use any interrupt vectors or restart vectors it needs without fear of conflict with any UCSD PASCAL software.

The stack pointer should be initialized to the highest even address in the large contiguous memory space. The configuration parameters described in section III.2.2 must be pushed onto this stack.

The ADAP8 disk contains the disk images for the UCSD PASCAL Bootstrapping disks and Interpreter disk configured for the 8080 processor. The ADAPZ disk contains the disk images for the UCSD PASCAL Bootstrapping disks and Interpreter disk configured for the Z80 processor.

2. 6502 Notes

2.1 Memory Configuration Constraints

All parameters in section III.2.2 referring to areas in memory are WORD pointers (the least significant byte of the address must be even). For example, the highest word address in memory is specified as FFFE hex instead of FFFF hex.

Pages 0 and 1 (addresses 0 through 1FF hex) must not be considered part of any address space where an interpreter, SBIOS, or UCSD PASCAL system can be loaded. It is strictly a data area for the interpreter and the SBIOS. Further constraints on page 0 are described in section C.2.2.

The address of the interpreter (section III.2.1.2) must be a page boundary. That is, the least significant byte of the address must be 0. If the interpreter starts on the beginning of the large contiguous RAM space, the lower bound of the large contiguous RAM space must be a page boundary as well.

The SBIOS may use any interrupt vectors it needs without fear of conflict with any UCSD PASCAL software.

The stack pointer must be initialized to FF hex.

2.2 Alternate Bootstrap Locations

The UCSD PASCAL systems on the Adaptable disk LO PAGE assume that they have exclusive use of page 0 addresses 0 through 7F hex. If this is the case, the ADAP00 disk should be used to furnish the UCSD PASCAL Bootstrapping disk and Interpreter disk images.

If page 0 addresses 0 through 7F hex are not available for the exclusive use of the UCSD PASCAL system then page 0 addresses 80 hex through FF hex must be. If this is the case, the HI PAGE disk should be used to furnish the UCSD PASCAL Bootstrapping disk and Interpreter disk images.

3. 6800 Notes

3.1 Memory Configuration Constraints

All parameters in section III.2.2 referring to areas in memory are WORD pointers (the least significant byte of the address must be even). For example, the highest word address in memory is specified as FFFE hex instead of FFFF hex.

Page 0 (addresses 0 through FF hex) must not be considered part of any address space where an interpreter, SBIOS, or UCSD PASCAL system can be loaded. It is strictly a data area for the interpreter and the SBIOS. Further constraints on page 0 are described in section C.3.2.

The address of the interpreter (section III.2.1.2) must be a page boundary. That is, the least significant byte of the address must be 0. If the interpreter starts on the beginning of the large contiguous RAM space, the lower bound of the large contiguous RAM space must be a page boundary as well.

The SBIOS may use any interrupt vectors it needs without fear of conflict with any UCSD PASCAL software.

The stack pointer should be initialized to the highest even address in the large contiguous memory space. The configuration parameters described in section III.2.2 must be pushed onto this stack.

3.2 Alternate Bootstrap Locations

The UCSD PASCAL systems on the Adaptable disk LO PAGE assume that they have exclusive use of page 0 addresses 0 through 7F hex. If this is the case, the ADAP00 disk should be used to furnish the UCSD PASCAL Bootstrapping disk and Interpreter disk images.

If page 0 addresses 0 through 7F hex are not available for the exclusive use of the UCSD PASCAL system then page 0 addresses 80 hex through FF hex must be. If this is the case, the HI PAGE disk should be used to furnish the UCSD PASCAL Bootstrapping disk and Interpreter disk images.

Appendix D
RECONFIGURING THE UCSD PASCAL INTERPRETER

1. Reconfiguring the Z80/8080 Interpreter

The UCSD PASCAL Interpreter disk contains modules that can be linked together to form a UCSD PASCAL interpreter for the Z80 or 8080 processor (see Appendix C). It must be extracted from the UCSD PASCAL Adaptable disk according to section III.7.

The UCSD PASCAL Interpreter disk contains utility programs and the BIOS, INTER, INTERP, RSP, and TERTBOOT modules:

FILE NAME	PURPOSE
INTERP.CODE	Contains no real number capabilities.
INTERP.FP.CODE	Capable of real number operations and transcendental functions.
RSP.CODE	I/O Interface between Z80/8080 interpreter and BIOS modules.
BIOS.CODE	No console, printer, or remote software queuing. Smallest of the BIOS modules.
BIOS.C.CODE	Contains console software queuing.
BIOS.CR.CODE	Contains console and remote software queuing.
BIOS.CRP.CODE	Contains console, remote, and printer software queuing. Largest of the BIOS modules.
INTER.CODE	Interfaces with the SBIOS.
INTER.X.CODE	Interfaces with the Extended SBIOS.
TERTBOOT.CODE	Final bootstrap for Z80/8080 interpreter.
BIOSLINKER.CODE	Program that links modules together to form a UCSD PASCAL interpreter.
SQUISH.CODE	Program that prepares a linked UCSD PASCAL interpreter for booting.

To create a new UCSD PASCAL Interpreter, perform the following steps:
1) X)ecute the BIOSLINKER program.
2) Pick the module of the INTERP class that has the desired characteristics. The INTERP.CODE module occupies much less run-time memory than the INTERP.FP.CODE module. Type the name of the chosen module as the response to the first linker prompt. Remember to specify the volume name as part of the module name.

WARNING : If the UCSD PASCAL system is informed there is a system clock (through the SETUP program) and the INTERP.CODE module is linked into the running interpreter, an execution error will occur at the end of a compilation when the compiler tries to use floating point numbers to calculate the compile speed.
3) Type 'RSP.CODE' as the response to the second linker prompt.
4) Pick the module of the BIOS class that has the desired characteristics. If an Extended SBIOS is not being used, only the BIOS.CODE and the BIOS.C.CODE modules should be used. Type the name of the chosen module as the response to the third linker prompt.
5) Pick the module of the INTER class that has the desired characteristics. The INTER.CODE module must be used if an SBIOS is being used. The INTER.X.CODE module must be used if an Extended SBIOS is being used. Type the name of the chosen module as the response to the fourth linker prompt.
6) Type 'TERTBOOT.CODE' as the response to the fifth linker prompt.
5) Type a <carriage return> in response to the next three linker prompts. The linker will terminate with a linked version of the desired configuration in the file SYSTEM.WRK.CODE on the system disk.
6) X)ecute the SQUISH program.
7) Enter 'SYSTEM.WRK.CODE' as the response to the first prompt.
8) Enter 'NEW.INTERP' as the response to the second prompt. The SQUISH program will create an object file named NEW.INTERP that contains the UCSD PASCAL Interpreter object.
9) Enter the F)iler by typing 'F'. Save a copy of the current interpreter by C)hanging SYSTEM.INTERP to OLD.INTERP. Make the new interpreter current by C)hanging NEW.INTERP to SYSTEM.INTERP.
10) Back up the UCSD PASCAL Bootstrapping disk as suggested in section IV.1.2.3.

The initial interpreter configuration on the UCSD PASCAL Bootstrapping disk is INTERP.CODE, RSP.CODE, BIOS.C.CODE, INTER.CODE, and TERTBOOT.CODE.

2. Reconfiguring the 6502 Interpreter

The UCSD PASCAL Interpreter disk contains modules that can be linked together to form a UCSD PASCAL interpreter for the 6502 processor (see Appendix C). It must be extracted from the UCSD PASCAL Adaptable disk according to section III.7.

The UCSD PASCAL Interpreter disk contains utility programs and the BIOS, INTER, INTERP, RSP, and TERTBOOT modules:

MODULE NAME	FEATURES
INTERP.CODE	Contains no real number capabilities.
INTERP.FP.CODE	Capable of real number operations but no transcendental functions.
BIOS.CODE	No console, printer, or remote software queuing. Smallest of the BIOS modules.
BIOS.C.CODE	Contains console software queuing.
BIOS.CR.CODE	Contains console and remote software queuing.
BIOS.CRP.CODE	Contains console, remote, and printer software queuing. Largest of the BIOS modules.
INTER.CODE	Interfaces to the SBIOS.
INTER.X.CODE	Interfaces to the Extended SBIOS.
TERTBOOT.CODE	Final bootstrap for the 6502 interpreter.

To create a new UCSD PASCAL Interpreter, perform the following steps:
1) X)ecute the BIOSLINKER program.
2) Pick the module of the INTERP class that has the desired characteristics. The INTERP.CODE module occupies much less run-time memory than the INTERP.FP.CODE module. Type the name of the chosen module as the response to the first linker prompt. Remember to specify the volume name as part of the module name.

If the INTERP.FP.CODE module is chosen, transcendental functions may be used by including 'USES TRANSCEND;' as the second line of a PASCAL program.

WARNING : If the UCSD PASCAL system is informed there is a system clock (through the SETUP program) and the INTERP.CODE module is linked into the running interpreter, an execution error will occur at the end of a compilation when the compiler tries to use floating point numbers to calculate the compile speed.
3) Pick the module of the BIOS class that has the desired characteristics. If an Extended SBIOS is not being used, only the BIOS.CODE and the BIOS.C.CODE modules should be used. Type the name of the chosen module as the response to the second linker prompt.
4) Pick the module of the INTER class that has the desired characteristics. The INTER.CODE module must be used if an SBIOS is being used. The INTER.X.CODE module must be used if an Extended SBIOS is being used. Type the name of the chosen module as the response to the third linker prompt.
5) Type 'TERTBOOT.CODE' as the response to the fourth linker prompt.
6) Type a <carriage return> in response to the next three linker prompts. The linker will terminate with a linked version of the desired configuration in the file SYSTEM.WRK.CODE.
7) X)ecute the SQUISH program.
8) Enter 'SYSTEM.WRK.CODE' as the response to the first prompt.
9) Enter 'NEW.INTERP' as the response to the second prompt. The

SQUISH program will create an object file named NEW.INTERP that contains the UCSD PASCAL Interpreter object.
10) Enter the F)iler by typing 'F'. Save a copy of the current interpreter by C)hanging SYSTEM.INTERP to OLD.INTERP. Make the new interpreter current by C)hanging NEW.INTERP to SYSTEM.INTERP. (Note that if the UCSD PASCAL system is using the first half of page 0, the name of the interpreter file here is SYSTEM.INTERP' instead of SYSTEM.INTERP.)
11) Back up the UCSD PASCAL Bootstrapping disk as suggested in section IV.1.2.3.

The initial interpreter configuration on the UCSD PASCAL Bootstrapping disk is INTERP.CODE, BIOS.C.CODE, INTER.CODE, and TERTBOOT.CODE.

3. Reconfiguring the 6800 Interpreter

The UCSD PASCAL Interpreter disk contains modules that can be linked together to form a UCSD PASCAL interpreter for the 6800 processor (see Appendix C). It must be extracted from the UCSD PASCAL Adaptable disk according to section III.7.

The UCSD PASCAL Interpreter disk contains utility programs and the BIOS, INTER, INTERP, RSP, and TERTBOOT modules:

MODULE NAME	FEATURES
INTERP.CODE	Contains no real number capabilities.
INTERP.FP.CODE	Capable of real number operations but no transcendental functions.
BIOS.CODE	No console, printer, or remote software queuing. Smallest of the BIOS modules.
BIOS.C.CODE	Contains console software queuing.
BIOS.CR.CODE	Contains console and remote software queuing.
BIOS.CRP.CODE	Contains console, remote, and printer software queuing. Largest of the BIOS modules.
INTER.CODE	Interfaces to the SBIOS.
INTER.X.CODE	Interfaces to the Extended SBIOS.
TERTBOOT.CODE	Final bootstrap for the 6800 interpreter.

To create a new UCSD PASCAL Interpreter, perform the following steps:
1) X)ecute the BIOSLINKER program.
2) Pick the module of the INTERP class that has the desired characteristics. The INTERP.CODE module occupies much less run-time memory than the INTERP.FP.CODE module. Type the name of the chosen module as the response to the first linker prompt. Remember to specify the volume name as part of the module name.

If the INTERP.FP.CODE module is chosen, transcendental functions may be used by including 'USES TRANSCEND;' as the second line of a PASCAL program.

WARNING : If the UCSD PASCAL system is informed there is a system clock (through the SETUP program) and the INTERP.CODE module is linked into the running interpreter, an execution error will occur at the end of a compilation when the compiler tries to use floating point numbers to calculate the compile speed.
3) Pick the module of the BIOS class that has the desired characteristics. If an Extended SBIOS is not being used, only the BIOS.CODE and the BIOS.C.CODE modules should be used. Type the name of the chosen module as the response to the second linker prompt.
4) Pick the module of the INTER class that has the desired characteristics. The INTER.CODE module must be used if an SBIOS is being used. The INTER.X.CODE module must be used if an Extended SBIOS is being used. Type the name of the chosen module as the response to the third linker prompt.
5) Type 'TERTBOOT.CODE' as the response to the fourth linker prompt.
6) Type a <carriage return> in response to the next three linker prompts. The linker will terminate with a linked version of the desired configuration in the file SYSTEM.WRK.CODE.
7) X)ecute the SQUISH program.
8) Enter 'SYSTEM.WRK.CODE' as the response to the first prompt.
9) Enter 'NEW.INTERP' as the response to the second prompt. The SQUISH program will create an object file named NEW.INTERP that contains the UCSD PASCAL Interpreter object.
10) Enter the F)iler by typing 'F'. Save a copy of the current interpreter by C)hanging SYSTEM.INTERP to OLD.INTERP. Make the new interpreter current by C)hanging NEW.INTERP to SYSTEM.INTERP. (Note that if the UCSD PASCAL system is using the first half of page 0, the name of the interpreter file here is SYSTEM.INTERP' instead of SYSTEM.INTERP.)
11) Back up the UCSD PASCAL Bootstrapping disk as suggested in section IV.1.2.3.

The initial interpreter configuration on the UCSD PASCAL Bootstrapping disk is INTERP.CODE, BIOS.C.CODE, INTER.CODE, and TERTBOOT.CODE.

RETURN LETTER

Title: UCSD Pascal, User's Guide RCSL No.: 42-i1515

A/S Regnecentralen af 1979/RC Computer A/S maintains a continual effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback, your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

_____

_____

_____

_____

Do you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

How can this manual be improved?

_____

_____

_____

_____

Other comments?

_____

_____

_____

_____

_____

Name: _____ Title: _____

Company: _____

Address: _____

Date: _____

Thank you

42-i 1288

..... Fold here .....

..... Do not tear - Fold here and staple .....

Affix
postage
here

REGNECENTRALEN
af 1979

Information Department
Lautrupbjerg 1
DK-2750 Ballerup
Denmark

FINLAND
RC SCANIPS OY
Espoo, 0 51 35 22

FRANCE
RC COMPUTER S.A.R.L.
Paris, 12 33 53 63

HOLLAND
REGNECENTRALEN (NEDERLAND) B.V.
Gouda, 1820-29455

KUWAIT
KUWAITI DANISH COMPUTER CO. S.A.K.
Safat, 83 01 60

NORWAY
A/S RC DATA
Jessheim, 29 70 220

PHILIPPINES
CARDINAL ELECTRONICS CORPORATION
Metro Manila, 882478

SWEDEN
SCANIPS DATA AB
Stockholm, 8 34 91 55

SWITZERLAND
RC COMPUTER AG
Basel, 61 22 90 71

UNITED KINGDOM
REGNECENTRALEN (UK) LTD.
London, 1 606 3252

UNITED STATES
LOCKHEED ELECTRONICS COMPANY, Inc.
New Jersey, 201 757 1600

WEST GERMANY
RC COMPUTER G.m.b.H.
Frankfurt, 611 66 4006

RC COMPUTER
A/S REGNECENTRALEN af 1979

HEADQUARTERS: LAUTRUPBJERG 1 - DK 2750 BALLERUP - DENMARK
Phone: + 45 2 65 80 00 - Cables: rcbalrc - Telex: 35 214 rcbaldk