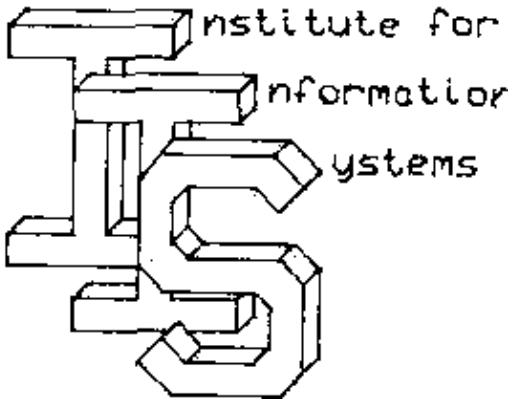


PASCAL System II.0 User's Manual March 1979



Frontice

Table of Contents

Frontice **1**

UCSD (MINI−MICRO COMPUTER) PASCAL VERSION II.0, MARCH 1979 ..... **2**
Disclaimer: 2
Acknowledgements: 2
Documentation Authors: 2
Software Authors: 2
Collected and Edited by: 3
With special thanks to: 3

Table of Contents **4**

Introduction **6**

A.1 Materials Available **7**

A.2.1 The First Time Through **8**

A.2.2. 8080/Z80 With CP/M & 3740 Disks **9**
Booting Pascal 9
MODIFICATiONS TO CP/M 9
Creating a Bootstrap on a Pascal Disk 10

A.3 Differences Among Implementations **12**

A.4 Changes Made In Recent Releases **13**
Summary Of Differences Between UCSD Pascal Releases I.5 And II.0 13
(1.1) Operating System 13
(1.2) File Handler 13
Editors (Sections 1.3 And 1.4) 13
Purpose 14
P−Code Changes 15
Operators with new or changed functional behavior: 15

1.1. Introduction and Overview **16**
1.1.1. The UCsD Pascal System: An Overview 16
1.1.2 Outermost Level Commnds: An Overview 17
E(dit 17
F(iler 17
C(omp 17
R(un 18
X(ecute 18
A(ssem 18
D(ebug 18
L(ink 18
1.1.3. Utility Programs 18
1.1.4. An Introduction To The UCSD Pascal System 19

Table of Contents

1.2. File Handler **22**
1.2.1 Files 22
1.2.2 Volumes 22
1.2.3. The ‘Workfile’ 23
1.2.4. File Specification 23
1.2.5. Commands and Use 25
1.2.5.1. G(et 25
1.2.5.2. S(ave 26
1.2.5.3. N(ew 27
1.2.5.4. Q(uit 27
1.2.5.5. W(hat 27
1.2.5.6. V(olumes 27
1.2.5.7. L(dir 28
1.2.5.8. E(xtended list 29
1.2.5.9. C(hange 30
1.2.5.10. R(emove 31
1.2.5.11. T(ransfer 32
1.2.5.12. D(ate 34
1.2.5.13. P(refix 35
1.2.5.14. B(ad blocks 35
1.2.5.15. eX(amine 35
1.2.5.16. K(runch 36
1.2.5.17. M(ake 37
1.2.5.18. Z(ero 37

1.3. Screen Oriented Editor **39**
1.3.2. Getting Started 40
1.3.3. Detailed Description of Commands 43

1.4. Yet Another Line Oriented Editor **66**

1.5. Debugger **74**

1.6. Pascal Compiler **75**
1.6.1. Compile Time Options 75

1.7. Basic Compiler **81**

1.8. Linker **86**

1.9. Assembler **89**

1. The UCSD Pascal System **101**

Intrinsics Parameters **102**

Table of Contents

2.1.1. String Intrinsics **104**

2.1.2. Input and Output Intrinsics **107**

2.1.3. Miscellaneous Routines **111**

2.1.5. Character Array Manipulation **114**

2.1. Intrinsics **116**

2.2. Differences Between UCSD Pascal and Standard Pascal **117**

2. The UCSD Pascal Language **137**

3.1. Drawline **138**

3.2. File Formats **140**

3.3.2. Linkage to Externally Compiled and Assemble Routines **141**

3.3.3. Long Integers **150**

3.3. Procedure Notes **152**

3.4. Pseudo Machine Architecture **153**

3.4.1. Hardware (Emulation) 153
3.4.2. Operating System/p−Machine Communication − SYSCOM 153
3.4.3. Exception Handling − XEQERR 154
3.4.4. Operand Formats 154
3.4.5. Instruction Set Format 157
3.4.6. English Instruction Set Description 157
5.A. Variable Fetching, Indexing, Storing, and Transfering 158
5.A.1. One Word Loads and Stores 158
5.A.1.b Local One Word Loads and Store 158
5.A.1.c. Global One Word Loads and Store 158
5.A.2. Multiple Word Loads and Stores (Sets and Reals) 159
5.A.3. Byte Arrays 159
5.A.4. Strings 160
5.A.5. Record and Array Indexing and Assignment 160
5.A.6. Dynamic Variable Allocation and De−allocation 161
5.B. Top of Stack Arithmetic and Comparisons 161
5.B.1. Logical 161
5.B.2. Integer 162
5.B.3. Reals 162
5.B.4. Sets 163
5.B.5. Strings 164
5.B.6. Byte Arrays 164
5.B.7. Array and Record Comparisons 165

Table of Contents

3.4. Pseudo Machine Architecture
5.C. Jumps....165
5.D. Procedure and Function Calls and Returns....165
5.E. System Programs Support Procedures....167
Byte Array Procedures....167
Compiler Procedures....167
Debugger....168
Miscellaneous....168

3.5. Introduction to the Pascal Pseudo−Machine....169

3.6. Byte Swapped....180

3. Implementors' Guides....182

4.2. Librarian....183
Example....183
Note....184

4.3. Setup — System Reconfiguration....185
4.3.1. Miscellaneous Information....185
4.3.2. General Terminal Information....186
4.3.3. Control Key Information....187
4.3.4. Video Screen Control Characters....188
4.3.5. Quit....189
4.3.6. Quick Reference Summary....189

4.4. Bootstrap Copier....191

4.5. Patch/Dump....192

4.6. RT11 to Pascal Conversion Kit....195

4.7. GotoXY Binder....196

4.8. Duplicate Directory....199
COPYDUPDIR....199
MARKDUPDIR....199

4.9. P−Code Disassembler....200
4.9.1. Disassembly....200
4.9.2. Data Segment Reference Statistics....201
4.9.3. Opcode, Procedure Call, and Jump Statistics....202

4.10. Library Map....203
Example....203

Table of Contents

4. Utility Programs **205**

5. Tables **206**

A. Index **236**

Frontice

UCSD (MINI–MICRO COMPUTER) PASCAL VERSION II.0, MARCH 1979



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

Release serial rumber: 4135

Copyright © 1978 Regents of the University of California, San Diego Campus. This software, its source, object, and all other forms, is the property of the Institute for Information Systens and may be used or copied by others only with written authorization from the Institute for Information Systens. All rights reserved.

Disclaimer:



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

Acknowledgements:

The work described in these notes has been supported significantly by the following organizations: United States Navy Personnel Research and Developnent Center, Sperry Univac Minicomputer Operations, EDUCOM, Digital Equipment Corporation, Processor Technology Inc., Springer–Verlag, Terak Corporation, General Automation Corporation, The UCSD Computer Center, grants fran the University of California Instructional Improvement Program, Tektronix Corporation, Micropolis Inc., Phillips Research Labs, Lawrence Livermore Labs, Pascal Computing.

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

Kenneth L. Bowles

Documentation Authors:

Gillian M. Ackland, S. Dale Ander, Lucia A. Bennett, Raymond S. Causey, Charles "Chip" Chapin, 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.

Software Authors:

Lucia A. Bennett, Marc Bernard, J. Greg Davidson, Barry Dernchak, Gary J. Dismukes, William P. Franks, Julie E. Erwin, Robert J. Hofkin, Albert A. Hoffman, Richard S. Kaufmann, Peter A. Lawrence, Joel J. McCornack, Robert A. Nance, Mark D. Overgaard, David A. Reisrier, Keith A. Shillington, David M. Steinore, Roger T. Sumner, Steven S. Thompson, David B. Wollner.

Collected and Edited by:

Keith Allan Siillington and Gillian M. Ackland.

With special thanks to:

Tracy Barrett and the entire support staff.



Table of Contents

Version II.0, March 1979

SECTION

0 ADDENDA, ERRATA AND NOTES

  1 NOTES ON OTHER MATERIALS AVAILABLE
  2 BRINGING UP THE PASCAL SYSTEM
    1 ON PDP-11
    2 ON 8080/Z80 SYSTEM WITH CP/M AND 37L10 DISKS
    3 DIFFERENCES AMONG IMPLEMENTATIONS FOR DIFFERENT PROCESSORS.
    4 CHANGES IN RECENT RELEASES

1 THE UCSD PASCAL SYSTEM

  1 INTRODUCTION AND OVERVIEW
  2 FILE HANDLER
  3 SCREEN ORIENTED EDITOR
    1 INTRODUCTION
    2 GETTING STARTED
    3 DETAILED DESCRIPTION OF COMMANDS
    4 REFERENCE
    5 EXPERIMENTAL LARGE FILE VERSION (L2)
  4 YET ANOTHER LINE ORIENTED EDITOR - YALOE
  5 DEBUGGER
  6 PASCAL COMPILER
  7 BASIC COMPILER
  8 LINKER
  9 ASSEMBLER

2 THE UCSD PASCAL LANGUAGE

  1 INTRINSICS
    1 STRING
    2 INPUT/OUTPUT
    3 MISCELLANEOUS
    4
    5 CHARACTER ARRAY MANIPULATION
  2 DIFFERENCES BETWEEN UCSD'S PASCAL AND STANDARD PASCAL
    1 CASE STATEMENTS
    2 COMMENTS
    3 DYNAMIC MEMORY ALLOCATION
    4 EOF
    5 EOLN
    6 FILES
    7 GOTO AND EXIT STATEMENTS
    3 PACKED VARIABLES
    9 PARAMETRIC PROCEDURES AND FUNCTIONS

   10 PROGRAM HEADINGS
   11 READ AND READLN
   12 RESET
   13 REWRITE
   14 SEGMENT PROCEDURES

15 SETS
15 STRINGS
17 WRITE AND WRITELN
13 IMPLEMENTATION SIZE LIMITATIONS
19 EXTENDED COMPARISONS
20 LONG INTEGERS
21 UNITS
22 TABLE OF UCSD INTRINSICS

3 IMPLEMENTORS' GUIDES

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

4 UTILITY PROGRAMS

1
2 LIBRARIAN
3 SETUP - SYSTEM RECONFIGURATION
4 BOOTSTRAP COPIER
5 PATCH/DUMP
6 RT11 TO PASCAL CONVERSION KIT
7 GOTOXY PROCEDURE BINDER
8 DUPLICATE DIRECTORY
9 P-CODE DISASSEMBLER
10 LIBRARY MAP

5 TABLES
1 EXECUTION ERRORS
2 IORESULTS
3 UNITNUMBERS
4 RESERVED WORD LIST
5 SYNTAX ERRORS
6 ASSEMBLER SYNTAX ERRORS
7 AMERICAN STANDARD CODE for INFORMATION INTERCHANGE
8 P-MACHINE OPCODES
9 UCSD PASCAL SYNTAX DIAGRAMS

A INDEX

Introduction

A.1 Materials Available



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

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



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

A.2.1 The First Time Through

Version II.0, March 1979



Welcome to UCSD Pascal. If you put the Advanced System 1 diskette in your booting drive, went through your normal boot−strapping procedure, and were greeted in a similar fashion, you do not need to read this section.

If this is not the case then here are a few of the problems we encountered with I.4, and I.5 coming up in strange and foreign lands:

1. Some revisions of the LSI−11 refuse to boot with the clock running. If you have a switchable clock, turn it off to bootstrap; if and when the system greets you with the welcome message and the date, turn the clock back on.
2. You do not have enough memory. The minimum requirement for memory is 24K 16−bit words.
3. You have a system configured for RK−05 hard−disk and you have an unformatted disk on line. The system will hang waiting for a reply from the disk which cannot be generated if the disk is unformatted. Take the disk off−line and try again.
4. You have a system configured for BK and RX and the RX or BK is not present. Both must be present at the standard DEC UNIBUS or QBUS addresses for these devices.
5. We haven't encountered your problem before. Call: The number listed on the front page of this docuTlent.



A.2.2. 8080/Z80 With CP/M & 3740 Disks

Version I.5, September 1978

The CP/M Implementation of UCSD Pascal

Booting Pascal



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

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

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

MODIFICATiONS TO CP/M

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

1. If there is no disk in a drive and an access is made from that disk, the driver should not wait to perform that access until a disk is inserted, as the Pascal system often attempts to read from empty drives when searching for a particular disk. Instead, simply return a 1 to indicate a bad I/O operation.
2. If you have a keyboard interrupt handler, it should recognize the character [cntrl−f] as a “flush−output” toggle and signal the character−out routine to gobble any characters until signaled again. When it receives another [cntrl−f] the keyboard handler should signal the output handler causing the output handler to resune outputting characters sent to it.



The keyboard interrupt handler should also recognize the character [cntrl−s] as a “stop output” toggle and wait until it receives another [cntrl−s] before allowing program execution to continue.

If your keyboard has no alphalock, the input driver can use any character not used for some other purpose as an alphalock toggle. [Cntrl−p], [return], [cntrl−i], [cntrl−s], [antrl−f], [cntrl−c] or any character in `SYSCOM.CRTINFO` should be excluded from consideration. We suggest [cntrl−a].

Pascal expects the tab character ([cntrl−i]) to cause the terminal cursor to advance to the nearest eight column. If the terminal does not do this itself, then the driver in the BIOS should.

Creating a Bootstrap on a Pascal Disk

Note: These instructions are for a standard BIOS with 512−byte blocks. For instructions for a non−standard BIOS, reference file `READ.ME` on the CP/M disk in the distribution packet.

On the CP/M disk are two programs, `PGEN.COM` and `PINIT.ASM`. The program `PGEN.COM` is a program used to write out a buffer (which will be filled by boot code and BIOS) to track 0. `PINIT.ASM` is the boot code that reads `SYSTEM.MICRO` from a Pascal disk, loads the BIOS into the correct place, and starts the interpreter's boot routine.

You must create a file `PPOOT.HEX`, which will require a slight modification of your current BOOT program. `PBOOT` will reside on track 0, sector 1 and, when executed, will load track 0, sectors 2 thru 13 into memory starting at location (MSIZE−148)*1024 + OBAOOH, and jump to that location.

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

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

```
A>ddt pgen.com          ; load PGEN.COM into memory. PBOOT, PINIT, and BIOS will be overlayed
                        into PGEN's data area, after which a memory image will be saved.
DDT VERS 1.3
NEXT PC
0400 0100
-ipboot48.hex           ; set PBOOT48.HEX as input file
ipinit48.hex            ; set &lsquo;PINIT48.HEX&rsquo; as input file
-h980 BA00              ; PINIT starts at location BA00H in a 48K system (in general
C380 14F80              (MSIZE−48)*1024 + BA00H), and we want it at location 980H
-r4f80                  ; read it in
NEXT PC
OA7d BAC0
-ibios48.hex            ; and lastly read BIOS into location D80H
-hd80 be00
C380 4F80
-r4f80
NEXT PC
0F76 0000
-[cntrl-c]              ; leave DDT...
A> save 16
pgen48.com              ; ...and save the program.
A> pgen48               ; sample execution of the program...
PGEN VI.0
PUT BOOTER?(Y/N)y
WRITING BOOTER TO DRIVE A, TYPE RETURN
AGAIN? (Y/N) n
GET BOOTER? (Y/N) n     ; put a Pascal disk (preferably a copy of the master) in drive A before hitting
                        [return].
```

```
REBOOTING CP/M, TYPE ; put the CP/M disk back in drive A before hitting [return].
RETURN
A>
```

A.3 Differences Among Implementations

Version II.0, March 1979

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



1. The definition of *div* is different (thereby changing the values returned by *mod*):

   a *div* b = floor(a / b)
   a *mod* b = a − b * (a *div* b)
2. The I/O drivers are all written for synchronous operation. This means that [break] has no effect. [Cntrl−s] and [cntrl−f] will not perform as described unless you have a keyboard interrupt handler, and this handler is modified as specified below in *Modifications to CPM*.

   This also means that UNITBUSY, UNITCLEAR, and UNITWAIT are meaningless. (In the future it may be possible to use the UNITBUSY and UNITCLEAR operations on the keyboard, but this is currently infeasible.)
3. The interpreter is called `SYSTEM.MICRO` instead of `SYSTEM.PDP−11`.
4. The CP/M implementations have bootstraps that are not accessible to Pascal, hence the program BOOTER will not work. See the appropriate section of this document for instructions on copying and/or creating a bootstrap.
5. There are no long integer functions available with the Z80/8080 system. They will be available in a later release.

A.4 Changes Made In Recent Releases

Version II.0, March 1979

Summary Of Differences Between UCSD Pascal Releases I.5 And II.0



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

(1.1) Operating System

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

(1.2) File Handler

Substantial modifications have been made in the syntax of user responses to filer prompts. The symbol "$" means ‘same name’. This symbol may be used on the right–hand side of a transfer command expression. If the filer detects as some time that two volumes on line have the same name, the warning message:

> Warning: units *N* and *M* have the same name

will appear beneath the prompt line. This message will remain until some action is taken to convince the filer that this is no longer true.

R(emove command prompts for verification always.

K(runch command allows space to be opened anywhere on the disk.

The bad block scan allows for scanning of any nunber of blocks.

Editors (Sections 1.3 And 1.4)



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

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

XYADRESSING
: go directly to a given row and coli.znn on the screen

NDFS
: non–destructive forward space (the inverse of back–space)

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

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

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

Proper use of EDITOR requires that the system disk be left online while editing.

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

Purpose



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

1. the volume ‘`REMOTE:`’ has been split into two volumes — ‘`REMIN:`’ and ‘`REMOUT:`’. Programs using ‘`REMOTE:`’ will have to be modified to reflect this change.
2. User programs (rather sophisticated ones presumably) which called the system procedure FBLOCKIO must be changed to accommodate an additional parameter. Uses of BLOCKREAD and BLOCKWRITE are unaffected.

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

1. The Pascal compiler will run in swapping mode automatically (without the `(*$S+*)` directive) if it determines that no useful program could be compiled without swapping.
2. Codefiles for system programs are no longer required to reside on the system disk. The operating system will, at initialize time, (looking first on ‘`*`’ volume) scan on−line volumes for the files: `SYSTEM.EDITOR`, `SYSTEM.FILER`, `SYSTEM.COMPILER`, `SYSTEM.LINKER`, and `SYSTEM.ASSMBLER`, and remember where they were found. Furthermore, if one of these files cannot be found when it is actually invoked by the user, the system will look again at that time.

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

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

strings and packed arrays, and byte−sex difficulties.

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

P−Code Changes



Decimal value	I.5	II.0	Description
157	S2p	unused	
166	LCA	LSA	Load String Address
167	LDO	unused	
169	MVB	LDO	Load Global
208	S1P	LPA	Load Packed array Address
209	IXB	unused	
210	BYT	unused	

Operators with new or changed functional behavior:

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

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

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

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

1.1. Introduction and Overview

Version II.0, February 1979



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

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

Tne following is a tutorial book on PASCAL:

> Kenneth L. Bowles,
> (Microcomputer) Problem Solving Using PASCAL
> Springer−Verlag, New York, © 1977

We suggest the following book as a PASCAL reference guide:

> Kathleen Jensen and Niklaus Wirth,
> PASCAL User Manual and Report
> Springer−Verlag, New York, © 1975

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

1.1.1. The UCsD Pascal System: An Overview

The structure of the UCSD Pascal system is best conceptualized in terms of the “tree−like” structure diagram figure 0.1.



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

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

By typing “F” the user will “descend” a level within the structure diagram into a

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

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

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

1.1.2 Outermost Level Commnds: An Overview

E(dit



Typing “E” while at the outermost conmand level of the system causes the editor program to be brought into memory from disk. The user may, while in the editor, alter text inside his workfile or any textfile. See Section 1.3 for details. The workfile text (if present) is read into the editor buffer, otherwise the Editor prompts for a file.

F(iler

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

C(omp

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

R(un

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

X(ecute

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

A(ssem



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

D(ebug

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

L(ink

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

1.1.3. Utility Programs

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

1.1.4. An Introduction To The UCSD Pascal System

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



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

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

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

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

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

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

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



SYSTEM.PDP−11
SYSTEM.MICRO
SYSTEM.PASCAL
SYSTEM.SYNTAX
SYSTEM.ASSMBLER
SYSTEM.COMPILER
SYSTEM.EDITOR
SYSTEM.FILER
SYSTEM.LINKER

SYSTEM.STARTUP
SYSTEM.SWAPDISK
SYSTEM.CHARSET
SYSTEM.LIBRARY
SYSTEM.WORK.TEXT
SYSTEM.WRK.CODE
SYSTEM.WRK.INFO
SYSTEM.LST.TEXT

In most cases these files contain the system segment of the name they carry. That is to say that the EDITOR, FILER, LINKER, COMPILER, ASSEMBLER are the files that are invoked by the main level. of the system when ‘E’, ‘F’, *etc* is typed. Some of the files are machine specific. PDP−11 and MICRO are the files which contain the interpreters for the particular machine being used. CHARSET is a file which appears on disks meant for TERAK computers only and contains the definition for the soft character set, and the data for the Triton logo prompt. LIBRARY is a file containing separately assembled or compiled

routines for use by the Linker in producing executable code files. PASCAL contains the operating system, and the Debugger. SWAPDISK is a file used by some of the system segments during open/close operations on files if a memory shortage exists. It is a 2048 byte file which gets a portion of memory swapped to it when a directory needs to be read into core. When the directory work is complete, the memory is restored to its original state. STARTUP is a file which can be created at the user's option. If it exists on a disk, the operating system considers it a runnable code−file, and executes it at initialize time. This allows the user to have a program that runs before the main command prompt comes up, and will run anytime the I(nitialize command is typed. WRK.TEXT and WRK.CCDE are the current work−file after some action has occurred to the work−file. They appear after having done some text editing on a work−file (SYSTEM.WRK.TEXT) or compiling a work−file (SYSTEM.WRK.CODE). To change the editor which is invoked by “E”, one simply names the codefile which is to respond to the ‘E(dit’ command SYSTEM.EDITOR. This is true for all system segments which have named files associated with their command.

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

1.2. File Handler

Version II.0, February 1979

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 File handler, 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 charged, 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 respnd with at least one file specification. The diagram below illustrates the syntax of file specification.

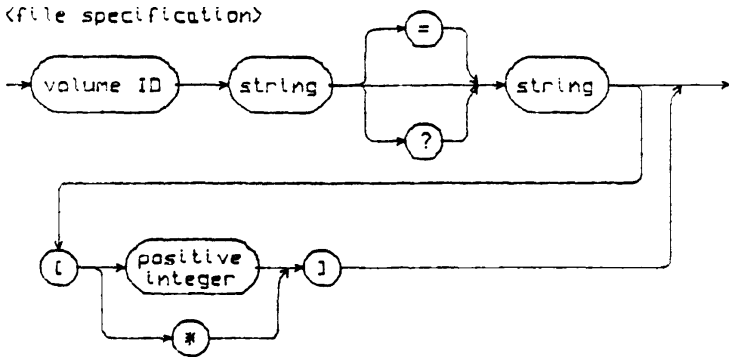


Figure 2

Volume i.d. syntax can be expanded thusly:

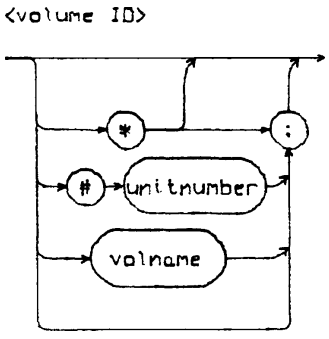


Figure 3

Volume names for block−structured volumes can be 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 Systen.

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 volune, so typing ‘=’ or ‘?’ alone causes the Filer to perform the appropriate action on every file in the directory.

Given an example directory for volume MYDISK:

```
NAUGHTYBITS      6  23-Jun-54
MOLD.TEXT        4  29-Jun-54
USELESS.CODE    10  19-May-54
MOLD.CODE        4  29-Jun-54
NEVERMORE.TEXT  12   5-Apr-54
G00NS            5  10-Sep-52
```

Example:

Prompt: `Remove what file?`
Response: Typing ‘N=’ *generates the message:*

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

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



Typing ‘N?’ *generates the message:*
`Remove NAUGHTYBITS: ?`
After the user types a response, the Filer asks:
`Remove NEVERMORE.TEXT: ?`

Example:

Prompt: `Dir listing of what vol ?`
Response: Typing ‘=TEXT’ *causes the Filer to list*

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

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

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

1.2.5. Commands and Use

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

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

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

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

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

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



For each cammand 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 cammand 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. Comands 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 prampt for the second member of the pair. If an error is detected in the list, the remainder of the list will be flushed.

1.2.5.1. G(et

Loads the designated file into the workfile.

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

Given the example directory:

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

Example:

Prompt: `Get what file?`
Response: `STASIS`
The Filer responds with the message

```
'Text & Code file loaded'
```

since both text and code file exist. Had the user typed ‘STASIS.TEXT’ or ‘STASIS.CODE’, the result would have been the same — both text and code versions would have been loaded. In the event that only one of the versions exists, as in the case of `A.OUT`, then that version would be loaded, regardless of whether text or code was requested. Typing ‘`ABSURD.TEXT`’ in response to the prompt would generate the message: ‘`Code file loaded`’. Working with the file may cause the files `SYSTEM.WRK.`*`xxxx`* to be created, as part of the workfile. These files will go away when the S(ave comand 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 1.2.24 ‘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 `.TETX` 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 filenane 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 exanpie, 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 a file called *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 nane and state (saved or not) of the workfile.

1.2.5.6. V(olumes

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

A typical display might be:

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

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

1.2.5.7. L(dir

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

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

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

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



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

```
Dir listing of what vol?

MYDISK:
FILERDOC2.TEXT  28  1-Sep-78
ABSURD.CODE     18  1-Sep-78
HYTYPER.C0DE    12  1-Sep-78
STASIS.TEXT      8  1-Sep-78
LETTER1.TEXT    18  1-Sep-78
ASSEMDOC.TEXT   20  1-Sep-78
FILERDOC1.TEXT  24  1-Sep-78
STASIS.CODE      6  1-Sep-78
10/10 files <listed/in-dir>, 144 blocks used, 350 unused, 200 in largest
```

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

Example:

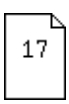
L(dir transaction involving wildcards:

Prompt: `Dir listing of what vol ?`

User response: `#4:FIL=TEXT`
generates the following display:

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

Example:



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

> *Prompt:* `Dir listing of what vol ?`
> *User response:* `*FIL=TEXT,FRINTER:` causes
>
> ```
> MYDISK:
> FILERDOC2.TEXT   28  1-Sep-78
> FILERDOCI.TEXT  214  1-Sep-78
> 2/10 files <listed/in-dir>, 62 blocks used, 364 unused, 200 in largest
> ```

to be written to the Printer.

Example:

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

> *Prompt:* `Dir listing of what vol ?`
> *User response:* `#4:FIL=TEXT,#5:TRASH` creates the file `TRASH` on the volume associated with unit 5. The `TRASH` file would contain:
>
> ```
> MYDISK:
> FILERDOC2.TEXT   28  1-Sep-78
> FILERDOC1.TEXT  214  1-Sep-78
> 2/10 files <listed/in-dir>, 62 blocks used, 364 unused, 200 in largest
> ```

1.2.5.8. E(xtended list

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

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

```
MYDISK:
FILERDOC2.TEXT  28  1-Sep-78    6  512  Textfile
ABSURD.CODE     18  1-Sep-78   34  512  Codefile
<UNUSED>        10              52
ABSURD          14  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
smsis.cocc       6  1-Sep-78  348  512  Codefile
<UNUSED>       154             354
10/10 files <listed/in-dir>, 138 blocks used, 36~4 unused, 200 in largest
```



1.2.5.9. C(hange

Changes file or volume name.

This canmand 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 coma (‘,’). 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 ecample file `F5.TEXT`, residing on the volune 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 comand does not affect the filekind. In the above case, `H00HAH` would still be a text file. However, since the G(et canrnand 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:* `Charge what file?`
> *User Response:* `NOTSANE:,WRKDISK:`
>
> ```
> NOTSANE: --> WRKDISK
> ```



1.2.5.10. R(emove

Remove file entries from the directory.

This canmand 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 coma 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[2O]`

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?`

Responding ‘`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 comand 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)`

Responding ‘`Y`’ initiates the K(runch. Typing an ‘`N`’ will cause the prompt:

Prompt: `Starting at block # ?`

Respond with the block nunber 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> ?`

Responding with a ‘`Y`’ generates

Prompt: `Duplicate dir ?`

Responding with a ‘Y’ 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)`

Responding with an ‘N’ generates

Prompt: `# of blocks on the disk ?`

Respond by typing the number of blocks desired. The table following this section gives the correct number of blocks for several types of disks.

A response of ‘Y’ generates

Prompt: `New vol name ?`
Response: User types any valid volume name.
Prompt: `<new volume name> correct ?`

Responding with ‘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	Number of blocks
Terak	Single−density soft−sectored 8" floppy	493
Northwest Micro	Double−density soft−sectored 8" floppy	1101
Zilog	Single−density hard−sectored 8" floppy	607
North Star	Single−density hard−sectored 5¼" floppy	167
DEC	RKO5 / per volume	4871

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

1.3. Screen Oriented Editor

Version II.0, February 1979



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

The Concept of a ‘Window’ into the File

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

The Cursor

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

The Concept of a Prompt Line

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

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

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.



1.3.2. Getting Started

Entering the Workfile and Getting a Program

On entering the Editor

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

appears.

There are three ways to answer this question :

1. With a name, for example “*string1* <ret>”. The file named *string1* will now be retrieved. The file *string1* could contain a program, also called *string1*, as in Fig. 2.1. after typing the name, a copy of the text of the first part of the file appears on the screen.

   Figure 2.1

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

2. With a <return>. This implies that a new file is to be started. The only thing visible on the screen after doing this is tha 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 4.3 for more information on setting control keys.

The cursor does not like to be outside of the text of the program. For example, after the “N” in

“BEGIN” in Fig. 2.2 , push the <right−arrow> and the cursor moves to the “W” in “WRITE”. Similarly at the “W” in “WRITE('TOO WISE ');”, use <left−arrow> to move to after the “N” in “BEGIN”.

Figure 2.2

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

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

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

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

Figure 2.3

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

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

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} t<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 comnands, in the next section, explains in greater detail the options available at >Quit.

1.3.3. Detailed Description of Commands

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 ‘I’ 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

The 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, the 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



The 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 jumnp 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

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

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

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

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

Auto−Indent

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

```
&ldquo;ONCE UPON A TIME THERE− WERE&rdquo;.
```

(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 at 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 REPIACE 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(opy 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 in 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 twp 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(nvironrnent mode displays the current target and substitution strings.



FIND

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

Figure 3.7

>Find[1J: L(it <target> :>

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

TI~ 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 rii.~nber "n" is the <repeat−factor> and is shown on the prompt line in the brackets "tJ".

Example 1: In the STRING1 program with the cursor at the first "F" 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{1J: L)it <target> =>`WRITE'

After typing the last quote mark the cursor j~.znps to immediately after the "E" in the first WRITE. Example 2: In the STRING 1 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> =>IWRITELN/

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

Page ~45



Figure 3.8

PROGRP1M STRING1;

BEGIN

WRITE('TOO WISE 0; WRITE('YOU ARE');

WRITEL.N(',');

(`CURSOR

FINISHES

IN

THIS

LINE')

WRITELN('TOO WISE 0;

WRITELN('yOtJ 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.

REP LA~ REPLACE REPLACE mode is reached by typing "R" while at the Edit level. On enter irig 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

>ReplaceCL~]: L(it V(fy <targ> <sub> =>

)Replace[L~: T(ok V(fy <targ) <sub> :>

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

>ReplaceClJ: L)it V)fy <targ> <sub> :>L/QX//YZ/

This cairnand will change: "VAR SIZEQX:INTEGER;" to "VAR SIZEYZ:INTEGER;". Literal mode is necessary because the string ~X is not a token but is part of the token S~EQX.

Example 2: In Token mode REPLACE ignores spaces between toKens when finding patterns to replace. For example, using the lines on t~e left hand side of Figure 3.10 and typing: "2RT/(',')!.LN." The prompt line should appear as:

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

Page ~46



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

Figure 3.10

WRITE(',

0;

~RITELN;

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 replac~nent 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 nunber of times an occurrence is found, riot the nunber of times you actually type "R1'. Use "I" as a <repeat− factor> in order to examine every occurrence of the target string. If the Editor can not find the target string the ntznber of times specified, the prompt:

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

appears.

FORMATTING C~4MANDS

ADJUST

ADJUST node 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.

Page L~7



In order to adjust a whole sequence of lines, adjust one line, then use <up−arrow> (<thwn−arrow>) cairiarids and the line above (below) will be automatically adjusted by the sane amount. Repeat−factors are valid when used before any of the <arrow> c~rnands while in ADJUST mode, ir~lt~ing `I'. 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. Si.milarly typing "R" right−justifies to the set margin and typing "C" will cause the line to be centered bet~en the set margins. Typing <up−arrow> (or <~wn−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 caitnand, 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 c~nand does not appear on the ">Edit:" line.

There are t~ 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 sane 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

Page 48



Figure 3.13

This q~rter, the equi~nent 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 equi~nent 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 cirsor 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.

C~t+1AND 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 c~iinand character as though it were a blank line, that is, it will consider that line to terminate (begin) the paragraph.

~rning: tb not use the MARGIN command when in a line beginning with the Command character.

MISCELLANEOUS CCWANDS

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(nvironmerit <esc>

Page 149



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. ~ce set, it is possible to jump to these markers using the M(arker option in the JL1~1F mode. When in the SET mode, type "M" for M(arker and the following pr~npt 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 sane letter are considered to be different characters. The marker will be entered at the position of the ctrsor 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 "3'l", the pranpt:

Figure 3.15

Marker ov

fiw.

.

Which one

to replace.

0) narnel

1) nane2

9)namelO

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 foruse 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(nvironrnerit:

The Editor enables the user to set the environment which the user deter~nines 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.

Page 50



Fi~~ure 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(a~znand oh T(oken def True 71136 bytes used, 12~O available

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

Date Created: ZI_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 conmand. 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.

Page 51



C(a~and ch: ~ie Comma~d 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 exanpie typing "C" , "i" will change the Cc~rnand character to "s". This change will be reflected in the prompt.

T(oken def:

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

VER IFY

The VERIFY command is executed by typing "`1" ~i1e at the Edit level. The status of the Editor is verified by redisplaying the screen. Thc Editor attempts to adjust the window so that the cursor is at the center of the scree~.

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

Figure 3.17

)Qui. t: U(pdate the workfile and leave E(xit without updating R(eturn to the editor without updating ~(rite to a file nane and return

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

U(pdate:

This causes the Editor to write the file just m~iuied into the ~rkfile and store it as SYST~'1.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.TEXr as text file.

Page 52



E(xit:

This causes the Editor to leave without making any changes in SYST~1.WRK. TEXT. This means that any modifications made since entering the Editor are not recorded in the permanent t.~rkfile. 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 ~then "Q" was typed. Usually this c~nand is used after unintentionally typing "Q".

W(rite:

This option puts up a further prompt:

Figure 3.18

>Quit:

Name of output file (<or> to return) −−>

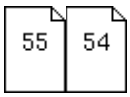
The modified file may now be written to any file nane. If it is written to the name of an existing file, the modified file will replace the old file. This ca~tnand can be aborted by typing <return> instead of a file name and return will be to the Editor. After the file has been written to disk, the Editor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

Page 53



********** ********** * REFERENCE SECTION * * Section 1 . 3. ~ * ********** **********

<down−arrow> moves <repeat−factor> lines down <up−arrow> " lines up <right−arrow> " " spaces right <left−arrow> It 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

tt<?t It ~ change direction to backward ~ ~ "+" change direction to forward `I.." moves to the beginning of what was just found/replacedlinserted/ exchanged

<repeat−factor> is any ntxnber typed before a canmand. 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 sane 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 canmands 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. ~tS't use same string as before. I(nsert: Inserts text. Can use <backspace> and <del> to reject part of your insertion.

J(u.znp: Jumps to the beginning, end or previously set marker.

~4(argin: ~justs anything between t~ blank lines to the margins which have been set. Canmand characters protect text fran being margined. Invalidates the copy buffer.

P(age: Moves the cursor one page in direction. Repeat−factors are valid, direction is applied.

Q(uit: Leaves the editor. You may U)pdate, E)xit, W)rite, or R)eturn.

Page 55



R(eplace: Operates in L(iteral or T(oken mode. Replaces the <targ> string with the <subs> string. V(erify option asks you to verify before it replaces. "S" option uses the Sane string as before. Repeat−factors replace the target several times. Direction is valid.

S(et: Sets M(arkers by assigning a string name to theii. Sets E(nvironxnent for A(uto−indent, F(illing, margins, T(oken, arid C(cvmand 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 ~zition of the last thing found/replaced/inserted as an anchor and deletes everything between the anchor and the current cursor position.

Page 56



********** ********** * L2 EDITOR * * Section 1.3.5 * ********** ******,**********

Version 11.0 February 1979

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

Entering the Workfile and Getting a Progran

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

ERROR: Not erough room for backup!

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

The sane prompt line is displayed; see section 1.3.2.

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

Figure 5.1

Copying to filenane.back. > Ed it Reading....

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

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

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

Leaving the Editor and Updating the ~orkfile

Page 57



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

Figure 5.2

>~it:

U(pdate the ~rkfile

and

leave

E(xit (but

~rkfile

not

updated)

.

R(eturn to

the Editor without doing anything.

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

U(pdate:

This ~rks in the sane manner, however additional information is supplied indicating the name of file updated and the length.

When a new file is created, the following appears:

Figure 5.3

Writing. * The ~rkfile, *SYST~1.WRK.TEXT, is n blocks long.

When an existing file has information now given:

Figure 5*J4

been

used, this example shows the extra

.

Writing. *

The ~rkfile, *X:F1.TEXT, is LU4 blocks The backup file is X:F1.BACK.

long.

The newiy edited file is referred to as .TEXT, while the .BAC~ file

contains the original file with no modifications.

E (x it:

This causes the Editor to return to the ccmnand level w~th~it making any changes in the workfile. No .BACK file is made and the existing .BACK is r~noved. For example, if F1.TEXT is the file being used, then a copy F1.BACK will be made on entering the editcr and on leaving by using the E option, F1.BACK will be r~noved and only F1.TEXT will r~nain. However, since F1.TEXT is a copy of the origir~al, it will be in different place in the directory.

R(eturn:

This is the sane. See section 1.3.3. Page 58



MOVING CG~1MANDS

JUMP

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

Typing "M" causes the Editor to display:

Jump to what marker?

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

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

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

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

NEXT

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

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

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



Figure 5.5 SYMBOLIC FILE

left st~k

I

right stack

1

Back~ardz

1

BUFFER

Forward

1

Start

1

1

End

PAGE

See section 1.3.3.

EQJALS

See section 1.3.3.

TEXT CHAN~NG C~t41ANDS

INSERT

See section 1.3.3.

DELETE

See section 1.3.3. ZAP

See section 1.3.3.

COPY

See section 1.3.3.

EXCHANGE

See section 1.3.3.

FIND

Read sectio 1.3.3. The Editor will display: Finding and if the pattern is not in the buffer: End of buffer encountered. Get more frctn disk? (Y/N)

Page 60



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

REPLACE

See section 1.3.3.

FO~4ATTING C~+1ANDS

ADJUST

See section 1.3.3.

MARGIN

See section 1.3.3.

MISCELLANEOUS CO~1MANDS

SET

See section 1.3.3. The same pranpt line is displayed.

M(arker:

Read section 1. 3. 3. The names of the markers can be seen by typing "SE" for Set Environment while at the Edit level. To set the marker, type "3.1". In the event that 20 markers have already been set, this will be indicated by:

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

E(nvironrnent:

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

Figure 5.5

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

Page 61

11582 bytes used. 275Z~ available.

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

Markers:

<P1 P2 >P3

Creates August 15, 1978: Last u~ated August 15, 1978 (Revision 1).

By typing the appropriate letter, any or all of the options can changed. See section 1.3.3. The arrow before the marker name indicates the relative position of the marker in the file to the buffer. ~b arrow indicates that the marker is in the current buffer.

It is now possible to vary the tabstops. Type "S" `~Inile in the environment and the following pranpt will appear:

Set tabs: <right,left vectors> C(olfl N(o R(ight L(eft D(ecirna3. stop <etx>

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

VER IFY

See section 1.3.3.

Page 62

1.4. Yet Another Line Oriented Editor

Version II.0, February 1979



This text editor is intended for use on systems that do not have powerful screen terminals. It is designed to be very similar to the text−editor which accompanies DEC's RT−11 system. Its name is pronounced: Yaw−loo−ee.

The editor assi~xnes, but is not dependent on, the existence of the ~rkfile text. Upon reading it YALOE will proclaim ~rkfile 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 ~rkfile will no longer be empty.

The elitor operates in one of t~ 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 C~riand Mode. The Text Mode is entered whenever the user types a command which must be followed by a text string. After the ca~and 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. ~bte: 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 c~nnand strings in YALUE with <esc><esc> to execute them. (This is unlike the rest of the systems `immediate' commands.)

1.14.1 SPECIAL KEY C~1MANE6

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 443. (<eso> is the most particular anomaly to YALOE.)

<esc> Echoes a `$`. A single <esc> terminates a text string. A double <esc> executes the ccm~and string.



RIBOUT Deletes current line. ~, hard−copy terminals <linedel> ecI~ces `<ZAP' and a carriage return. C~ others, it clears the current line on the screen. In both cases the contents of that line are discarded by the editor.

CTRL H Deletes character from the current line. ~ <chardel> 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 exanpie: *I~LE ~ND !~ITh<CTRL X> *

A <linedel> would cause deletion of only I~ITh; CTRL X would erase the entire command. CTRL 0 Will switch you to the optional character set (i.e. bit 7 turned on). This works only on the TERAK 8510A. The CTRL 0 is used as a toggle bet~en 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 0. To get back just type CTRL 0 again.

OTRL F All output to the terminal is discaried by the <flush> syst~n until the next CTRL. F is typed.

CTRL S Al]. output to the terminal is held until <stop> another CTRL S is typed.

All other control characters are ignored and discarded by YALOE.

1.L~.2 Cc*IAND ARGUMENTS

Page 5J4



A cocrnnand argument precedes a catrnand letter ani is used either to indicate the number of times the command shoi.~d 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 arguTlent is needed; other coniTlands, howeier, require an argument.

Ccxi~nand arguments are as follows:

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).

rn m is a number 0. .9.

0 `0' refers to the beginning of the current line. I `I' means 32700. `−I' means −32700. It is useful for a large repeat factor. ~ is used only with the J, D and C coriTnands and represents −n, ~nere n is equal to tt−~ length of the last text argument used, for example *GT}jIS$_D$$ finds and removes THIS.

1.LI.3 CtMMAND STRINGS

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

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

*B GTHE INSERTED$ −3CING$ 51< 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 "51<" deletes text fran 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.

Page 65

As a rule, commands are separated Fran one another by a single <esc>. This separating <eso> is not needed, however, if the cam~and requires rio text. Commands are terminated by a single <esc>; a second <esc> signals the end of a caTlnarid string, which will thei be executed. Wien the execution of the command string is complete, the editor prompts for the next carinand with ".

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

1.L~4 THE TEXT B(FFER

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

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

1.4.5 THE CURSOR

The "cursor'~ is the position in your text w~re the next caTiTland will be executed. In other words it is the current "pointer" into the Text Buffer. Most edit camnands 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/CrJTPt.rr Cct~1MAN~

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

The L(ist cccznand prints the specified n~nber of lines on the console terminal without moving the cursor.

Prints all chaacters starting at the second preceding line and ending at the curscr.

Prints all characters beginning at the cursor and terminating at the ~4th <Cr>.

Prints fran the beginning of the current line up to the cursor.

The V(erify canmand 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. Arg~inents are ignored. The V(erify comnand is equivalent to a OIL (list) cori~nard.

The W(rite command is of the form

*W(ffle title>$

File title is any legal file title as decribed in ~ction 1.2 less the file type. The editor will automatically append a ~ThX~ 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 fran the filename. Refer to Figure 2 in section 1.2.14 for details on filename specifications.

The W(rite carinand 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 roan for the Text Buffer on the volurre specified in the file title given, the message:

OJTPUT ERROR. HELP!

will be printed. It is still possible to write the Text

Buffer out by writing it to anotI~er voltxne.

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 `.lEXT' 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 c~xinand has several forms

QU Quit and update by writing out a new SYSTDI.WRK.TEXT QE Q.iit and escape session; do not alter SYST~'4 .WRK. TEXT QR Don't quit; return to the editor Q A pr~npt will be sent to the terminal giving all the above cndices; enter option mnemonic (U, E, or R) only.

Executing the QU c~rznand is a special case of the write command, and the attempt to write out SYSmI.WRK.TEXT may fail. In this case use the W c~iinand to write out your file arid then QE to exit the editor.

The QR canmand 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 caTinand (intended for CRT terminals) erases the screen.

1. 4.7 CURSOR RELOCATION CCWANDS

J(~.xnp, A(dvance, B(eginriing, G(et, F(ind

When using character and line oriented commands, a positive (ri or +n) argu~ent specifies the ntxnber of characters or lines in a forward direction, and a negative argiinent the n~rnber 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 sane as any other character. For exanple assune the cursor is positioned as indicated in the followIng text C' represents the current position of the cursor and does not

appear in actual use. It is present here only for clarification):

THERE WP~S A CROO~D ~1AN~(CR> AND HU~PTY DUMPTY FELL ON HIM<CR>

The J(tznp canmand moves the cursor over the specified riuither of characters in the Text Buffer. The edit command _J4J moves the cursor back 4 characters.

THERE WAS A CRC~I~D MAN<CR> AND HU4PTY DUMPTY FELL ON HIM<CR>

Page 58



The camiand 1OJ moves the cursor forward 10 characters and places it between the `H' and the `U'.

THERE WAS A CROOKED MAN<CR> AND H~'U1PTY D~1PTY FELL ON HIM<CR>

The A(dvance caiinand moves the cursor a specified ni.xnber of lines. The cursor is left positioned at the beginning of the line.

Hence the canmand OA moves the cursor to the beginning of the current line.

THERE WAS A CROOKED MAN<CR> ~AND H1R~1PTY DUMPTY FELL ON HIM<CR>

The canrnand −1A (or −A) moves the cursor back one line.

~THERE WAS A CROOKED MAN<CR> AND HUMPTY DUMPTY FELL ON f−JIM<CR> The B(eginriing canrnand moves the cursor to the beginning of the Text Buffer. Use /J to move to the end of the buffer. Search canmands are used to locate specific characters or strings of characters within the Text Buffer. The G(et and F(ind commands are synonymous. Startir~ 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 rith 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 canmand 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 C~1MANDS

I(nsert, D(elete, K(ill, C(hange, eX(cha~ge

The I(nsert camnand 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. ~casionally with large insertions the temoorary insert buffer becomes full. Before this happens a message will be printed on the console terminal, `Please finish'. In response

Page 69



type tw3 successive <esc>s. To continue, type I to return to the Text mode.

NOTE: Forgetting to type the I coi~and will cause the text entered to be executed as ca!riands.

The D(elete c~inand renoves a specified ni.xnber of characters from the Text Buffer, starting at the position of the cursor. Upon completion of the caTinand, the cursor's position is at the first character following the deleted text. a_2n$$ Deletes the t~ characters immediately preceding the cursor. *B$FHOSE $=D$$ Deletes the first string `HOSE ` in the Text Buffer, since :D used in combination with a search caTxnand will delete the indicated text string.

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

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

Deletes all lines in the Text Buffer after t1~e cursor.

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

*OCAPPL~$$ Replaces the characters from the beginning of the line up to the cursor with `APPLES', (equivalent to using OX). *B~OSE$_CLIzARD$$ Searches f or tI~ first occurrence of `HOSE' in the Text Buffer and replace it. with `LIZP1RD'.

The eX(change command exchanges n lines, starU.ng at the cursor, with the indicated text string. The cursor remains at the er.d of the changed text.

Page 70

71

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

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

~/XTEXT$$ Exchanges the lines from the cursor to the end of the Text Buffer with the text `TEXT', (equivalent to using IC or /DI).

1 .t~. 9 OTHER COMMANES

S(ave, tJ(nsave, M(acro, N (macro execution) and `?`

The S(ave carznand 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 cam3and ~uld have overflowed the Text Buffer, the editor will generate a message to this effect and not perform the save.

The U(nsave c~iinand 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 riot 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 cccmnand OU.

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

n~4~ccnrnand string%

This says to store the coaxnand 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 coaxnand string. Example of a macro definition:

Page 71



*94%~EGfl4 $=CEND BEGIN$V$~$$

This defines macro ni.~iber 5. When macro nuinber 5 is executed, it will look for the string `BEGIN', change it to `END BEGIN', arid 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 carznand, N, executes a specfied macro comrnar~ string. The form of the command is:

n ?~n$

Here n is simply any command argu~ient as previously defined; ~`n is the macro ntxriber (an integer O..9) to be executed. If m is omitted, 1 is ass~ined. Because the digit m is technically a command text string, the N camnand must be terminated by an <esc>.

Attempts to execute undefined macros cause the error message `Unhappy macntrn'. Errors encountered during macro exeo.ition cause the message `Error in macro'. Errors encountered in macro command syntax cause the message `Error in macro definition'.

The ? commar~ 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 nusbers of the currently defined macros.

Page 72



1.L~.1O S~}1MARY OF ALL COIMANDS

n – an argument m – macro ni.znber

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 <ese>. nD: Delete n characters. E: Erase the screen. nF: Firxl 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: ~fine macro ni.xnber m. nMn: Performinacro number m, n times. 0: – invalid – – invalid – Q: Qiit this session, followed by: U:(pdate Write out a new SYST~1.WRK.TEXT E:(scape Escape from session R:(eturn Return to editor 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! Put the next n lines of text from the cursor position into the Save Buffer. – 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 Write this file (Iran start of buffer); `W' must be followed by <filename> <esc>. nX: Delete ri lines of text, and insert the following text; terminate with <esc>. – invalid – Z: – invalid –

Page 73



1.5. Debugger

Version II.0, February 1979



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

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

Ed.



1.6. Pascal Compiler

Version I.5, September 1978



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

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

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

The identifiers appearing on the screen are the identifiers of the program and its procedures. The identifier for a procedure is displayed at the moment when compilation of the procedure body is started. The numbers within `[ ]` indicate the number of (16 bit) words available for symbol table storage at that point in the compilation. The numbers enclosed within < > are the current line numbers. Each dot on the screen represents one 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 nunbers will be accompanied by a textual message upon entry to the editor if the file `*SYSTEM.SYNTAX` iS available.

1.6.1. Compile Time Options

Compile time options in the UCSD Pascal compiler are set according to a convention described on pages 100−102 of *Jensen and Wirth*, where compile time options are set by means of special “dollar sign” comments inside the Pascal progran text. The syntax used in UCSD's compiler control comments is essentially as described in Jensen and Wirth. The actual options and the letters associated with those options

bear little resemblance to the options listed on pages 101 and 102 of Jensen and Wirth. Following is a description the various options currently available to the user of the UCSD Pascal compiler.

B

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

C

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

D

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

Default value: D−

D−

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

D+

causes the compiler to emit breakpoint instructions.

G



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

Default value: G−

G+

allows the use of the GOTO statement.

G−

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

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

I

When an ‘I’ is followed immediately by a ‘+’ or ‘−’, the control comment will affect the boolean variable IOCHECK within the compiler. An alternative use of ‘I’ in a compiler control comment causes the compiler to include a different source file into the compilation at that point. See **Include File Mechanism**, below, 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 110 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:

```
(*$I filename *)
```

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 filename 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 UCSD 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 editor's 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 disk file 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 disk file called “DEMO1.TEXT”:

```
(*$L DEMO1.TEXT*)
```

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

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

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



P

Page. Pages listing file.

Q

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

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

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

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

R

Thus 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: 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 full size, single density floppy disks this amounts to a factor of two reduction in compile speed. This option must occur prior the the compiler encountering any Pascal syntax.



Default value: S−

S+
: puts compiler in swapping mode.

S−
: puts compiler in non−swapping mode.

U

User Program Option

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

Default value: U+

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

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

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

Use Library Option

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

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

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


84

1.7. Basic Compiler

Version I.5, September 1978

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



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

The UCSD Basic Compiler has been written in the Pascal language. Some of the intririsics of the Pascal language, which are not found in standard Basic, are found within the U~SD version of Basic. Many of these are noted in the first section, all of them are noted or recapped in the second. 1X SYST~1.CCMPILER The UCSD BASIC Compiler is invoked just like the Pascal compiler, provided the compiler code is named *SYSTEM.CcT~PILER. Originally it will be named BASIC.COMPILER. If you want a disk to be BASIC oriented, you must change the name of, or remove, the Pascal compiler, and change the name of BASIC.C~PILER to *SYSTD~1.CCMPILER. That disk, and any copies of it, will now compile BASIC programs as a result of the C(ompile or R(un command.

DESCRIPTION ~ FEATURES INCLUDED

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

VARIABLE NAMES

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

INTRINSIC ARITHMETIC FUNCTIONS

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



~(x) Returns the base of the natural 1ogaritt~s raised to the power x.

INT(x) Returns the value of x roi.n~ded to the nearest integer.

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

LN(x) Returns the natural log of x.

MOD(x,y) Returns x modulo y.

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

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

INTRINSIC STRING FUNCTIONS

CAT$(si,s2,...) Returns a string which is equal to the concatenation of afl. the strings in the para~eter list.

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

DEL$(zi,x,y) Returns the contents of the string st with y consecutive characters oeleted. The deletion starts with the character a~ position x.

INS$(sl,s2,x) Returns the contents of string s2 with string si inserted ininediately before the character which is at position x.

LZ(sl) Returns the length of the string si.

P~(si,s2) Returns an integer which is equal to the position of the first character in the first occurrence of the string SI in the string s2.

OTHER FUNCTIONS

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

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

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

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



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

These functions return IORESULT as described in section 2.1.

PROGRAMMING STAT~ENTS

Aritlinetic statements aid operations – , + subtract,add / , divide ,multiply ** exponeritiation

Relational operators equals <> , >< not equals > greater than less than >: , greater than or equal less than or equal

INPUT list or INPUT I/c list

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

PRINT list or PRINT #~ list

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

FOR var expl TO exp2 STEP exp3

Page 87



NEXT var

Each execution of the loop increnents the loop counter "var" by the anouit of expression 3. If the S~P is ~nitted it is ass~.xned to be ~. oily increasing STEP values are allowed. Evaluation of limits arid incr~nents is done at the beginning of the loop. Note that RETURN's into or GOTO's into a FOR loop may cause the loop to be undefined.

~ expi (relation operator) exp2 THEN (line nunber) GO~

Either the reserved word THEN or GOTO can be use~ in this statement. If the relation between the expi and exp2 is found to be true the branch occ~.rs. A string is considered to be less than another string if it is lax icographically smaller.

ON exp GOTO(lnT,1n2..)

If t~ expression, ~ien rounded, evaluates to 1 it goes to the first line number (Thi) if it evaluates to 2 it goes to lri2, etc. This is the only form of the canputed GOTO which is available. if the expression is out of range an error occurs.

D~ FNnane(list)=expression or DEF FNnane(list)

FNEND

Single line and multi−line functions are allowable. The function n~e must be a legal variable name for the type of value returned. Functions may be defined recursively. The parameter list is called by value, triat is, changes ir.side the function don~t affect the value of the external. par aneters.

LET var=exp or var:exp

This c~~iand assigns a new value to the variable. If the vari~bl.e is string, the expression must evaluate to a string, and if a rea, evaluation must be to a real.

DIM var (n1,n2,...)

A single or multidimensional array may be declared with this conrand.. The variable name determines the type of the array. The array indices are O..nl,O..n2,... Both real and string multidimensional arrays can be used. If no dimensions are declared the dimensions are assumed to be O..1O, O..1O, O..1, O...1 ... The number of dimensions aut~naticaliY declared depends on the nunber of dimensions ~ich are used in the program, but must be consistant over all uses of any given array.

Page 88



GOSUB linenumber

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

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

GOTO linenumber

Prograii execution jumps to the given line number.

R~1 text

This line is a re~nark.

UNIQUE FEATURES OF UCSD BASIC

Arithmetic

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

Continuati~ri of statements is allowed. Any line not beginning with a line number is assumed to be the continuation of' the line above.

Functions: All paremeters of functions are call by value. You are not allowed to use the paraneters to return values from a function. Function calls are allowed to be recursive. Strings: The string functions and procedures are those found in the UCSD Pascal language.

Arrays: Arrays of more than t~ dimensions are allowed.

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

Subroutines: Subrwtines may be recursive.

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

PASCAL FUNCTIONs: The code of PASCAL FUNCTIONs may be added to the BASIC compiler as new standat~d BASIC functions. This is accomplished by a straight−forward addition to the BASIC compiler.

Page 89



FEATURES TO BE ADDED

Certain features of' the UCSD Basic car~piler are still in the process of being iznplenerited. The most ~inportant of these are listed below.

Data and Read: The standard initialization statements.

Matrix statement for standard matrix operations.

Integer variables.

More standard functions.

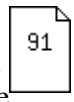
RUNNING A BASIC PR~RAM

Create the BASIC progran using one of the system text editors. ~ce you have ensured that the BASIC compiler has been naned SYSm4.CCT*IPILER, you can use the ca~imands C(anpile and R(un at the C~44AND level, just as if you were using Pascal on a disk which has the Pascal c~npiler as its SYSmI.CGMPILER. For a more detailed description of C~44AND see Section 1.1.

Page 90

1.8. Linker

Version II.0, February 1979



The UCSD LINKER allows t~ 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 progr~s 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 (eit~r procedures or functions), the calling program declares those routines to be EXItRNAL, much as PR()~EDURES 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. P. UNIT is a group of related routines which will be used together to perform a common task. UCSD TURTLEGRAP!−iICS is an example of a UNIT containing procedures and functions with which a ~turtle!I can be moved on the screen. A UNIT can be used by typing the reserved word USES <unitname> directly after the PROGRAM <identifier>. For more information on UNITs, see Section 3.3.2.

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

1.8.1 USING THE LINKER

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

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

Page 91



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 ~rkfile, or

(2) the external routines to be linked reside in library files other than *SYST~UB~Y

In order to explicitly invoke the LIM~R, the user types `L' at C~unand level and receives the prompt:

Host file?

The hostf lie is the file into which the routines or UNITs are to be linked. The LI~R appends .COCE to all file names typed in ~cept f~r *<ret>. Typing a <ret> in response to the prompt oauses the LINKER t~ use the

~rkfile as the hostfiie. The LIN}~R then asks for :he name(s~ of the library files in which the UNITs or EXTERNAL routines are to be found:

Lib file? <codefile identifier> Lib file? <codef lie 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 *5~T~jLI3R~Y The user will be notified about each library file that is successfully opened.

Example: Lib file? * <ret> Opening *SYST~.1.LIBRARY

For information on LIBRARIES and the LI3RARL~.N see Section ~4.2.

~`rien all relevant libf lie names have been entered the user

must type <ret> to proceed. The LINI~R will now prompt with:

Map file? <file identifier> <ret)

The LIN~R writes the map file to the file requested by the user. The map file contains relevant LIN?~ER info regarding the link~.ng process. Responding with <ret> to this prompt will suspend this option. Note that .TEXT is appended unless a `.` is the last lettar of the filename. The LIN~R 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 iame as that of the host file). Linking will conTnence after the <ret> following the output file name has~been typed. ~n empty line, <ret> only, causes the output file to be placed in the workfile e.g. *SYST~WRKCO~

Page 92



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

1.8.2 LINKER CONVENTIONS AND IMPLEMENTATION

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

RECORD DISKINFO: ARRAY{O. .15] OF RECORD CODELENG, CODEADDR : INTEGER END;

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

SEX3KIND: ARRAY[O.. 15] OF (LINKED,HOSTSEG,SEGPR~UNITSEG,

SEPRTSEG); TEXTADDR: ARRAYtO. .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 progran if the program has external references.

SEGPROC: the segkind assigned to a PASCAL segment procedure.

UNITSEG: the segkind assigned to a compiled SEG4ENT. (see Se~tiOfl

3.3.1)

Page 93



SEPRTSEG: This segkind is assigned to a separately compiled procedure or function. Ass~nbly language codefiles are always of this type, as ~ll 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, E~'PROC, EXTFUNC, SEPPROC, SEFFUNC, SEPPREF, SEPFREF);

LIENTRY=RECORD NAME: ALPHA; CASE LITYPE: LITYPES OF UNITREF, GLOBREF, PUBLREF, PR IVREF, SEP PREF, SEPFREF, ~NSTREF: (FO~1AT: O~OR~AT; (format of lientry.n~ne can be any of BIG, BYTE or WORD.) NREFS: INTEGER; (0 of references to lientry.riarne ~n compiled code segment) W~4~DS: LCRANGE); (size of privates in words) GLOBD~: (HGPRCC: PR~RANGE; (which procedure it occurs in) ICOFFSET:ICRANGE); (byte offset in p−code) PUBLDEF: (BASEOFFSET: LCRANGE); (compIler assigned word ~ffset~ CONSTDEF: (CONSTVAL: INTEGER); (users defined value) EXTFR~, ~(TFUNC, SEPPROC, SE?FUNC: (SRCPRcC: PR~RANGE; (procedure nunber in source segment) NPARAMS: INTEGER); (riunber of par~neters 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 B word records, but only the first NREFS of the,, are valid.

Page 9L4

1.9. Assembler

Version II.0, February 1979



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

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

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

1.9.1 USAGE

Before attempting to execute the assembler program for a specific machine, an opoodes file (Z80.OPCODES or 11.OPCODES) must be located on the system disk. The errors file (Z80.ERRORS or 11.~R0RS) contains the error mess~es that are used for error flagging during the assembly. This file is optional; if used, it must also appear on the system disk.

To use the UCSD assembler, type A(ssem from the Co~miiand line. This will execute SYST~1.ASSMBLER. (The user should arrange that the right version of the assembler (PDP–11 or Z80) have that title.)

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

The next prcnpt line is:

Page 95



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

As usual for a console or printer output the words C0~S0LE or PF~INTER must be followed by a colon, i.e. CONSOLE:. If the colon is neglected the outp&~ is sent to a file of the nane given. At this point, the progran reports whether or not the output device (if aty) is on line. The assembled code is written out to a file called eSYSTE?1.WRK.CODE which cannot be executed by itself but must be changed to link in with a host ftle.

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

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

INCLU~ <FILE. ID>

indicating which file has been ir~luded.

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

The reference symbol table consists of three parts. The first colLznn represents the symbol identifier, the second, the symbol type, and the third, the locaticn that it is defined or the value it has. Actual values are given for the symbols representing absolutes and definition locations are given for the symbols representing labels. The lccat~.on nu~ber is given as a hi−byte first ni.rnber and corresponds to the index niinbers on the left hand side of the listing. Only symbols which have definition locations or absolute values have nuithers in the third colunn; other types have dashes. Following is an example of an assembled listing with symbol table.
Page 96

Page 97

Pa~ge. r~LLmbvu(.Plg ~VLC.'~... ed.

Page 98



PAGE − 1 PRIMARYZ FILE: #5:PRIMARY.Z

oooo: Memory 0000 0030 00001 00001 00001 00001 00001 00001 oooo: 1000: woo: FD i00~1 CD 10071 FD 100BI CD 100EI FD 1012: CD 10151 C3 1018: 1002* 1810 1018: 1018: 00 10191 OA 1O1AI 0090 loic: 0002 1OIE 0000 10201 0010 1022: 00 10231 0817 1025 1 1009* 2510 10251 10251 00 10261 OA 10271 0093 1029: 0002 102B 0000 102D1 0010 10~1 00 10301 1017 1032 1010* 3210 1032 10321 00 10331 CA 103L~ 0095 10361 0002 10381 0000 103A 0010 103C 00 103D1 1817 103F1 1 03F1

.PROC PRIMAR~ 6068

;Rom−based floopy driver. ;First location in memory ;Entry point of bootstrap ;Sector start of 2nd bootstrap ;Sector start of BIOS part 1 ;Sector start of BIOS part 2

;Primary boot for ZILOG 033

;Get block for second bootstrap

;Get block for part 1 of BIOS

;Get block for part 2 of BIOS

;Junp into second bootstrap

;Unused ;Read command ;Memory bc. for second boot ;Number of bytes in boot ;Completion return address ;Error in return address ;Cornpletion result code ;Disk block of second boot

Unused ;Read command ;Memory location or BIOS part 1 ;Nurnber of bytes in BIOS part 1 ;Completion return address ;Error return address ;Cornpletion result code ;Disk block of BIOS part 1

;Unused ;Read command ;Memory location of BIOS part 2 ;Number of bytes in BIOS part 2 ;Cornpletion return address ;Error return address ;Completion result code ;Disk block of BI~ part 2

after initialization:

FLOPPY SEcME~4 SEC ENT SECt~K B1DSK B2LSK

* EQU * EQU * EQU * EQU * EQU * EQU

OBFCH 900011 900011 08H + 17O~1 iCIl + 170011 1811 + 170CI–J

*ORG 1000H

21 **** FDOB 21 ~ FDOB 21 ~ – F DOB 0090

PRIMARY LD CALL LD CALL LD CALL JP

SECREAD * BYTE * BYTE *WORD *WORD * WORD * WORD * B YTE * WORD

B 1 READ * BYTE * BYTE * WORD .WORD * WORD *WORD * BITE * WORD

B2READ

IY, SECREAD FLOPPY IY ,B 1 READ FLOPPY IY ,B2READ FLOPPY SECENT

s–s OAH SECMEN 200H s–s PRIMARY s–s SEC DSK

s–s OP~H SECMEN+300H 20011 s–s PRIMARY s–s B1DSK

*BYTE $–$ .BYTE OAH *WORD SECM~+50OH *WORD 20011 .WORD $–$ *WORD PRIMARY .BYTE $–$ .WORD B=DSK

* END

Page 99



PACZ– 2 PRflIARYZ FILE:#5:PRfl1ARY.Z SYMBOLTABLE DUMP

AB – Absolute LB – Label UD – Undefined MC – Macro – Ref – Def PR – Proc FC – F~x~c PB – Public PV – Private CS – Constant

B1E6K AB 17101 B1READ LB 10251 B2CSK AB 1718: B2REP~D LB 1032 FLOPPY AB C8FD PRIMARY LB 10001 PRB1ARYZ PR –~ SECDSK AB 7708 SECENT AB 9000 SE~1EM AB 9000 SECREAD LB 1018

NOTES:

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

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

If a star (`) appears after the location ntrnber at the left of the listing, it indicates that a forward reference occurring earlier in the asse~nbly has been resolved. The ni.znber to the left of the `" is the location where the reference occurred while the nunber to the right is the new contents of that location.

1.9.2 HIGH−LEVEL SYNTAX

All objects declared before the first .PR~C or .FUNC are available for use throughout the assembly. No code is allowed to be generated before the first .PR(X or .F1JNC. The symbol table is reduced at the beginning of each .PROC or .FUNC to the point where it was at the start of the first .PR~C or .FtJNC.

Only labels may begin in the first colunn and may optionally be followed by a col~i. Local labels must have `$` in the first coliznn and may be up to 8 digits long. If the statement has rio label, the first column must contain a space.

All assemblies must end with a .END. I4owever each .PROC or .FUNC need not because they are ended by the occurrence of the next .PROC or .FUNC. Only the last one needs a .END.

A genera]. railroad diagraii for all assembly files looks like:

Page 100



The non−code generating operations are:

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

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

1.9.3 E)~RESSIONS (one−pass restrictions)

Since the Adaptable Assembler makes only one pass through the source, scolething must be assumed (upon encountering an undefined identifier in an expression) about the nature of the identifier in order for the assembly to continue. It is therefore assumed that the undefined identifier will eventually be defined as a label, which is the most probable case. Any identifier which is riot a label must be defined before it is used.

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

Local labels are mainly used to jump around within a ~nall segment of code without having to use up storage area needed by regular labels. The local label stack may hold up to 21 labels. These are cut back every time upon encountering a regular label and are thus rendered invalid. An example of the use of local labels is shown below, the jump to label $O~4 being illegal.

Page 101



$03 STA L~ ;LEGAL 1~E CF LOCAL LABEL

JP NZ,$03

JP NZ,$OL~ ;IL.LEGAL USE CF LOCAL LABEL REALLAB .EQU $ $O~4 .EQU $

Identifiers are character strings starting with an alpha character. Other characters must be aiphanuneric or the ASCII underline C"). Quy t~ first 8 characters are meaningful to the assembler even ~ more t~y be entered.

The following operators can be used in expressions processed by this assembler. For unary operations: ~ plus `−` minus `` ones complement For binary operations: `+` plus `−` minus `~` exclusive or `I, multiplication `/` truncating division CDIV) `%` remainder division (MOD) `~` bit wise OR `&` bit wise AND ~:` equal (valid only in .IF) `(>` not equal (valid only in .IF)

All constants must start with an integer 0−9. All operations are applied to whole ~.ords.

The default radix is Hex for the Z80 version and c~tal for the ?~P−11.

1.9.L~ ASS~4BLER DIRECTIVES: OVERVIE'd

Assembler directives (also referred to as "pseudo−ops") allow the prograr~ner to instruct the assembler to do various functions other than provide direct executable code. The following directives are cc~nmon to all UCSD versions but may differ fran manufacturer's standard syntax.



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

For example: [label] .ASCII "<character string>" indicates that a label may be given but is not necessary and that between the double qu3tes must go the character string to be converted (not necessarily the words "character string").

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

value any rn.znerical value, label, constant, expression.

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

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

expression any legal .expression as defined in Section 1.9.3.

identifier:integer list a list of one or more identifier_integer pairs separated by commas. The colon−integer is optional in each pair arid the default is 1. ~all examples are included after each pseudo−op definition to supply the user with a reference to the specific syntax and form of that directive. The larger example, included in section 3.3.2, is used to show the combined use and detailed examples of directive operations.

1.9.14.1 DELThIITING DIRECTIVE FOR ROIJTINES

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

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

Page 103



ffiOCEDURE FARI~..E(X,Y:REAL) ;EXTERNAL; the associated declaration for the .PRC~ would be

.PRO FAR~U..E,L~

A .P~, .FUNC, or any assembly routine should be inserted intc the *SYSTEM.LmRARY (execute L~RARIAN) so that it can be referenced by the `SYST~1.LINKER and linked in at ~ time. An alternate method would be to execute the LINI*~R and tell it what files to link in. Either method works. f~wever, if the Pascal host is updated and the assembly routines aren't in the SYSTDI.LIBRARY, the linker will have to be executed after each update. Therefore, we suggest that the routines be inserted into the `SYSTEM LIBRARY to avoid this repetition. If the linker is called automatically using the Run command, it will search the SYSTB~hLIBRARY for the appropriate definition of the assembly routine and link the two together.

.PROC Identifies a procedure that returns no value. A .PRCC is

ended by the occurrence of a new .PRcC, .FtJNC, or .END.

FORM: .PROC <identifier>[,expression)

[expression) indicates the n~.rnber of' words of ~raneters expected by this routine. The default is 0.

EXAMPLE: .PRCC DL.DRIVE,2

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

FORM: .FUNC <identifier)E ,expression:

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

EXAMPLE: .FUNC RAN~1,~

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

1.9.14.2 LABEL DEFINITIONS AND SPACE ALLOCATION DIRECTIVES

.ASCII Converts character values to ASCII equivalent byte

Page lOLl



constants and places the equivalents into the code stream.

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

EXPNPLE: .ASCII "HELLO"

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

.ASCII "AB" .BYTE 314 ; 142 octal .ASCII "CD"

Note: The 314 is the ASCII niinber for a double quote in hex. The representation actually used will depend on the defaui~ radix of the particular machine in use.

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

FORM: Elabel) .BYTE [valuelist]

the default for no stated value is 0.

EXAMPLE: T~1P .BYTE

the associated output would be: 014

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

FORM: [label] .BLGCK <length>E,value]

Page 105



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

EXAMPLE: T9IP . BLOCK 14,6

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

.wa~D Allocates a hord of space in the code strean for each vaJ.ue in the valuelist. Associates the declaration label with the ~rd space allocation.

FORM: Elabel] .WORD (valuelist>

EXAMPLE: T~4P .WORD 0,2,14,...

the associated output ~uld be: 0000 0002 00014 (~rds ~th these values in them)

EXAMPLE: Li .WORD L2

1.2 .EQU $ $ represents the LC on the Z80 .WORD 5.

if LC was 50 at the .EQU the associated output ~uld be:

0050 (* assignment due to t}~ L2 value *)

0005 C' assignment due to the .WORD 5 `)

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

Page 106



FORM: <label> .EQ(J <value>

EXAMPLE: BASE .EQU R6

.ORG Sets the current location counter (LC) to the value of the .ORG. It ~uld normally be used in a stand−alone program. For example, there is one .cRG in the 8080/Z80 interpreter. The current impl~nentation allows one to .~G only in the forward direction.

1.9iL3 MACRO FACILITY DIRECTIVES

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

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

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

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

.MP~CRO Indicates the start of a macro and gives it an identifier.

.ENE~1 Indicates the end point of a MACRO.

Page 107



FORI: .MACRO <identifier>

* (macro body)

EN~1

EXNIPLE: .MACRO HELP STA %1 ; < comment > LDA %2 ; < cc~nent > .~

The 1 isting ~Eiere the macro call is made may look like:

FELP FIRST,SECOND STA FIRST LDA SECOND

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

1 .9. 14* 14 CONDITIONAL ASS~'1BLY DIRECTIVES

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

The conditional expression takes one of two forms. The first is the normal arit~netic/logica1 expression used elsewnere in the assembler. This type of expression is considered false if it evaluates to zero; true otherwise. The second form of conditional expression is comparison for equality or inequality (indicated by `:` and `<>`, respectively). cne may compare strings, characters, or arithneticflogical expressions.

.IF Identifies the beginning of the conditional.

.ENtC Identifies the end of a conditional .~

.~..SE Identifies the a1ternate~ to the .~. If the conditional

expression is egual to 0 then the else is used.

Page 108



FORM: Elabel] .IF <expression>

.ELSE (` only if there is an else *)

iNDC

where the expression is the corkiitional expression to be met.

EXI%MPLE: IF LABEL1−LABEL2 ;arithrnetic expression This text assembled only if subb−action result is now zero

.IF "%`I" :t~STUFFtt ;comparison expression This text ass~nbled if subtraction above was true and if text of first parameter (ass~xne we are in macro) is equal to "STUFF" .ENEC ;terrninate nested corid.

.ELSE This text assembled if subtraction result was zero

.ENDC ;terminate outer level ;conditional

.9. ~4. 5 PASCAL HOST Cc~1MUNICATION DI R~TIVES

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

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

Page 109



FORI: .CON~ <idlist>

EXAMPLE: (` see exanpie after .PRIVATE I)

.PUBUC Allows a variable declared in the global data segment of the PASCAL host to be used by an ass~nbly language routine and the host progran.

Pt~M: .PUBLIC <idlist>

EXAMPLE: (` see exanpie after .PRIVATE `)

.PRIVATE Allows variables of the ass~nb1y routine to be stored in the global data segment and yet be inaccessable to the Pascal host. These variables retain their values for the entire execution of the progran.

FORM: .PRIVATE (identifler:integer list>

the integer is used to communicate the n~nber of words to be allocated to the identifier. EXAMPLE: (* for .CONST, .PRIVATE, .PUBLIC I)

Given the following Pascal host progran:

PROGRAM EXAMPLE; CONST SETSIZE=50; LENGTH:S0; VAR I,J,F,HOLD,COUNTER,LtC:INT~ER; LST1:ARRAYEO..9) OF CHAR; BEGIN

END.

and the following section of an ass~nbly routine:

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

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



the LSr2:9 causes the variable LST2 to correspond with the beginning of a 9 word block of space in the global data segment.

1.9.'L6 EXTERNAL R~ERENCE DIRECTIVES

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

Identifies a label that is defined in the current routine

and available to be used in other .PR~Cs or .FUNCS.

FORM: .tEF <identifierlist>

ECP.MPLE: (* see listing in section 3.3.2.3 for example *)

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

FORM: .REF <identifierlist>

DCP~MPLE: (* see listing in section 3.3.2.3 for example *)

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

~ L~* 7 LISTING CONTROL DIRECTIVES

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

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

Page 111



FORM: .LIST or .~LIST

.PPIGE Allows the prograw~er to explicitly ask for top of form p~e breaks in the listing. PAGE

The title is only cleared at the start of the file. In section 1.9.1 the title S'iMBOLTA8LE D(JIP was not set by a .TITLE directive. That heading is always used on pages containing symboltable d~.tnps. Upon assembling a further procedure the heading printed returns to what it was before the syrnboltable dunp.

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

INTERP SI~1BOLTABLE DUMP

shown in Section 1.9.1 was caused by a .TITL~ directive.

FORM .TITLE <title> where <title> is a string EXAMPLE .TITLE QRC12 interpreter

l.9.Z~.8 Ffl_E DIRECTIVES

.INCLIJDE Causes the indicated source file to be inch~ed at tha.t point.

FORM: .INCLUDE <file identifier.TE~> where the file identifier is any file to be inclt.~ed. Only spaces are allowed between the end of the file name and the end of the Include line.

~RRECT EXAMPLE: .INCLUDE SHORTSTART.TE)Cr

C~RECT EXAMPLE: .INCLLJDE SHORTSTART.TEXT calls starter

IN−~RRECT EXAMPLE: .INCLUDE SHORTSTART.TEXT ; calls starter

For a list of general errors and also notes on the 280 and P~P−11 based machines see Table 6. machines see Table 6.

Page 112

1. The UCSD Pascal System

1.1. Introduction and Overview 1
1.2. File Handler 7
1.3. Screen Oriented Editor 31
1.4. Yet Another Line Oriented Editor 63
1.5. Debugger 75
1.6. Pascal Compiler 77
1.7. Basic Compiler 85
1.8. Linker 91
1.9. Assembler 95

Intrinsics Parameters

Version II.0, February 1979

Warning:



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

Parameters

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

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

ARRAY
: a PACKED ARRAY OF CHARacters

BLOCK
: one disk block (512 bytes)

BLOCKS
: an INTEGER number of blocks

BLOCKNUMBER
: an absolute disk block address

BOOLEAN
: any BOOLEAN value

CHARACTER
: any expression which evaluates to a character

DESTINATiON
: a PACKED ARRAY OF CHARacters to write into or a STRING, context dependent

EXPRESSION
: part or all of an expression, to be specified

FILEID
: a file identifier, must be

```
VAR fileid: FILE OF <type>;
or TEXT;
or INTERACTIVE;
or FILE;
```

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

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

RELBLOCK

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

SIMPLVARIABLE

any declared Pascal variable which is of one of the following types: `BOOLEAN`, `CHAR`, `REAL`, `STRING` or `PACKED ARRAY[..] OF CHAR`



SIZE

an INTEGER number of bytes or characters; any integer value

SOURCE

a `STRING` or `PACKED ARRAY OF CHAR`cters to be used as a read−only array, context dependent or as specified. In the case of string intrinsics, it must be STRING.

SCREEN

an array 9600 bytes long; or as needed.

STRING

any `STRING`, call−by−value unless otherwise specified, *i.e.* may be a quoted string, or string variable or function which evaluates to a `STRING`.

TITLE

a `STRING` consisting of a file name

UNITNUMBER

physical device number used to determine device handler used by the interpreter

VOLID

a volume identifier, `STRING[7]`

2.1.1. String Intrinsics



********** **ft**********

* STRING INTRINSI~ * * Section 2.1.1 * ********** **********

Version 11.0 February 1979

FUNCTION LENGTH (STRING) : INTEGER

Returns the integer value of the length of the STRING.

Example:

GEESTRING := `123L~567'; WRITELN(LENGTh(GEEZTRING),' `,LENGTh(')); Will print:

70

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:

S11JFF :: `TAKE THE BOTTLE WITH A METAL CAP'; PATTERN := `TAL';
WRITELN(POS(PATTERN, STUFF));

Will print:

26

FUNCTION CONCAT (SOURCEs) : STRING

There may be any nunber 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 STRI~ZG.

FUNCTION COPY (SOURCE , INDEX , SIZE) : STRING

This fixiction returns a string containing SIZE tharacters copied fran SOURCE startir~ at the INEEXth position in SOURCE. Exanpie:

TL := `KEEP S~4ETHING HERE'; KEPT :: COPY(TL,~S('S',1L),9); WRITELN(KEvr);

Will print:

Sa'IETHING

PROCEDURE DELETE C DESTINATION , INDEX , SIZE)

This procedure r~noves SIZE characters fran DESTINATION starting at the INDEX specified. Exanpie:

OVERSIUFFED : = `THIS STRING HAS FAR TOO MANY CHARACTERS IN IT.';
CELETE(OVERSTUFFED,P~(`HAS' ,OVERSTUFFED)+3,8); ~RITELN(QVERSTUFFED);

Will print:

Th~ STRING HAS MANY CHARACTERS IN IT.

PROCEDURE INSERT (SOURCE , DESTINATION , INDEX)

This inserts SOURCE inth DESTINATION at the INDXth pcsitior~ ~) DESTINATION.

Example:

ID := `INSERTIONS'; MORE :: ` DD~iONSTRATE'; DELETE(MORE, LENG1'H(MORE), 1);
INSERT(MORE,ID,P~('IO' ,ID)); WRITELN(ID); Will print:

fl4SERT DE.MONSTRATIONS S

PROCEDURE STR (LONG , DESTINATION)



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

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

Will print:

$1020395.03

Note about using strings and string functions:

In order to n~intain 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 ~predictable results if range−checking is off or cause a run−time error Cl) if range checking is on.

Page 119



2.1.2. Input and Output Intrinsics



**********]********** **********

* INPUT AND OUTPUT INTRINSICS * * Section 2.1.2 * ********a**********
****

Version 1.5 September 1978

PROCEDURE RESET C FILEID II, TITLE]); PROCEDURE REWRITE C FILEID, TITLE);

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

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

RESET (FILEID) without optional string parameter "rewinds" the file by setting the file pointers back to the beginning (zero th record) of the file. The booleari functions ~ 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 IGRESULT as a result of the implied GET caused by the RESET.

PROCEDURE UNITREAD (UNITNUMEER, ARRAY, LENGTH, [BLOCKNUMEER], [INTEGERJ); PROCEDURE UNIIWRITE C UNITNUIIBER, ARRAY, LENGTH, IBLOCKNUMBER], [INTEGER]); { sequential 1 C 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. Unitnunber 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 fronVto. The LENGTH is an integer value designating the nunber of.

bytes to transfer. The BLOCIQIT.NBER is required only when using a block−structured device (i.e. a disk) and is the absolute bloc~cuznber at which the transfer will start fran/to. If the BLcCKNLMBER is left out, 0 is assuned. The INTEGER value is optional (asstrned 0) and ir~icates (if 1) that the transfer is to be done asynchronously. The bloclc~.xnber is not necessary. A `, ,INT~ER' will be sufficient. (See UNITBUSY and UNIIWPJT.)

FINCTION UNITBUSY (UNITN~1BER) : BOOLEAN;

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

Ex3nple: UNITREAD(2 (non−echoing keyboard} ,CH[0~, 1(for one character},(no block no.i,1{asynchrcnousI); WHILE UNITB(JSY(2){While the READ has not been completed ~O `~1RITELN(OUTPUT,'I ~n waiting for you to type somethirg~; WRITELN(ouTptJr,'Thank you for typing a ~,CHt0J);

Execution of this exanpie will continuously type out the lir.e `I an waiting for you to type something' until ~ character is struck or. the keyboard. Suppose a `!` were typed. r~ message Th.ank you fc~ typing a !` will then appear, and progran execution will proceed normally.

Currently implemented only on DEC computers.

PROCEDURE UNfll'JAIT (U NITNUMBER);

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

WHILE UNITBUSY(n) DD {waste ~ small anount of time:';

Currently implemented only on ~EC computers.

PROCEDURE UNITCLEAR (UNITN.NBEH);

UNITCLEAR cancels all I/Os to the specified unit and ~eset5 ~ hardware to its power−up state. Sets ICRESULT non−zero if unit ~S ~ present.

FUNCTION BLOCKREAD (FILEID, ARRAY, BLOCKS, [RELBLOCK]) : INTEGER; FUNCTION BLOCIGIRITE C FILEID, ARRAY, BLOCKS, [RELBLOCK]) : INTEGER; { sequential }

Thfse functions return an INTEGER value equal to the number of blocks of data actually transferred. The FILE must be an untyped fji~ (i.e. FILEID: FILE;). The length of ARRAY sh~ld 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 n~inber of blocks you want transferred. RELBLOCK is the bloc~umber 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 fran the directory. The unit will go off−line if the device is not block structured.

The CRUNCHoption 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 E~F THEN...). EOLN and EOF return false after the file specified is RESET. They both return true on a closed file. When EG~ (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 accarinodate the PUT or WRITE, EOF(FILEID) will return true.

FUNCTION IORESULT : INTEG~;

After any I/O operation, IORESULT contains a~ INTEGER value corresponding to the values given in Table 2. If the compiler is allowed (i.e. (`$1−') has not been used), it will generate checks after each I/O operation, causing the progran to get a run−time error on any bed I/O operation. These are not generated any time after any UNITREAD or UNITvIRITE.

PROCEDURE GET. (FILED); − PRCCEDURE PUT (Ffl..EID);

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 CF <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 <t:';e>.

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 (Ffl..EID) puts the contents of FILEID~ into the file at the location of the current file pointers and then updates those pointer:. The actual physical disk access may not occur until the next time the physically, associated block of the diskis no−longer considered the current ~rking block. The kinds of operation which tend to force t~ block to be written are: a S~K 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} (FU.EID, SOURCE);

These procedures may be used only on TEXT (FLE CF CFiA~) or INTERACTIVE files for I/O. If `FILEID, ` is omitted, INP~JT or OLT~" (whichever is appropriate) is assLrned. A REP~D(STRING) will re~.i up to and not including the end−of−line character (<a carriage return>) ~ leave EOLN(FILEID) true. This means that any subsequent REACs of' STRING variables will return the null string until a READLN or READ(ch~aracter) is executed.

Page 12L~



There are t~ree files of ty~ 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 progr~riner complete control of the response to user typing. OUTPUT allows the user to halt or flush the output.

PR~EDURE PAGE (FILEID);

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

PR~EDURE SEEK C FILE ID, INTEGER);

This p−ocedure 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 ni.xnbered from 0. A GET or PUT must be executed between S~K calls since two S~Ks in a row may cause unexpected, unpredictable junk to be held in the window and associated buffers. Sets EOF' and EOLN to false.

move optimization in the section on MOVELEFT. The notes about M0VEL~T also apply to FILLCHAR.

The intrinsic SIZEOF (Section 2.1.6) is meant for use with tf~se intrinsics; as it is convenient not to have to figure out or remember the number of bytes in a particular data structure. (Which may chare at the progranrners whim.)

Page 125



2.1.3. Miscellaneous Routines



********** ********,********

* MISCELLANEOUS ROUTINES * * Section 2. 1.3 * ********** **********

Version 11.0 February 1979

FUNCTION SIZEOF (VARIABLE OR TYPE IDENTIFIER) : INTEGER;

This function returns the number of bytes that the "item" passed as a parameter occupies. SIZE~F is particularly useful for FILLCHAR and MOVExxxx intrinsics.

FUNCTION L~ (NUMBER) : REAL;

This function returns the log base ten of the NUMBER passed as a parameter.

PR~CEDURE TIME (VAR HIWORD, LG~IORD: INTEGER); (* may not be implemented in all machines *)

This procedure returns the current value of the system clock. It is in 6Oths 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 W.4ORD may be negative since the time is represented as a 32−bit unsigned number.) ~th HIWORD and LG~ORD must be VARiables of type INTE~R.

FUNCTION F~1R~TEN (E)~ONENT: INTEGER) : REAL;

This function returns the value of 10 to the E)~ONENT power. EXPONENT must be an integer in the range 0. .37. PROCEDURE MARK (VAR HEAPPTR: INTEGER) PROCEDURE RELEASE (VAR HEAPPTR: ~INT~ER); These procedures are used for returning dynamic memory allocations to the system. HEAPPTR is of type ~INTEGER. MARK sets HEAPPTR to the current top−of−heap. RELEASE sets top−of−heap pointer to HEAPPTR.

PROCEDURE HALT;

This procedure generates a HALT opcode that, when executed, causes a non−fatal run−time error to occur. At this point in execution, the Debugger is invoked, therefore, if the Debugger is not in core when this occurs, a fatal run−time error, #114, will occur.

PROCEDURE GOTO)C1 C XCOORD , YCOORD: INTEGER);

Page 127



This procedure sends the cursor to the coordinates specified by (XCOORD, YCOORD). The upper left corner of the screen is assiined to be (0,0). This procedure is written to default to a Datanedia−type tenninal. If

your syst~n uses other than a Datanedia or Terak 8510a, you will need to bind in a new GOTOXY using the GOTOXY package described in Section ~4.7.

FUNCTION M~tAVAIL: INTEGER;

This fiz~ctjon returns the nunber of words currently between the top−of−stack and top−of−heap. This can be interpreted as the ano~.r~t of menory available at that time. ~e must take into consideration the size of evaluation stacks, and error−procedure cal is.

Page 128



********** *******,*******a* * .0 0 P S * * Section 2.1.14 * ********** **********

Version 1.5 ~ptember 1978

Out Of Place Section

Page 129



2.1.5. Character Array Manipulation

Version I.5, September 1978

CAUTION



These intrinsics are all byte oriented. Use them with care. Read the descriptions carefully before trying them out as no range cI~cking of any sortis performed on the parameters passed to these routines. The programmer shzuld 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 fu~ction 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: :: ` THE TERAK IS A MEMBER CF THE PTERODACTYL FAMILY.';

SCAN (−26,:':' ,DEM[30]); will return −.26 SCAN(100,<>' .`,DEM); will return 5 SCAN(15,:' `,DEMrOn; will return 8

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

Page 131



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

Soaie implementations of these intrinsics may do optimization of such a move for the sp~ific hardware involved.

In short: MOVEL~T starts at the left end of both arrays arid copies bytes traveling right. MOVERIQ~T starts at the right end of both arrays a~d copies bytes traveling left. The reason for having both of these is if you are ~rk1ng 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;

(*123l456789a123i~567S9b123~67S9c*) ARAY: THIS IS THE TEXT IN Th'IS ARRAY
MOV~I~T(ARAY[1O~,ARAY[1~, ~); ARAY: UiE TEXT INE TEXT IN THIS ARRA~.'~
MCVELEFT(ARAYt1LARAYE3),10) ARAY: HEHEHEHEHEHETEXT IN THIS ARRAY

MCVELEFT(ARAYt23~,ARAY~2],8); ARAY: HIS ARRAYENETEXT IN THIS ARRAY

PRCCEDURE FILLCHAR C DESTINATION, LENGTH, CHARACTER);

This procedure takes a ~subscripted) PAC~D ARRAY OF CHARACTERS and fills it with the rnrnber (LENGTH) of CHARACTERS specified. This can be done by:

ACC~ :: <character expresS:Ofl> McVELEyT(A~C:,A:1J,LENGTH–i ;

but FILLCHAR is twice as fast, as no me~cry reference :s riee~e: f: source.

See the note about move optimization in the section on W)VEL~T. The notes about MOVELEFT also apply to FILLCHAR. The notes about MOVELEFT also apply to FILLCHAR. The intrinsic 5IZE~ (Section 2. 1.6) is meant for use with t~es~ intrinsics; as it is convenient r~t to have t~ figure out or rr~ ~r the n~znber of bytes in a particular data structi.~e. Cwhich ~iay ~nar~;e at the prograt~n~ers whim.)

Page 132

2.1. Intrinsics

	Intrinsics Parameters	115
2.1.1.	String Intrinsics	117
2.1.2.	Input and Output Intrinsics	121
2.1.3.	Miscellaneous Routines	127

2.2. Differences Between UCSD Pascal and Standard Pascal

133

********** ********** *
DIFFERENCES BEP,4EEN UCSD PASCAL AND STANDARD PASCAL* * Section 2.2 *
********** **********

Version 11.0 February 1979

This section is a simnary and quick referrence guide which notes the areas in which UCSD Pascal differs fran Standard Pascal, and refers the user to the appropriate docunents 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 Nik].aus Wirth (Springer−Verlag, 1975).

Many of the differences lie in the area of FILES and I/O in general. It is recomnended 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 STAT~1ENTS

Jensen and Wirth on page 31, state that if there is no label equal to the value of the case statenent 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:

PROGRAN FALLTHROUGH; VAR CH:CHAR; BEGIN CH: :`A'; CASE CH OF `B': WRITELN(OUTPTJr,'HI THERE'); `C': WRITELN(OUTPUT,'THE CHARACTER ISA ``C'") END; wRITELN(O~JrPlrr, `THAT''S ALL FOLKS'); END.

Page 133

134

2.2.2 COMMENTS

Ih~ UCSD Pascal compiler recognizes any text appearing between either the symbols ~(*~ and fl*)tt or the symbols fl{PI and "}" as a comment. Text appearing between these symbols is ignored by the compiler unless the first character of the carrnent is a dollasjgn, in ~.Eiich case the comment is interpreted as a compiler control coent. See section 1.6 "Pascal Compiler" for details on compiler con&ol comments.

if the beginning of the comment is delimited by the "(" symbol, the end of the CaTlnent must be delimited by the matching n*)it symbol, rather than the "1" symbol. when the comment begins with the "f" symbol) the c~iTner1t continues until the matching "}" symbol appears. This feature allows a user tO "Ca~nrnerit out" a section of a progr~n ~.&iich itself contains ccnrnents. For example:

(XC? :: XC? + 1; C' ADJUST FOR SPECIAL CASE... *)

The compiler does not keep track of nested cannei t.~. Wh~r~ a comment symbol is encountered, the text is scanned for the rnatchlr~ c~inent symbol. The following text will result in a stntax error:

(* THIS IS A C~MENT (* NESTED CCMM~~~~ ~. E~JD CF FIRST C~1MENT ~; ~error here.

2.2.3 DYNAMIC M~1ORY ALLOCATION

The standard procedure DISPOSE defined on page 158 of Jensen and Wirth is not implenented in UCSD Pascal. Fbwever, 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 ~IS?CSE a~ one cannot explicitly ask that the storage occupied by or.e particular variable be released by the system for other uses.

The current USD implementation allocates stcrs~e for variables created by use of the standard procedure NE~ in a ~ structure called the "heap". The following program is a 3~p~ demonstration of how MARK and RELEASE can be used to :har.g~ in of the heap.

PROGRAM S~1ALLHEAP;

TYPE PERSOt~: RECORD NAME: PAC?~D ARRAYCO.. 15~ OF CHAR; ID: INTEGER END;

Page i3Z~



VAR F: ~PERSct~;(* "~" means elpojnter to" as defined in J&W *) HEAP: ~INTEGER;

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

The above program first calls MARK to place the address of the current top of heap into the variable HEAP. HEAP must be declared to be a pointer to an INTEGER. The parameter supplied to MARK must be a pointer to an INTEGER. This is a UCSD restriction. This is a particularly handy construct for deliberately accessing the contents of memory which is otherwise inaccessable. Below is a pictorial description of the heap at this point in the program's execution:

TOP CF HEAP −−> _____ <−−−. HEAP

contents of heap at start of program

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

TOP CF HEAP −−−>

I

I

contents of heap at start of program

After the RELEASE the heap is as follo~:

contents of heap at start of program

TOP ~F HEAP −.−−>

<−−− HEAP

<−−− HEAP

Page 135



&~ce the progra~ no longer needs the variable P~ and wishes to "release" this tu~nory space to the systen for other uses, it calls RELEASE which resets the top of heap to the address contained in the variable HEAP.

If the above sanpie progr~ had made a series of calls to the standard procedure NEW between the calls to MARK and RELEASE, the storage occupied by several variables ~u].d have been released at once. Note that due to the stack nature of the heap it is not possible to release the tn~nory space used by a s~r~gle item in the middle of the heap. It is for this reason the use of MARK and RELEASE can only approximate the f~z~ction of DISPOSE as described in Jensen and Wirth.

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

2.2.L~ EOF(F)

To set EOF to TRUE for a textfile F being us~d as an input t'il! from the CONSOLE device, the user must type the EO~ character. The EOF character can be altered by a suitable re~onfig~ration of the system variable SYSCOM.CRTI~fO.EOF using ~ For i~urther information concerning system configuration ansi the SETUP pr~ram se~ Section 44.3.

If F is closed, for any FILE F, EOF(F) will return the value TRUE. If EOF(F) is TRUE , and F i~ a FILE of type TEXT, ECLN(F) is also TRUE. After a RES~(F), EOF(F) is FA~E. If EOF(F) becomes TRUE during a ~T(F) or a READ(F,...) the data obtained thereby is nct valid.

When a user progr~ starts execution, the system perf~r~s a RESET on the predeciared files INPtJ1', CL~PJ~, ar.~ KEY3C~.R~. See secticn 2.2.11 READ for further details concerning the pre~ecare~ fi1.~ ~YBOARD. As defined in Jensen and Wirth, ECF and EOLN ~y ief~: ~ refer to the file :NPUT if no file identifier ~ soec~.fie~.

2.2.5 EOL~(F)

EOU4(F) is defined only If the <type> of the window ~rari~b1e~ r, is of type CHAR. EOLN becomes TRUE only after reaoing the end~ line character. The end of line character is a carriage return. the example progr~n below, care must be taken as regards when the carriage return is typed while inputing ~ta:

Page 736



PROGRAM ADDLINES; VAR K,SLM:INTEGER;

BEGIN WHILE NOT EOF(INP1IT) DO BEGIN Slt4:=O; READ(INPUT,K); ~4HILE NOT EOLN(INP!JT) DO BEGIN SLP1:~SUM+K; REA~D(INPUT,K); END; WRITELN(OUTPUT); WRITELN(OUTR.Tr,'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 OUTPL.TI' 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 Jens'en and Wirth. This type and its use is demonstrated in the sample program below:



(`$1−') PROGRAM FlL~EMO;

VAR BLOCKN~1BER, BLOCKSTRANSFERRED: INTEGER; BADIO: BCOLEAN; G,F: FILE; BLFFER: PACED ARRAY[O. .511) OF CHAR;

(* This progran rea's a diskfile called `ZOURCE.DATA' and copies the rile into another diskfile cal].ed `DESTINATION' using untyped files and the intririsics BLOC?~EAD and BLOCK~4RITE `)

BEGIN BADIO: FALSE; RESET(G, `SO(JRCE.DATA'); R~1RITE(F, `DESTINATION'); BLOCKNUMBER: =0; BL~KSTRA~.N.SFERRED: :BLOCKREAD(G,BIFFER, 1 ,BLOCKNUMBER); WHILE (NOT EOF(G)) AND (IORESULT:O) AND (NOT BADIO) kND (BLOCKSTRANSFERRED:1) DO BEGIN BLOGKSTRANSFERRED: =BLUC~GJRITE(F, BUFFER, 1 , BLXKNUMEER); BADIO:=((BLOCKSTRAN~ERRED<1) CR (IORESULTOO)); BLOCKN1J'~BER; =BLOCKNt)IBER+I; BLOCKSTRANSFERRED::BLOCKREAD(G,BUFFER, 1 ,BLOCKNUMBER); END; CLOSE C F, LO~() END.

The t~ files which are declared and used in the ~cve sampe progran are both untyped files. An untyped file F can be thought of ~ a file without a window variable r to which all I/C must be accomplished by using the

functions BL~KREAD and BLCCK~RITE. Note that any number of blocks can be transferred using either ?LOC~READ or BL~KWRITE. The functions return the actual number of bloc~3 read. A somewhat sneaky approach to doing a quick transfer ~uld be:

WHILE BLOCKI~RITE(F,BUFFER,BLOCKREAD(G,BUFFER,BUFBLCCKS))>C 10

This is, however considered unclean. The progran above has been compiled using the (*$I_*) Compile Time ~tion, thereby requiring that the function IORESIJLT and the number of blocks ~ransferreo be checked after each BLCCKREAD or BL0C~RITE in order to detect any I/C errors that might have occurred.

Page 138



C. RANDQI 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 t~ paraneters, the first being the file identifier, and the second, an integer specifying the record ni.znber to which the window should be moved. The first record of a structured file is nuTlbered record 0. The following sample program demonstrates the use of SEEK to randomly access and update records in a file:

PROGRAM RAN~1ACCESS; VAR RECNU'1BER: INTEGER; CH: CHAR; DISK: FILE OF RECORD NAME: STRING[20); DAY,MONTH,YEAR: INTEGER; ADDRESS: PACI~D ARRAY[O. .L~9] OF CHAR; ALIVE: BOOLEAN END;

BEGIN

RESET(DISK, `RECORDS.DATA');

WHILE NOT EOF(INPtJ~) DO BEGIN WRITE(OIrTPUT,'Enter record ntrnber −−>`); READ(INPJI~, RECN(1~1~R);

SEEK(DISK, RECNU'IBER); GET (DISK);

WITh DISK~ DO BEG IN WRITELN(OITrPLIT,NAME,DAY,MONTH,YEAR ,ADDRESS); WRITE(OUTPtJI~,'Enter correct name −−−>`); READLN(INPUT,NAME);

END;

(* M~t point the window back to the record since GET(DISK) ath'arices the window to the next record after loading DISK~

SEEK(DISK,RECN1J~ER); PW~(DISK); END; END.

Page 139



Attempts to PUT records beyond the physical end of tile 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.) S~K always

sets EOF arid EOLN to FALSE. The subsequent GET or PUT will set these conditions as is appropriate. See Section 2.1.2. D. READ AND WRITE FRC~I 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 STAT~tENTS

UCSD has a more limited form of GOTO statement than is defined as the standard in Jensen and Wirth. UCSD'z GOTO statement prohibits a GOTO statement to a label which is not within the sane procedure block as the GOTO statement itself. The exanpies 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 paraneter 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 assigrrnent to the FUNCTION identifier is made prior to the execution of the EXIT statement. Below is an exanple of the use of the EXIT statement:

PRGORft~ EXITDDIO; VAR T: STRING; CN: INTEGER;

PR~EDURE Q; FORWARD;

PR~EDURE P; BEGIN READLN(T); WRITELN(T); ~ T~1~='#' THEN EXIT(Q); WRITELN('LEAVE p1); END;

Page 1L~O



PROC ~URE Q; BEGIN F; WRITELN('LEAVE Q'); END;

PROCEDURE R; BEGIN IF CN <~ 10 THEN Q; WRITELN('LEAVE R'); END;

BEGIN CN:~O; WHILE NOT E~F DO BEGIN CN:CN+1;

WRITELN; END; END.

If the above program were supplied the fo1lowir~ 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) state~ent 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.

Page un



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 exanpie, one or both of the procedures P and Q declared and opened sane local files, an implicit CLOSE(F) is done when the EXIT(Q) statenent is executed, as if the procedures P and 0 terminated normally.

The EXIT statement may also be used to exit a Pascal program by EXIT(PRO~AM) or EXIT(prograннane).

The creation of the EXIT statement at LJCSD was inspired by the occasional need for a straightforward means to abort a complicated an~ 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 state!nentis, nevertheless, discour~ed.

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 ~`crd PACKED. For example, consider the following declarations:

A: ARRAYEO..9] OF CHAR;

B: PACKED ARRAYEO..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 t~ 8 bit characters. In this mann~er each element of the PACKED ARRAY B is S bits long. PACKED ARRAYs need not be restricted to arrays of type CHAR, for example:

C: PACKED ARRAYCO. .1) CF 0.. 3;

D: PACKED ARRAYE1..9) OF SET OF O..15;

02: PACKED ARRAYCO. .239,0. .319) CF BOOLEAN;

Each element of the PACKED ARRAY C is only 2 bits long, sirce only 2 bits are needed to represent the values in the range C..3. Therefore C occupies only one 16 bit ~rd of memory, and 12 of the bits in that word are unused. The PACKED ARRAY 0 is a 9 word array, since each element of 0 is a SET which can be represented in a minim~.zn of 16 bits. Each element of a PACKED ARRAY OF BOOLEAN, as in the case of 02 in the above example, occupies only one bit.

Page 1~i2



The following 2 declarations are not equivalent due to the recursive nature of the compiler:

E: PACKED ARRAYEO..9] OF ARRAYEO..3] OF CHAR;

F: PACKED ARRAY{0..9,O..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 140 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[O..9] OF PACKED ARRAY[O..3] OF CHAR;

or as

E: ARRAYEO..9] OF PACKED ARRAY[O..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;

0 will be an array which occupies 14 16 bit words.

Packing never occurs across word boundaries. This means that if the type of the element to be pacled 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 CF 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".

Page 1143



A PA(~CED ARRAY CF CHAR may be output with a single write statement:

PR~RAM V~YSLICK; VAR T: PACKED ARRAY(O..10] CF CHAR; BEGIN T:'HELLO THERE'; WRITELN(T); END.

Initialization of a PACKED ARRAY OF CHAR can be accomplished very efficiently by using the (JCSD intrinsics FILLCHAR and SIZEOF:

PRCGRAM FILLFAST; VAR A: PACKED ARRAYCO.. 10] CF CHAR; BEGIN FILLCHAR(ACO],SIZEOF(A),' `); END.

The above sample program fills the entire PACKED ARRAY A with blanks. For further doc~inentation on FILLCHAR, SIZEOF, and the other UCSD intinsicz see section 2.1.5 "OiARACTER ARRAY ~1ANIPUL.ATICN INTRI~ ICS". B. PACKED REC0R~

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: O..31; B: BOOLEAW 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 sane word.

In much the sane manner that PACKED ARRAYs can be multidimensional PACKED ARRAYs, PAQ'~D RECORDS may contain fields wriich 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 t~o declarations are not equivalent:

VAR A:PACKED RECORD VAR B:PACKED RECORD C:INTEG~; C:INTEGER; F:PACKED RECORD F:RECORD R: CHAR; R:CHAR; K: BOOLEAN K:BOOLEAN END; END; H:PACKED ARRAYtO..3] OF Q4AR H:PACKED ARRAYEO..3) OF CHAR END; END;

Page 11L~



As ~ath 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 ~ample, 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 ca~ 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 amourig 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: CM: PACKED ARRAY[O..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 1~4 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 PARP~NETERS

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 ~~CEDURES

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.

Page 1~45



2.2.9 PARAMETRIC ffiCCEDURES AND FUNCT~NS

UCSD Pascal does not support the construct in which PROGEDURES and FUNCTIONS may be declared as formal parameters in the parameter list of a ~~EDURE or FUNCTION.

See Section 5.9 for a revised syntax diagram of <parameter–. list>.

2.2.10 PROGRAM HEADINGS

Although the UCSD Pascal canpiler will permit a list of file parameters to be present following the program identifier, these parameters are ignored by the canpiler and will have no affect on the program being ocinpiled. As a result the following t~ program headings are equivalent:

PROGRAM D~1O(INPUT,OUTPtrr); and PROGRAM D~iO;

With either of the above program headings, ~ user program will have three files predeclared and opened by the system. These are: INPUT, OUTPUT, and K~fBOARD and are defined to be of (type> INTERACTIVE. It 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 REPLDLN

Given the followir declarations:

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

the zt.atement READ(F,CH) is defined by Jensen and Wirth on page 35 to be equivalent to the t~ statement sequence:

CH::F~; J & W GET(F); method

In other ~rdz, the standard definition of the standard procedure READ requires that the process of opening a file load the "window variable" FA with the first character of the tile. In an interactive prograninig 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 ~.culd "tang" until a character was typed, whether or not the program performed any input operations at all. In order to overcane this problem, UCSD Pascal defines an additional file <type> called INTERACTIVE. ~claring a file F to be of (type> INTERACTIVE is equivalent to declaring F to be of tyje TEXT, the difference being that the definition of the statement READ(F,~) is the reverse of the sequence specified by the standard

Page 1L~6



definition for files of <type> TEXT: i.e.

This difference affects the way in which EOLN must be used within a progran when reading fran a textfile of type INTERACTIVE. As in section 5, EOLN becomes true only after reading the end of line character, a ca~riage 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 exanple, the left fragment is taken from Jensen and Wirth; only the RESET and RE~4RITE statements have been altered. The program on the left will correctly copy the textfile represented by the file X to the file Y. The progran 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 progran in order to produce the desired result.

PROGRAM JAN~; VAR X,Y:TEXT; CH: CHAR; BEGIN RES~(X, `SOURCE.IEXT'); REWRITE(Y, `SOMETHING.TEXT');

WHILE NOT ECF(X) DO BEGIN WHILE NOT EOLN(X) DO BEGIN READ(X, CH); WRITE(Y,CH); END; READLN(X); WRITELN(Y);

END; C LOSE (Y LOCK)

END.

Note that the textf lies X and Y to be opened by using the tJCSD extended procedures RESET and RE~i1RITE.

PROGRAM (JCSDVERSION; VAR X, Y: INTERACTIVE; CH:CHAR; BEGIN RESET(X, `CONSOLE:'); RB~1RITE(Y, `SOMETHING. TEXT'); READ(X,CH); WHILE NOT E'F(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.

in the above t~ programs had form of the standard

GET(F); CH: :F~;

UCSD method

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.IEXT". Likewise, the textfile X could have been a diskfile instead of caning 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,

Page 1147



and is always defined to be of <type> INTERACTIVE. The statement REA~D(INPUT,CH) ~iere CH is a character variable, will echo the character typed fran the CONSOLE back to the CONSOLE device. WRITE stat~nents to the file O~.T1'PUT will, by default, cause the output to appear on the CONSOLE device. The

file I~BOARD is the non−echoing equivalent to INPUT. For ecemple, the t~ statements

RE~D(~YBOP.RD,CH); WRITE(OUTpur, CH);

a−e equivalent to the single statement READ(INPUT,CH).

Reading the type int~er causes preceding blanks and end−of− lines to be flushed t~til a non−blank character is observed. Reading the type BOOLEAN is not implemented.

For more doc~.xnentation regarding the use of files see sections: 2.2.6 "FILES" 2.2.14 "EOF" 2.2.5 "EOLN" 2.2. 17 "WRITE AND WRITEL.N" 2.2. 72 "RESET"

See section 2. 1.2 "INPUT/OUTPUT INTRI~ICS" for more details on the UCSD irttrinsics.

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 r~uther 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 irnplenentation 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 t~ 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 t~ paraneters, 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 morei information on this use of RESET.

Page 1~8



2.2.13 RB4RHE(F)

The standard procedure RB4RITE is used to open arid create a new file. R~1RITE has t~ parameters, ~ (`i~;, 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 RF~4RITE(F,'Sa4EHFO.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 "Sc?1EIl~FO.TEXT". See section 2. 1.2 "INPUT/OUTP(.TI INTRINSICS" for further docu~entation regarding the use of REWRITE to open a file.

2.2. 1L~ SEQIENT PROCEDURES

END;

SEGMENT PROCEDURE ~O; SEIMENT PROCEDURE THREE; B~IN

END; BEGIN (* SEGMENT THREE; PRINT(`SEGMENT END;

J~CCEDURE PRINT; BEGIN WRITELN(OUTPT,T); END;

BEG IN ThJO; WRITELN('I' `M EONE'); END.

Ihe concept of the SEGMENT PROCEDURE is a UCSD extension to Pascal, the primary purpose of which is to allow a progranrier 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 snail interactive computers at one time. The following is an example of the use of SEGMENT PROCEDURES:

PROCRP~M SEGIENTDEMO;

(* GLOBAL DECLARATIONS GO HERE *)

PROCEDURE PRINT(T:STRING); FORWARD;

S~1ENT PROCEDURE ONE; BEGIN PRINT('SEQ.IENT NLt~1BER ONE');

ONE; PRINT('SEGMENT NUMBER THREE');

NUMBER ThIC *)

NUMBER TWO');

Page 149



The above program will give the following output:

S~1ENT NUMBER ONE S~MENT N(JIBER THREE SEQ4ENT NUMBER NO I'M DONE

For further doc~znentation on S~1ENT PR~EDURES, their use and syntax governing their declaration, see Section 3.3 – `SEGMENT PRcCEDURE~.

2.2.15 SETS

UCSD Pascal supports all of the constructs defined for sets on pages 50–51 of Jensen and Wirth. Sets (of ent.~neration 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 14080 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..149, 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 comparability, implies that the comparisons and operations on the sets S and R in the following program are legal:

PRWRA~I SETCcM PARE; VAR 5: SET OF 0..149; R: SET CF 1.. 100;

BEGIN S:= C0,5,10,15,20,25,30,35,140,145); R.= f1O,2O,3O,L~O,5O,60,70,80,90], 5 = R THEN WRITELN('... oops ...`) ELSE WRITELN('sets ~rk'); S :: S + R; END.

In the following example, the constr~~ct I = J is not legal since the t~ sets are of two distinct underlying types.

Page 150



PROGRAM ILLEGALSE~; TYPE STUFF:(ZERO,ONE,IWO); VAR I: SET CF STUFF; J: SET CF 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 PAQ(ED ARRAYs CF 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 [J 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 character ~*)

Note that a STRING variable has an absolute maximum length of 255 characters. Assig~ents to string variables can be performed using the assignment statement, the UCSD STRING intririsics, or by means of a READ statement:

TITLE::' THIS IS A TITLE

or

RE~DLN(TITLE);

or

NAME:: COPY(TITLE,1,20);

The irdividual characters within a STRING are indexed frai' 1 to the LENGTH of the STRING, for example: TITLEI1]:: `A';

Page 151



TITLEt LENGTH(TITL.E)):= `Z';

A variable of type STRflC may not be indexed beyond its current dynanic LENGTH; be~re of strings of length zero! The following sequence will result in an invalid index run time error:

TITLE:: `12314'; TITLEt5):: `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. t.kilike 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 CaI PARESTRIN~; VAR S: STRING; T: STRING[~4O]; BEGIN S: `SOMETHING'; T:: `SO'IETHING BIGGER'; IF S :T THEN WRITELN('Strings do not work very well') ELSE IF S > T TJEN WRITELN(S,' is greater than `,T) ELSE IF .5 < T THEN WRITaN(S,' is less than `,T); IF S = `SOMETHING' THEN ~4RITEL.N(S,' equals `,S); IF S > `SP~METHING' THEN WRITELN(S,' is greater than SAMETHI~'); IF S = `SOMETHING THEN WRITELN('BLANKS DON"T COUNT') ELSE WRITELN('BLANKS APPEAR TO :~AKE A DIFFERENCE'); S:='XXX'; T:='ABCD~'; IF S > T THEN WRITELN(S,' is greater than `,T) ELSE WRITELN(S,' is less than `,T); END.

Page 152



The above program produces the following output:

SOMETHING is less than SOMETHING BIGGER SCMETHING equals SOMETHING SOMETHING is greater than SAMETHING BLANKS APPEAR TO MAKE A DIFFERENCE XXX is greater than ABCDEF

One of the most carrnon uses of STRING vari~les in the UCSD Pascal syste~n is reading file names fran the CONSOLE device:

PRCX3RAM LISTER; VAR BUFFER: PACF~D ARRAYEO..51fl OF CHAR; FILENAME: STRING; F: FILE;

BEGIN WRITE(tEnter filename of the file to be listed −−−>`); READLN(FILENAME); RESET(F,FILENAME); WHILE NOT EOF(F) ~ 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 32 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 IRUE or FALSE when writing out the value of a BOCLEAN variable.

Page 153

For a description of WRITE statements of variables of type STRING see Section 2.1.1 "STRING INTRI~ICS".

U~D's WRITE and WRITELN do support the writing of entire PAC~ ARRAYs OF CHAR in a single WRITE statement:

VAR BUFFER: PACKED ARRAYCO.. 10] OF CHAR; BEGIN BUFFER:: `HELLO THERE'; C' contains exactly 11 characters `) WRITELN(OlTrPUT, BUFFER); END.

The above construct will ~rk only if the ARRAY is a PACKED ARRAY OF CHAR. See section 2.2.8 PACKED VAR IABL~ for further information.

The following progr~n demonstrates the effects of a field width specification within a WRITE statement for a variable of type STRING:

PROGRAII WRITESTRIN~; VAR S:STRING;

BEG IN S::'THE BIG BRO.4N FOX JU4PED...'; WRITELN(3); WRITELN(S: 30); WRITELN(S: 10); END.

The above progran will produce the following output:

THE BIG BRO~1N 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 n~.xnber of characters written is equal to the dyn~nic length of the string. If the field width specified is longer than the dynanic length of the string, leading blanks are inserted anc written. If the field width is snaller than the dynanic length of the string, the excess characters will be truncated on the right.

2.2.18 IMPLEMENTATION SIZE L~ITS

The following is a list of maximun size limitations imposed upon the user by the current iznplementat ion of UCSD Pascal:

Page 15Z~

1. Maxixm.zn rn.znber of bytes of object code in a PROCEDURE or FUNCTION is 1200. L~ca1 variables in a PROCEDURE or FUNCTION can occupy a maximtxn 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.

~. Maximum number of SEQ~1ENT PROCEDUREs and SEQ1ENT FIJNCTIONs is 16. (9 are reserved for the Pascal system, 7 are available for use by the user program)

5. Maximi.xn ntznber of PROCEDURES or FUNCTIONs within a segment is 127.

2.2.19 EXTENDED CcMPARISONS.

UCSD Pascal allows and <> comparisons of any array or record structure.

2.2.20 LONG INTEGERS.

UCSD Pascal allows integers of up to 36 digits. See section 3.3.3 for details r~arding long integers.

2.2.21 UNITS.

UCSD Pascal now supports the modularity concept of UNITs. See section 3.3.2 for details regarding UNITs.

2. 2.22 SUMMARY OF UCSD INTRINSICS

INTRINSIC SECTION # DESCRIPTION

BLOCKREAD 2.1.2 Function which reads a variable number of blocks from an untyped file. BL~CKWRITE 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..

Page 155



GOTOXY 2. 1 .3 Procedure used for cursor addressing %`k~ose two parameters X and Y are the colutn and line n~inbera on the screen where the cursor is to be placec. FILLCHAR 2.1.5 Fast procedure for initializing PACKa) ARRAYs CF CHAR.

HALT 2. 1.3 Halts a user prograi which may result in a call to the interactive Debugger. IDSEARCH Routine used by the Pascal c~npiler, and the PDP–l 1 assen'bler. INSERT 2.1.1 STRING intrinsic used to insert characters in STRING variables. IDRESULT 2.1.2 Function returning the result of the previous I/O operation. (See Table 2 for a list of values) LENGTH 2. 7. 1 STRING intrinsic which returns the d~~ic length of a STRING variable.

MARK 2.1.3 Used to mark the current top of the heap in ~ynanic memory allocation.

M~1AVAIL 2.1.3 Returns nunber of ~~crds between Heap and Stack.

MOVELEFT 2.1.5 Low level intrinsic for rr~ving mass amounts of bytes.

POVERIGHT 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 a~ existing file.

P03 2.1.1 STRING intrinsic returning the position of a pattern in a STRING variable.

P~4RcFTEN 2.1.3 Function which returns as a REAL result the ntznber 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 randcii accessing of records withing a fie.

S~E~ 2.1.3 Function returning the nunber of bytes allocated to a variable.

STR 2.1.1 Procedure to convert long integer into string.

Page 156



TIME 2.1.3 Function returning the time since last bootstrap of systen. (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 peripher~i device.

UNITREAD 2.1.2 Low level intrinsic for reading from a perip~iera1 device.

UNIThIAIT 2.1.2 Low level intrinsic for waiting until a peripheral device has completed an I/O operation.

UNITi~RITE 2.1.2 Low level intrinsic used for writing to a peripheral device. device.

Page 157



2. The UCSD Pascal Language

2.1. Intrinsics 115
2.1.1. String Intrinsics 117
2.1.2. Input and Output Intrinsics 121
2.1.3. Miscellaneous Routines 127
2.1.5. Character Array Manipulation 131
2.2. Differences Between UCSD Pascal and Standard Pascal 133

3.1. Drawline

159

********** ********** * IMPL~4ENTATION NOTES − DRA~'ILINE * * Section 3.1 * ********** **********

Version 11.0 February 1979

The DRAWLINE intrinsic uses an incremental technique toplot line segments on a point−addressable matrix. The algorithm guarantees a best (least squares) approximation to the desired line. In general this approximation is not unique. DRAWLINE may pick different representations for a line depending on the starting point. (This could be corrected by always starting at the sane end of the line.) No range checking is performed on paraneters passed to this intrinsic.

The algorithm is essentially the one described in Newnan and Sproul, Principles of Interactive Computer Graphics as the Digital Differential !~nalyzer. It has been modified to perform only integer arithmetic. Pascal source code is included below. The procedure first determined whether the line will be more horizontal or vertical. In the discussion below, we assixne the horizontal case; vertical is similar.

There will be DELTAX points plotted with horizontal increment of 1 each. The vertical increment will be ABS (DELTAY / DELTAX) <: 1. The Y coordinate arithmetic is scaled by DELTAX to eliminate fractions. An additional savings in execution time has been gained by maintaining the address of the previous point, and doing only addition and subtraction to reach the next point to be plotted.

The RADAR function is complicated as t~ intersecting lines may have no plotted points in cc~non. The detection condition is either (1) the computed point is TRUE, or (2) both the next horizontal and the next vertical points are TRUE. Condition (2) could be weakened: when the line is more horizontal, only the next vertical point need be checked.

Refer to Section 2.1.4 for a description of the parameter calling sequence.

A PASCAL implementation follows: PR~EDURE DRAWLINE (VAR RANGE: INTEGER; VAR SCREEN: SCREENTYPE; RO~WIDTh, XSTART, YSTART, DELTAX, DELTAY, INK: INTEGER);

VP~R X, Y, XINC, YINC, COUNT: INTEGER;

PR~EDURE DRAWDOT;

Page 159

160

PROCEDURE RADAR; VAR GOTIT: BOOLEAN; BEGIN GOTIT :: FALSE; COUNT :: COUNT + 1; IF SCREEN CY, X] THEN GOTIT : = TRUE ELSE (`WE MIGHT GO THROUGH A LINE') IF SCREEN (Y#1, Xl THEN GOTIT := SCRE~ EY, X+1]; IF GOTIT THEN BEGIN RANT := COUNT; EXI~(DRAWLINE) END; END (`RADAR');

(`LANDED ON ThE POINT')

```
BEGIN (`DRAWDCT') CASE INK OF 0 (`NONE'): 1 (*~.jf~TE*): 2 (`BLACK'): 3 (`REVERSE'): ~4
(`RADAR'): END (`CASE') END (`DRAWDOT');

EXIT (DRAWLINE); SCREEN tY, X] SCRE~ EY, Xl := SCREEN tY, XJ :: RADAR

(`THEY HAD NO TRUE; FALSE; NOT SCREEN [1,

BUSINESS HERE"

~fl.

PROCEDURE DCFO~(; (`MORE HORIZONTAL') VAR ERROR, I: INTEGER; BEGIN IF DELTAX = 0
THEN EXIT (DRAWLINE); (`THEY'RE GOING NGgHERE') ERROR : = DELTAX DIV 2; I :: DELTAX;
REPEAT ERRCR :: ERROR + DELTAY; IF ERR CR >= DELTAX THEN BEGIN ERROR :: ERROR −
DELTAX; Y :: Y + Y1NC END; X :: X # XINC; DRAWECT; I :: I − 1; UNTIL 1 = 0; END (`DCFORX');

PROCEDURE DCFCRY; (`MORE VERTICAL') VAR ERROR, I: INT~ER; BEGIN ERROR :: DELTAY
DIV 2; I :: DELTAY; REPEAT ERROR :: ERROR + DELTAX; IF ERROR >: DELTAY THEN BEGIN
ERROR :: ERROR ~` CELTAY; Y := Y + YINC;

X := X + XINC

END;
```

Page 160




```
DRAW DOT I :~ I − 1; UNTIL I 0; END (*DIJFORY*);

BEGIN (*DRAWLINE*) X := XSTART; IF DELTAX < 0 THEN BEGIN XINC : −1; I~LTAX : =
−DELTAX END ELSE XINC :: 1; Y :: YSTART; IF DELTAY < 0 THEN BEGIN YINC := −1; DELTAY :=
−DELTAY END ELSE YINC :: 1; COUNT :: 0; IF DELTAX >= DELTAY THEN DCFORX ELSE
DCFORY; IF INK = 4 (*RADAR*) THEN RANGE := COUNT; (*HIT ThE LIMIT GIVEN*) END
(*DR~1jJLINE*);
```

Page 161



3.2. File Formats

Version II.0, February 1979

Text files are of the format:



<1024 bytes> header page, information for editors. This space is reserved for use by the text editors, and is respected by all portions of the system. When a user program opens a TEXT file, and REWRITES or RESETS it with a title ending in ‘.TEXT’, the I/O subsystem will create and skip over the initial page. This is done to facilitate users editing their input and/or output data. The file– handler will transfer the header page only on a disk–disk transfer, and will omit it on a transfer to a serial device (*i.e.* transfers to PRINTER:, and CONSOLE: will omit the header page).

> <1024 byte pages> where a page is defined:
> <[DLE][indent][text][CR][DLE)[indent][text][CR]...[nulls]>

Data Link Escapes are followed by an indent–code, which is a byte containing the value 32+(# to indent). The nulls at the end of the page follow a [CR] in all cases, and are a pad to the end of a page. Because the compiler wants integral numbers of lines on a page. The Data Link Escape and corresponding indentation code are optional. In a given text file some lines will have the codes, and some won't.

Foto files are declared in PASCAL as follows:

```
TYPE SCREEN= PACKED ARRAY[0..239,0..319] OF BOOLEAN;
VAR FOTOFILE: PACKED FILE OF SCREEN;
```

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

Data files are up to the user.

Code files have one block of information which describes the code kept in the file. First is an array of 16 word pairs, the first word in the pair describes the block which starts the code of the segment which is numbered as the position in the array. The second word is the number of bytes in that segment. For example if the third word in the first block of a code file is an 8, and the fourth work is 1084, you now know that segment 1 of this code file starts on block 8 of the file, and has 1084 bytes of code. See Sections 3.4 and 3.5 for notes on codefiles.

Following this array is an array of arrays of characters. The array is an array of 8 character arrays which describe the segments by name. These 8 characters are those which identify the segment at compile time. Here again, the position in this array corresponds to the segment number.



Following the array of names is an array, again 16 words long, of state descriptors. The values in this array indicate what kind of segment is at the described location. The values for this array, at present, are: LINKED, HOSTSEG, SEGPRCC, UNITSEG, SEPRTSEG.

The remainder of the block, 144 words, is reserved for future use by later versions of the system. The format of the first block will most probably change completely for version II.0.

3.3.2. Linkage to Externally Compiled and Assemble Routines

Version I.5, September 1978

EXTERNAL CQ~1PILATION UNITS



The UCSD Pascal 1.5 system supports a facility for integrating externally compiled and assembled routines and data structures. Use of separately compiled structures allows the user to create files of frequently used routines. After a structure is compiled, the user adds it to a library, using the librarian. Files that reference that structure need not compile it directly into their code file, rather, the linker copies the existing code into the host code file. Separate compilation or assembly is supported in these areas: between portions of programs written in Pascal; between assembly language routines and Pascal hosts; and finally, between assembly language routines. Each of these areas is discussed in turn by the following sections.

3.3.2.1 PASCAL TO PASCAL LINKAGES – UNITS

A UNIT is a group of interdependent procedures, functi~ris, 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 IMPL~1ENTATION 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 comunicate with the UNIT while the IMPL~1ENTATION 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 liI~ MOVE, TURN, and PENCOLOR, which allow tI–e user to move the turtle and change colors. TURTLEGRAPHICS also employs DRAWLINE, an externally assembled procedure, to draw the lines and the turtle.

Page 167



A program that uses TURTLEGRAPHICS has no need for DRAWLINE, and, consequently, DRAWLINE is private to that UNIT.

PROGRAN DRAWPOLYGON; USES TURTLEGRAPHICS; VAR I:INTEGER; S ~E, NItISIDES: INTEG~;

BEGIN ININURTLE; (* Initialize the UNIT's variables *) WRITE('What size polygon?'); READLN(5IZE); WRITE('Ebw many sides?'); READLN(NUMSIDES); FOR I:rl TO NUISIDES DO BEGIN MOVE(SIZE); TURN(360 DIV NLHSIDES); END; END. ~(AMPLE A

A oro~–am must indicate the UNITs that it USES before the LABEL declaration part of the program. At the occurrence of a US~ stat~nent, 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. Nar~ie 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~ C NONE, WKITE, BLACK, REVERSE

PROCEDURE INITrURTLE; PROCEDURE TURN(RELAN~..E: Integer); PROCEDURE MOVE(REL.DISTANCE: Integer); PROCEDURE MOVETO(X, Y: Integer); PRCCEDURE TURNTOC ANGLE: Integer); PROCEDURE PENCOLOR C PCOLOR: TGCOLOR);

Page 168



IMPLEMENTATION

CONST

TERXSIZE 319; TER'1~IZE 239; RADCONST 57.29578;

TYPE

SCREEN Packed

Array [O..TERXSIZE, 0. .TERYSIZEJ of Boolean;

(~ Private variables *) TGXPOS: Integer; TGYPOS: Integer; TGHEADING: Integer; TGPEN: TGCOLOR;

I, J: Integer; S: SCREE~i;

(* Externally ass~nbled procedure ~) PROCEDURE E~AWLINE(S/ar 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, paraneters declared above *) BEGIN MOVETO(Round (TURTLEX + DIST*Cos(TURTLEANGLE/RAOCONST), Round(TURTLEY + DIST'Sjn(TURTLEANGLE/RADCONST));

Page 169



~CCEDURE MO VETO; VAR R: Int~er; BEGIN DRAWLINE(H, S, 20, 160+TURTLEX, 120−TURTLEY, X−TURTLEX, TURTLEY−Y, ~D(TURTLEP~4)); END;

PR~ED1JRE TURN;(' Public procedure, paraneters declared above *) BEGIN HEADING := (HEADING+RELANGLE) mod 360; END;

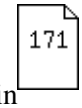
PR~EDURE TURNTO; BEGIN HEADING :: ANGLE; END; PR~EDURE PENCOLOR; BEGIN TGPEN :: PCOLOR; END;

END. (* End of unit `)

EXAMPLE B

Exanpie B is a skeleton for a TURTLEGRAPHICS UNIT. Note that the procedures MOVE, TURN, and INITrURTLE, and the TYPE TGCOLOR, are declared in the INTERFP1CE 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 ThWL~4ENTATION part, fo~ai parameter specifications are not repeated. A UNIT may also USE another UNIT, in ~ich case the USES declaration must appear at the beginning of the INTERFACE pert. In example C, PICTUREGRAPHICS indicates in the INTERFACE part that it USES TURTLEGRAPHICS. Note that the program USEGRAPHICS, which USES PICTUREGRAPHICS, indicates that it 1~ES TURTLEGRAPHICS before using PICTUREGRAPHICS. It is important that the INTERFACE part of TURTLEGRPLPHICS be defined before PICTUREGRAPHICS makes references to it, therefore this ordering is required.

Page 170



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 cc~npilation. This is due to the nature of generation of initialization code for FILEs.

PROGRAM USEGRAPHI~;

U4IT PICTUREGRAPHICS; INTERFACE USES TURTLEGRAPHI~; (~ TURTLEGRAPHICS is defined in the TYPE (* *syst~ library see section III below *) PVECTOR~ ~VECTO1~; VECTOR:RECORD DELHEADING: INTEGER; DELDISTANCE: INTEGER; PENDGiJN:BOOLEAN; NEXTVEC: PVECTOR END; (* record `)

VAR START:PVECTOR; (* Head of list of lines 4) HEAP: "INTEGER;

PROCEDURE MAKESUBPICTURE;

PROCEDURE DRAWSIJBPICTURE;

IMPLEMENTATION

PROCEDURE MA~SUBPICTURE; BEGIN (4 Calculates next subpicture and stores on heap *) END;

PROCEDURE DRAWSUBPICTURE; BEGIN LPVEC:~START; (4 Start at beginning of' list 4) WHILE LPVEC<>NIL EC (` and draw each that's there 4) WITH LPVEC" DO BEGIN TURN(DELHEADING); MOVE (DELDISTANCE); IF PENDGI4N THEN TGPEN:~WHITE ELSE TGPEN:=NONE; LPVEC:~NEXTVEC; END; END; (* drawsubpicture 4)

Page 171



END;

USES TURiEGRAPHICS,PICTUREGRAPHI~; (* picturegraphics uses `)

BEGIN (` turtlegraphics INITrURTLE; REPEAT MARK~HEAP); MA!~SUBPICTURE; DRAWSUBPICTURE; RELEASE (HEAP); UNTIL START=NIL; E ND.

EXAMPLE C

< Compilation unit > ::: < Progr~i heading > ; { < Unit definiticn > < Uses part > (Block > I < Unit definition > { ; < Unit definition > ~.

< Unit definition > :~: < unit heading >; < Interface part > < Implementation part > End < Unit heading > ::= Unit < Unit identifier > Se~rate unit < Unit identifier)

< Unit identifier > < Identifier >

< Interface part > Interface < Uses part > (Constant definition part > < Type definition part > < Variable declaration part) < Procedure heading > < Funct~.cn neadir.g >

< Imple~entation part> ::= Implementation < Label declaration part > < Constant definition part > (Type definition ~rt) (Variable declaration part > < Procedure and Function declaration oart >

< Uses part > :: Uses < Unit identifier > { , < Unit identifier>) ; < Empty >

See Section 5.9 for Syntax diagrans. r

DIAGRAM D Page 172



The user may define a UNIT in−line, after the heading of the host program. In this case the user compiles both the UNIT, and the host program together. Any subsequent changes in the UNIT or host program require the user to recompile both. The user may also define and compile a UNIT (or a group of UNITs) separately, and use the library manager to store it C 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 syst~n to r~nind you to link the file

Changes in a host program require only that the user recompile the program and link in the UNIT. changes in the IMPLEMENTATION part of a UNIT only require the user to compile the UNIT, and then to relink all compilation units that use that UNIT. changes in the INTERFACE part of a UNIT require that the user recompile both the UNIT and all compilation units that use that UNIT. In this case all these compilation units must again be linked. For more information see section 1.8 LINKER or section 14.2 LIBRARIAN.

The compiler generates LINKER information in the contiguous blocks following the code for a program that uses UNITs. This information contains locations of references to externally defined identifiers. Section 1.8 explains the format of this information.

3.3. 3.2 PASCAL TO ASSEMBLY LANGUAGE LINKAGES − EXTERNAL PR~EDURES

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.

Page 173



A host progra~ declares that a procedure is external in much the sane 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 canpiler checks that calls to the external agree in type and rnzaber of paraneters with the external declaration. It is the user's responsibility to assure that the assanbly language procedure respects the Pascal external declaration. The linker checks only that the ni.xnber of words of paraneters agree between the Pascal and assanbly language declarations. For more information see section 1.8 Linker and 1.9 Assenbier(s).

The conventions of the surrounding systen concerning register use and calling sequences must be respected by writers of assembly language routines. These conventions for the PDP−11 and Z80/8080 implenentations are given here.

First, for the PDP−11, registers RO and Ri are available f~r use; any others affected by a routine must be saved on entry and restored on exit. The following call and return sequence is reca~mended for procedures. It has the advantage that calls can ~e made directly fran assembly language as well as Iran Pascal.

.PRcC ENTRY, 2

PARAM1 .EQU 6 ;Offset for first paraneter PARAM2 .EQU 4 ;Offset for second paranter RETADDR .EQU 2 ;Offset for return address OLDR5 .EQU 0 ;Offset for original value of R5 LOCALI .EQU −2 ;Offset for first local LOCAL2 .E~J −4 ;OfTset for second local

MDV R5,−(SP) ;Save contents of R5 MDV SP,R5 ;Use R5 to get at locals and parameters CLR −(SF) ;Reserve and Initialize CLR −(SF) ;Two local variables

;Inside routine MDV PARAM(R5),LOCAL1(R5) ;Sainple statement

EXIT: MOV R5,SP ;Cut back to entry SF MOV (SP)+,R5 ;Restore previous R5 MOV (SP)+,RO ;Get return address ADD #NFARAMS,SP ;Discard paraneters, nt.znber of bytes JIP R0 ;Return to caller

Page 174



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,LOCAL 1 ,LOCAL2, PARAM 1, PARAM2 ;Reserve st~'tic 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 PARP~M2 LD (PARAN2),HL FOP HL ;Get and save PARAM 1 LD (PARP~M1),IL

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 sane, 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 mechanisizns 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.

Page 175



.PROC DRCOLLECT,0 ; Nane of routine for use by linker. * CONST ~B1FLENG ; Public constant. *PtJBLIC DR8UFFER ; Public variable.

DRADDR *EQU 167770 DRVECT .EQU 1~40 MOV IFLANDLR,#DRVECT ;L.oad address of interrupt MOV #3Z~O,#DRVECT+2 ;handler and set priority. MOV #DRBUFLENG,R0 ;Load RO with size of' buffer. MOV ~RBUFFER,R1 ;Load Ri with address of buffer. BIS #100,#DRADDR ;Enable interrupts on DR interface.

LOOP: .TST RO ;Exit loop when buffer full. BNE LOOP BIC #100,#DRADDR ;Disable interrupts. RTS PC ;Return to PASCAL host prcgrarn. HANDLR: ~1OV ~#DRAD~+2,(R1)+ ;Load buffer with next ~rd, DEC RO ;increment Ri, decrement RC. RTI ;Return fran interrupt.

PRGGRP~M COLLECTDATA; CONST DRBUFLENG = 256;

TYPE DATABUFFER Array E1..DRBIJFLENG) of integer;

VAR I: Integer; DRBUFFER: DATABUFFER; DATAFILE: File of DATABLIFFER;

PR~CEDURE DR COLLECT; External;

BEGIN (*~f Collect Data*) Rewrite(DATAFILE, `SAMPLE.~TA'); For I::1 to 10 do BEGIN DRCOLLECT; DATAFILE~ : =DRBIFFER; Put(DATAFILE); END; Close(DATAFILE, Lock); END.

Page 176



3.3.2.3 ASSE>IBLY LANGUAGE TO ASS~1BLY LANGUA~ LINKAGES

The third way in which separate routines may share data structures and subroutines is by linkage frctn assembly language to assembly language. This is made possible through the use of the .LEF 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 .cEF for a label to be referenced by the second routine C Note that this is only a skeleton of the actual MOVETO routine):

.PRcC MOVETO,6 ; THE 3 REAL PARAMETERS OCCUPY 6 WORDS

PROCEDURE MOVETO(X, Y, Z: REAL);

~1PUTES A ND~ PSXPOS & PSYPOS FRO4 PSMATP AND AN ASSUMED 1 0 AS THE INPW~ VECTOR HCt~1OGENOUS COORDINATE...

CX Y Z 1) dot PSMATP~ = CX? Y' Z' W') PSXPOS := PSYPOS YVW';

THESE ARE GLOBALS IN THE PASCAL HOST .PUBLIC PSXPOS PUBLIC PSYPOS .PUBLIC PSMATP

MOVETO ENTRY POINT

MG/ R5,−(SP) ; R5 USED AS FRAME POINTER MOV SP,R5 MOV #PSMATP,RO ; RO IS TOS MATRIX POINTER

PARAMETER DISPLAC~1ENTS FR~4 R5 FRAME POINTER X .EQU 1~4 Y .EQU 10 2 .EQU `~ W .EQU −~4

CQ~PUTE W', H~t~OZNEOUS COORD AND LEAVE IT ON STACK

Page 177



~IPUTE PS~OS

NO.4 CQIPUTE PSYPOS

; CLEAN UP STACK AND RETIJRN

ROUND: ; ROUND REAL ON STAQC TO INTEGER ~ < 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 C~AWUNE routine. MOVETO must be linked irit~ LINETO before the routine can be linked in as an external procedure t~ a PASCAL UNIT or PROGRAM.

.PRCC LINETO,6 ; PARAMETERS OCCUPY 6 WCR~

PR~~URE LINETO(X, Y, Z: REAL);

DRAWS A LINE FRO~ TFE LAST POINT CONTAINED IN PSXPOS & PSYPOS TO THE NB4 TRANSFORMED POINT GIVEN BY X, Y, & Z...

SAVEX :: PSXPOS; SAVEY :: PSYPOS; MOVEIO(X, Y, Z); DRAWLINE(JUNK, PSBUFP~, 20, 160+SAVEX, 120−SAVEY, PSXPOS−SAVEX, SAVEY−PSYPOS, 1);

.PUBLIC PSX?OS PUBLIC PSYPOS .FUBLIC PSBUFP .PRIVATE RANGE

* REF MO VETO .REF DRAWLINE

LINETO ENTRY POINT

Page 178



MOV R5,−(SP) MOV SP,R5 ; USE R5 AS STACK FRAME POINTER SAVEX .E~J −2 SAVEY .EQU X .EQU 1L~ Y .EQU 10 Z .EQU

SAVEX :~ PSXPC~; SAVEY :: PSYPOS;

; MOVEID(X, Y, 2);

JSR PC,1~~10VETO

DRAWLINE(...);

JSR PC,&/DRAWLINE

ALL DONE... RETURN

NP ~R0

END

For exanpies and more information see section 1.9 ASS~i

Page 179


180

3.3.3. Long Integers

Version I.5, September 1978



With the predeclared type INTEGER the optional use of a length attribute constitutes a new type and will, in the remainder of this document, be referred to as LONG INTEGER. The LONG INTEGER is suitable for business, scientific or other applications which need extended number lengths with complete accuracy. This extension supports the four basic standard iNTEGER arithmetic operations (addition, subtraction, division and multiplication) as well as routines facilitating conversion to strings and standard INTEGERs. Strong type checking is enforced throughout in the Pascal spirit. Input/Output, iii 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 INTEG~ 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 ~uld be created by:

VAR X: INTEGERE8J;

Constants are defined in the normal manner:

~NST RYDBERG = 10973731;

In the above example RYDBERG ~uld 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 ~rds 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.

Page 181



Examples:

VAR I: flflEG~; L: INT~ERN; {~Eiere N is an integer constant <=36} S: REAL;

I:: L; (syntax error, see TRUNC(L) below) L:=–L; (correct, with the usual excepticn} L:= I; {always correct) L:: S; {riever accepted) 5:: L; (will be imple~ented with 11.3)

Arithnetic operations which may be used in conjunction with LONG INTEGERs are any or all fran the set [+,–,~`,DIV,unary plus/minus). O~i assigrrnent 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 rn.znber 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 21147Z~836~47, length attributes of 10 thru 114 may store thru 1A40737L~88355327, 15 tnru 18 922337203685~4775807. 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 L0~G INTEGER as well as a REAL as an argu'nent). 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 ~t in the STRING `3'. The following program segment will provide a suitable dollar and cent routine:

STR(L.,S); INSERT('.',S,LENGTh(S)−l); WRITELN(S);

where `L' and `S' are appropriately declarec. 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 circunvented by creating a type which is a LCNG INTEGER. For example:

Page 182



TYPE LO~3 INTEGER 18; PR~EDLTRE BIGNU1BER~NKA~T: LONG);

The LONG INTEGER is stored as a multi−word, twos complement binary nunber. System and interpreter routines do the I/O conversions as required. Maximun storage efficiency is achieved by dynamic expansion and contraction of word allocation as required. flaring 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.

Page 183



3.3. Procedure Notes

Version I.5, September 1978

Declarations of SEGMENT procedures and functions are identical to standard Pascal procedures and functions except they are preceded by the reserved word “SEGMENT”, for example:




```
SEGMENT PROCEDURE INITIALIZE;
BEGIN
  (* PASCAL code *)
END;
```

Program behavior differs, however, as code and data for a SEGMENT procedure (function) are in memory only while there is an active invocation of that procedure.

Advantages and benefits:

The user may now put large pieces of one−time code, eg. initialization code, into a SEGMENT procedure. After performing the initialization, for example, the now−useless code is taken out of memory thus increasing the available memory space.

Furthermore the user may now compile his/her program in chunks, specifically in SEGMENTS. The LIBRARIAN program (described in Section 4.8) can be used to link together the separate segments to produce one large code file.

Requirements and limitations:

The disk which holds the codefile for the program must be on− line (and in the same drive as when the program was started) whenever one of SEGMENT procedures is to be called. Otherwise the system will attempt to retrieve and execute whatever information now occupies that particular location on the disk, usually with very displeasing and certainly unexpected results.

A maximum of seven (7) SEGMENT procedures are ordinarily available to the user, including the main body segment.



SEGMENT procedures must be the first procedure declarations containing code−generating statements.

For further details and examples see Section 3.5, Introduction to the Pascal Pseudo Machine.

3.4. Pseudo Machine Architecture

Version I.5, September 1978



The UCSD Pascal P−machine, designed specifically for the execution of Pascal programs on small machines, is an extensively modified descendant of the P−2 pseudo−machine from Zurich. It supports variable addressing, including strings, byte arrays, packed fields, and dynamic variables; logical, integer, real, and set top−of− stack arithmetic and comparisons; multi−element structure comparisons; several types of branches; procedure/function calls and returns, including overlayable procedures; miscellaneous procedures used by systems programs; and basic I/O subsystem.

This Section, to be used in conjunction with Section 3.5, describes the P−machine "hardware," communication with the operating system, exceptional condition handling, the instruction set, the I/O system, and the boot−loading process.

3.4.1. Hardware (Emulation)

The P−machine uses 16−bit words, with two 8−bit bytes per word. It has several registers and a user memory, in which are kept a stack and a heap. All registers are pointers to word−aligned structures, except IPC, which is a pointer to byte−aligned instructions. The registers are:

SP	Stack Pointer is a pointer to the top of the execution stack. The stack starts in high memory and grows toward low memory. It contains code segments and activation records, and is used to pass parameters, return function values, and as an operand source for many instructions. The stack is extended by loads and procedure calls, and is cut back by stores, procedure returns, and arithmetic operations.
NP	New Pointer is a pointer to the top of the dynamic heap. The heap starts in low memory and grows upward toward the stack. It contains all dynamic variables (see Jensen and Wirth, Chapter 10). It is extended by the standard procedure 'new', and is cut back by the standard procedure 'release'.
JTAB	Jump TABle pointer is a pointer to the procedure attribute table of the currently executing procedure. (See Section 3.5, figure 5.)
SEG	Segment Pointer points to the procedure dictionary of the segment to which the currently executing procedure belongs. (See Section 3.5, figure 6.)
MP	Most recent Procedure is a pointer to the activation record of the currently executing procedure (see Section 3.5, figure 7). Variables local to the current procedure are accessed by indexing off MP.
BASE	BASE Procedure is a pointer to the activation record of the most recently invoked base procedure (lex level 0). Global (lex level 0) variables are accessed by indexing off BASE.



3.4.2. Operating System/p−Machine Communication − SYSCOM

It is sometimes necessary for the operating system and the p−machine to exchange information. Hence there exists a variable SYSCOM in the outer block of the operating system, and a corresponding area in memory known to the hardware. The fields in SYSCOM actually relevant to this conTnunication 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	copy of the BASE register.
LASMP	copy of the MP register.
SEG	copy of the SEG register.
JTAB	copy of the JTAB register.
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.
HLYLINE	contains the line number of the last conditional halt executed. (See BPT instruction.)
BRKPTS	contains up to four line numbers of breakpointed statements. (See BPT instruction.)
CRTINFO.EOF	contains the end−of−file character (see console input driver).
CRTINFO.FLUSH	contains the flush−output character (see console input, output drivers).
CRTINFO.STOP	contains the stop−output character (see console output and input drivers).
CRTINFO.BREAK	contains the break−execution character (see console input driver).
SEGTABLE	contains the segment dictionary for the Pascal system.



3.4.3. Exception Handling – XEQERR

Whenever a run−time error occurs, the P−machine stops executing the current instruction (ideally leaving the evaluation stack in as nice a condition as possible) and transfers control to the XEQERR routine. This routine

1. enters the error code into SYSCOM^.XEQERR.
2. calculates what MP will be after step 4, and sets SYSCOM^.BOMBP to that. (The size of EXECERROR's activation record must be known by the P−machine.)
3. stores the current value of IPC into SYSCOM^.BOMIPC.
4. points IPC to a CXP 0,2 (call operating system procedure EXECERROR) instruction.
5. resumes execution of interpreter code, starting with the CXP.

3.4.4. Operand Formats

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

Boolean:

One word. Bit 0 indicates the value (false:O, 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 1.4) support.

PDP11:

	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Word 1:	low mantissa															
Word 0:	s	exponent							high mantissa							

Z80/8080:

	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Word 1:	low mantissa								middle mantissa							
Word 0:	s	high mantissa							exponent							

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

Pointers:

One or three words, depending on type of pointer.

Pascal pointers, internal word pointers:
one word, containing a word address.

Internal byte pointers:
one word, containing a byte address.

Internal packed field pointers:
three words.

word 2: word pointer to word field is in.

word 1: field−width (in bits).

word 0: right−bit−number of field.

Set:

0..255 words in data segment, 1..256 words on stack. Sets are implemented as bit vectors, always with a lower index of zero. A set variable declared as set of *m..n* is allocated *(n+15) div 16* words. When a set is in the data segment, all words allocated contain valid information.

When a set is on the stack, it is represented by a word containing the length, and then that number of words, all of which contain valid information. All elements past the last word of a set are assumed not to be elements of the set. Before being stored back in the data segment, a set



must be forced back to the size allocated to it, and so an ADJ instruction must be issued.

Records and Arrays:

any number of words (up to 16384 words in one dimension). Arrays are stored in row−major order, and always have a lower index of zero. Only fields or elements are loaded onto the stack − never the structure itself. Packed arrays must have an integral number of elements in each word, as there is no packing across word boundaries (it is acceptable to have unused bits in each word). The first element in each word has bit 0 as its low−order bit.

Strings:
: 1..128 words. Strings are a flexible version of packed arrays of char. A string[n] occupies *(n div 2)+1* words. Byte 0 of a string is the current length of the string, and bytes *1..length(string)* contain valid characters.

Constants:
: constant scalars, sets, and strings may be imbedded in the instruction stream, in which case they have special formats. All scalars (excluding reals) not in the range 0..127: two bytes, low byte first.

Strings:
: all string literals take *length(literal)+1* bytes, and are byte aligned. The first byte is the length, the rest are the actual characters. This format applies even if the literal should be interpreted as a packed array of char (see S1P and S2P below).

Reals and sets:
: word aligned, and in reverse word order.

3.4.5. Instruction Set Format

Instructions on the P−machine are one or two bytes long, followed by zero to four parameters. Most parameters specify one word of information, and are one of five basic types.

UB unsigned byte:
: high order byte of parameter is implicitly zero.

SB signed byte:
: high order byte is sign extension of bit 7.

DB don't care byte:
: can be treated as SB or UB, as value is always in the range 0. .127.

B big:
: this parameter is one byte long when used to represent values in the range 0..127, and is two bytes long when representing values in the range 128..32767. If the first byte is in 0..127, the high byte of the parameter is implicitly zero. Otherwise, bit 7 of the first byte is cleared and it is used as the high order byte of the parameter. The second byte is used as the low order byte.

W word:
: the next two bytes, low byte first, is the parameter value.

Any exceptions to these formats are noted in the instructions where they occur.



3.4.6. English Instruction Set Description

In the following section, references to an element on the stack are context−dependent, and can mean anywhere from one word to 256 words. Also, unless specifically noted to the contrary, operands are popped off the stack – they are not left around.

Abbreviations are used widely, but use fairly simple conventions. Parameters are written as X or X*n*, where X is UB, SB, DB, B or W, and *n* is an integer indicating the parameter position in the instruction. Tos means the

operand on the top of stack, tos−1 the next operand, etc. Mark Stack Control Word is abbreviated to MSCW.

Many instructions refer to the activation record of a procedure, and this document assumes the reader has a general knowledge of procedure calling in stack machines, and the concept of stack frames. An activation record as defined in this document specifically consists of:

1. the local data segment of the procedure, and
2. the MSCW, containing addressing information (static links), and information on the calling procedures environment when the procedure was called. (See Section 3.5, figure 7.)

The dynamic chain refers to the calling chain, traversed using the `MSCW.MSDYN` links. The static chain refers to the lexical or ancestor chain, traversed using the `MSCW.MSSTAT` links.

5.A. Variable Fetching, Indexing, Storing, and Transfering

5.A.1. One Word Loads and Stores

5.A.1.a. Constant One Word Loads

Mnemonic	Opcode	Parameters	Full Name and Operation
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

Mnemonic	Opcode	Parameters	Full Name and Operation
SLDL1 ... SLDL16	216 ... 231		Short load local word. SLDL*x* fetches the word with offset *x* in MP activation record and pushes it.
LDL	202	Big	Load local word. Fetches the word with offset B in MP activation record and pushes it.
LLA	198	Big	Load local address. Fetches address of the word with offset B in MP activation record and pushes it.
STL	204	Big	Store local word. Stores tos into word with offset B in MP activation record.



5.A.1.c. Global One Word Loads and Store

Mnemonic	Opcode	Parameters	Full Name and Operation
SLDO1 ... SLDO16	232 ... 247		Short load global word. SLDO*x* fetches the word with offset *x* in BASE activation record and pushes it.
LDO	167	Big	Load global word. Fetches the word with offset B in BASE activation record and pushes it.
LAO	165	Big	Load global address. Pushes the word address of the word with offset B in BASE activation record.
SRO	171	Big	

			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

Mnemonic	Opcode	Parameters	Full Name and Operation
LOD	182	DB, Big	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, Big	Load intermediate address.
STR	184	DB, Big	Store intermediate word.

5.A.1.e. Indirect One–Word Loads and Store

Mnemonic	Opcode	Parameters	Full Name and Operation
STO	154		Store indirect. Tos is stored into the word pointed to by tos−1.
SIND0	248		Load indirect.



5.A.2. Multiple Word Loads and Stores (Sets and Reals)

Mnemonic	Opcode	Parameters	Full Name and Operation
LDC	179	UB, *block*	Load multiple word constant. UB is the number of words to load, and <block> is a word aligned block of UB words in reverse word order. Load the block onto the stack.
LDM	188	UB	Load multiple words. Tos is a pointer to the beginning of a block of UB words. Push the block onto the stack.
STM	189	UB	Store multiple words. Tos is a block of UB words, tos−1 is a word pointer to a similiar block. Transfer the block from the stack to the destination block.

5.A.3. Byte Arrays

Mnemonic	Opcode	Parameters	Full Name and Operation
BYT	210		Byte conversion. Convert word pointer tos to a byte pointer. (NOP in the PDP11 and Z80/8080 implementations.)
LDB	190		Load byte. Push the byte (after zeroing high byte) pointed to by byte pointer tos.
STB	191		Store byte. Store byte tos into the location specified by byte pointer tos−1.
MVB	169	Big	Move bytes. Tos is a byte source pointer to a block of B bytes, tos−1 is byte destination pointer to a similar block. Transfer the source block to the destination block (This instruction is redundant due to word alignment, and will be replaced by MOV in the future.)
IXB	209		

			Index byte array. Push a byte pointer formed from the integer index tos and the byte pointer tos−1.

5.A.4. Strings

Mnemonic	Opcode	Parameters	Full Name and Operation
LCA	166	UB, *chars*	Load constant string address. Push a byte pointer to the location UB is contained in, and skip IPC past <chars>.
SAS	170	UB	String assign. Tos is either a source byte pointer or a character. (Characters always have a high byte of zero, while 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.
S1P	208		String to packed conversion on tos. Tos is a byte pointer to a string, and is incremented by one byte in order to point to the first character of the string.
S2P	157		String to packed conversion on tos−1. Tos and tos−1 are byte pointers, and tos−1 is incremented by one byte.
IXS	155		Index string array. Performs the same operation as IXB, except before indexing the index is checked to see if it is in the range 1..current length. If not, a run−time error occurs.



5.A.5. Record and Array Indexing and Assignment

Mnemonic	Opcode	Parameters	Full Name and Operation
MOV	168	Big	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	Big	Static index and load word. Indexes the word pointer tos by B words, and pushes the word pointed to.
INC	162	Big	Increment field pointer. The word pointer tos is indexed by B words and the resultant pointer is pushed.
IXA	164	Big	Index array. Tos is an integer index, tos−1 is the array base word pointer, and B is the size (in words) of an array element. A word pointer

			to the indexed element is pushed.
IXP	192	UB1, UB2	Index packed array. Tos is an integer index, tos−1 is the array base word pointer. UB_1 is the number of elements per word, and UB_2 is the field width (in bits). Compute and push a packed field pointer.
LDP	186		Load a packed field. Push the element 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

Mnemonic	Opcode	Parameters	Full Name and Operation
NEW	158 1		New variable allocation. Tos is the size (in words) to allocate the variable, and tos−2 is a word pointer to a dynamic variable. If GDIRP is non−nil, cut NP back to GDIRP and set GDIRP to nil. Store NP into word pointed to by tos−1, and increment NP by tos words.
MRK	158 31		Mark heap. Release GDIRP and set to nil if necessary, then store NP into word pointed to by tos.
RLS	158 32		Release heap. Set GDIRP to nil, then store word pointed to by tos into NP.

5.B. Top of Stack Arithmetic and Comparisons

5.B.1. Logical

Mnemonic	Opcode	Parameters	Full Name and Operation
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 compliment of tos
EQUBOOL	175 6		Boolean = comparison. Compare bit_0 of tos−1 to bit_0 of tos and push true or false.
NEQBOOL	183 6		Boolean <> comparison. Compare bit_0 of tos−1 to bit_0 of tos and push true or false.
LEQBOOL	180 6		Boolean <= comparison. Compare bit_0 of tos−1 to bit_0 of tos and push true or false.
LESBOOL	181 6		Boolean < comparison. Compare bit_0 of tos−1 to bit_0 of tos and push true or false.
GEQBOOL	176 6		Boolean >= comparison. Compare bit_0 of tos−1 to bit_0 of tos and push true or false.
GTRBOOL	177 6		Boolean > comparison. Compare bit_0 of tos−1 to bit_0 of tos and push true or false.



5.B.2. Integer

Mnemonic	Opcode	Parameters	Full Name and Operation
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 compliment 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 = comparison. Compare tos−1 to tos and push true or false.
NEQI	203		Integer <> comparison. Compare tos−1 to tos and push true or false.
LEQI	200		Integer <= comparison. Compare tos−1 to tos and push true or false.
LESI	201		Integer < comparison. Compare tos−1 to tos and push true or false.
GEQI	196		Integer >= comparison. Compare tos−1 to tos and push true or false.
GTRI	197		Integer > comparisons. Compare tos−1 to tos and push true or false.



5.B.3. Reals

All over/underflows cause a run−time error.

Mnemonic	Opcode	Parameters	Full Name and Operation
FLT	138		Float top−of−stack. The integer tos is converted to a floating point number.

FLU	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 aid Wirth*), then truncated and converted to an integer.
ABR	129	Abs 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. Take the cosine of the real tos.
ATAN	158 27	Arctangent. Take the arctangent of the real tos.
EXP	158 29	Exponential. Calculate e^{tos} and push result.
LN	158 28	Natural logarithm.
LOG	158 26	Log base 10.
SQT	158 30	Square root.
EQUREAL	175 2	Real = comparison. Push true or false.
NEQREAL	183 2	Real <> comparison. Push true or false.
LEQREAL	180 2	Real <= comparison. Push true or false.
LESREAL	181 2	Real < comparison. Push true or false.
GEQREAL	176 2	Real >= comparison. Push true or false.
GTRREAL	177 2	Real > comparison. Push TRUE or FALSE.

5.B.4. Sets

Mnemonic	Opcode	Parameters	Full Name and Operation
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 off high words of set), and its length word is discarded.

SGS	151	Build a singleton set. The integer tos is checked to ensure 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 = comparison. Push true or false.
NEQPOWR	183 8	Set <> comparison. Push true or false.
LEQPOWR	180 8	Set <= (subset of) comparison. Push true or false.
GEQPOWR	176 8	Set >= (superset of) comparison. Push true or false.

5.B.5. Strings

Mnemonic	Opcode	Parameters	Full Name and Operation
EQUSTR	175 4		String = comparison. Push true or false.
NEQSTR	183 4		String <> comparison. Push true or false.
LEQSTR	180 4		String <= comparison. Push true or false.
LESSTR	181 4		String < comparison. Push true or false.
GEQSTR	176 4		String >= comparison. Push true or false.
GTRSTR	177 4		String > comparison. The string pointed to by word pointer tos−1 is lexicographically compared to the string pointed at by tos. Push true or false.



5.B.6. Byte Arrays

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

Mnemonic	Opcode	Parameters	Full Name and Operation
EQUBYT	175 10		Byte array = comparison. Push true or false.
NEQBYT	183 10		Byte array <> comparison. Push true or false.
LEQBYT	180 10		Byte array <= comparison. Push true or false.
LESBYT	181 10		Byte array < comparison. Push true or false.
GEQBYT	176 10		Byte array >= comparison. Push true or false.
GTRBYT	177 10		Byte array &gt comparison. Push true or false.

5.B.7. Array and Record Comparisons

Mnemonic	Opcode	Parameters	Full Name and Operation
EQUW0RD	175 12		Word or multiword structure = comaprison. Push true or false.
NEQW0RD	183 12		Word or multiword structure comaprison. Push true or false.

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].

Mnemonic	Opcode	Parameters	Full Name and Operation
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 W3. Otherwise, tos−W1 is used as an index into the table, and IPC is set to byte address(casetable[index−min_index])−casetable[index−min_index].

5.D. Procedure and Function Calls and Returns

The general scheme used in procedure/function invocation is

1. Calculate the data size and parameter size of the called procedure by using the information in the current procedure dictionary (pointed to by SEG).
2. Extend stack by data size bytes.
3. Copy parameter size bytes from the old top−of−stack to the beginning of the space just allocated.
4. Build a MSCW, saving SP, IPC, SEG, JTAB, MP, and a pointer to the most recent activation record of the called procedure's immediate parent.
5. Calculate new values for SP, IPC, JTAB, MP, and if necessary, SEG. Check for stack overflow.
6. If the called procedure has a lex level of −1 or 0 save BASE and calculate a new BASE.

Mnemonic	Opcode	Parameters	Full Name and Operation

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 NP.
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 procecure 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.
2. Is desired segment in memory?
   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.
   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	UB	Call Standard Procedure. Scan this document for an opcode 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 one caller's top−of−stack. The information in the MSCW is then used to restore the caller's correct environment.
RBP	193		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. System Programs Support Procedures

See Section 2.1 for description of these procedures.

Byte Array Procedures

Mnemonic	Opcode	Parameters	Full Name and Operation
FLC	155 10		Fillchar(dst, len, char).
SCN	158 11		Scan(maxdisp, start, forpast, char, masx).
MVL	158 02		Moveleft(src, dst, niinbytes).
MVR	158 03		Moveright(src, dst, numbytes).

Compiler Procedures

Still undocumented.

Mnemonic	Opcode	Parameters	Full Name and Operation
TRS	158 08		Treesearch.
IDS	158 07		Idsearch.

Debugger

Mnemonic	Opcode	Parameters	Full Name and Operation
BPT	213	UB	Breakpoint (conditional HALT). The UB is the line number in the source file.

Miscellaneous

Mnemonic	Opcode	Parameters	Full Name and Operation
TIM	158 09		Time(var high, low: integer)
XIT	214		Exit. The p−Machine halts immediately. (Not to be confused with the EXIT opcode.)
NOP	215		No operation.



3.5. Introduction to the Pascal Pseudo–Machine

Version II.0, February 1979



UCSD uses an interpreter based implementation of Pascal. This means that the compiler emits code for a pseudo–machine which is emulated at run time by a program written in the machine language of the host. The compiler, program editor, stand–alone operating system, and various utilities are themselves written in Pascal and run on the same interpreter. Thus the entire system can be moved to a new host machine by rewriting the interpreter for the new host. This document describes the Pseudo–machine code files as they were in version I.3. Many of the segments mentioned are no longer resident in the code file used as an example. This does not affect the functionality of the description of the mechanisms put forth by this document.

Figure 3.5.10 (the last page of this document) is a skeleton version of a large Pascal program, here–in–after referred to as "The Program". This document is a top–down description of the realization of that program on the UCSD Pascal system. We will make occasional use of a helpful coincidence: The Program is the framework of the portion of the UCSD Pascal environment that's written in Pascal.

If The Program ware expanded to a complete Pascal system, it would consist of at least 6000 lines of Pascal and compile to more than 50,000 bytes of code — too big to fit all at once into the memory of a small machine (by our current definition of small). We have therefore extended Pascal so that a programmer can explicitly partition a program into *segments*; only some of which need be resident in main memory at a time. The syntax of this extension is shown in figure 3.5.1. (Any syntactic objects not defined explicitly there retain their standard interpretation as defined by Jensen & Wirth: *Pascal User Manual and Report*.) See Section 5.9 for revised syntax diagrams.

```
<program> ::= <program heading> <segment block> .

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

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

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

Figure 3.5.1. Segment Declaration Syntax.



Segment declaration syntax (figure 3.5.1) requires that all nested segments be declared before the ordinary procedures or functions of the segment body. Thus, a code segment can be completely generated before processing of code for the next segment starts. This is not a functional limitation, since forward declarations can be used to allow nested segments (COMPILER in The Program) to reference procedures in an outer segment body (CLEARSCREEN). Similarly, segment procedures and functions can themselves be declared forward.

Segmenting a program does not change its meaning in any fundamental sense. When a segment is called (e.g. the COMPILER segment in line A), the interpreter checks to see if it is present in memory due to a previous invocation. If it is, control is transferred and execution proceeds: if not, the appropriate code segment must be

loaded from disk before the transfer of control takes place. When no more active invocations of the segment exist, its code is removed from memory. For instance, in The Program, the code for the COMPINIT segment is not present in memory either before or after the execution of line A. Clearly, a program should be segmented in such a way that (non−recursive) segment calls are infrequent; otherwise, much time could be lost in unproductive thrashing (particularly on a system with low performance disk).

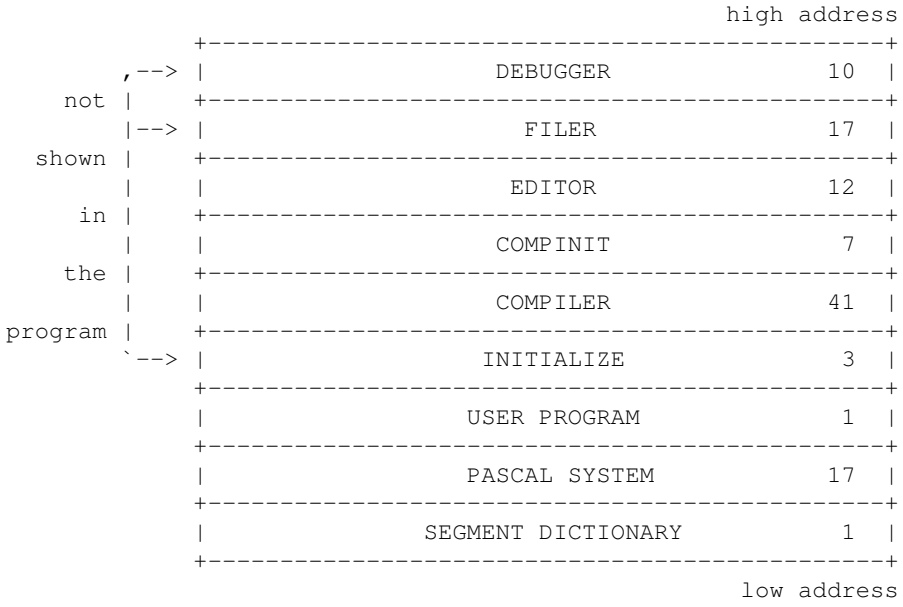


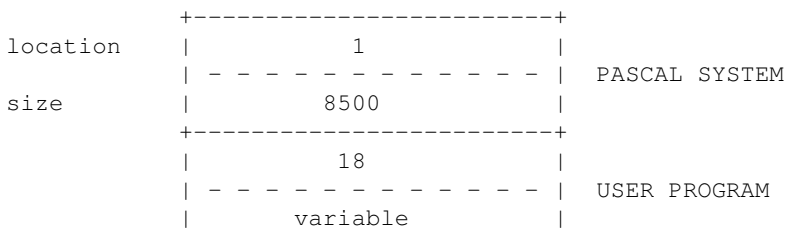
Figure 3.5.2. Pascal System Codefile.

The code file resulting from compilation of The Program is diagrammed in figure 3.5.2*. The file is a sequence of code segments preceded by a segment dictionary. The size of each segment is noted in blocks, the 512−byte disk allocation quantum used on most PDP−11 operating systems. The sizes indicated are representative of a full Pascal system. Each code segment begins on a block boundary. The ordering (from low address to high address) is determined by the order that one encounters segment procedure bodies in passing through The Program.

205

* An overview of the relationship between figures 3.5.2 through 3.5.8 (to be discussed in the following pages) is given in figure 3.5.9 at the end of this section. It is helpful to study figure 3.5.9 at this point for a better understanding of the section.

The segment dictionary in the first block of a code file contains an entry for each code segment in the file. The entry includes the disk location and size (in bytes) for the segment. The disk location is given as relative to the beginning of the segment dictionary (which is also the beginning of the code file) and is given in number of blocks. This information is kept in the system communications area (also called SYSCOM) during the execution of the code file, and is used in the loading of non−present segments when they are needed. Figure 3.5.3 details the layout of the table and shows representative contents for the Pascal system code file.



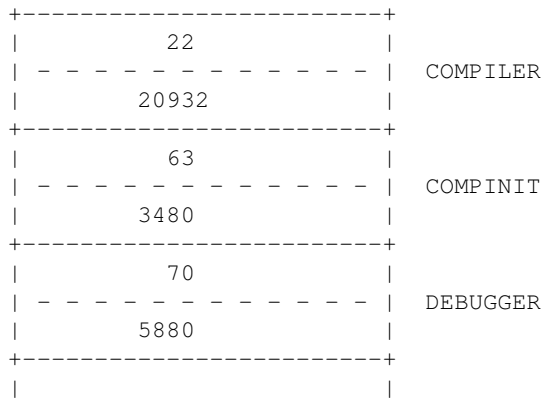


Figure 3.5.3. The Segment Dictionary

A code segment contains the code for the body of each of its procedures, including the segment procedure itself. Figure 3.5.4 is a detailed diagram of the code segment of The Program (Pascalsystem). Each of a code segment's procedure are assigned a procedure number, starting at 1 for the segment procedure, and ranging as high as 255 (current temporary limit of 127). All references to a procedure are made via its number. Translation from procedure number to location in the code segment is accomplished with the procedure dictionary at the end of the segment. This dictionary is an array indexed by the procedure number. Each array element is a self−relative pointer to the code for the corresponding procedure. Since zero is not a valid procedure number, the zero'th entry of the dictionary is used to store the segment number (even byte) and number of procedures (odd byte). Observe that CLEARSCREEN is the first procedure for which code is generated and that it appears at the beginning of the segment. The outer block code is generated and appears last.

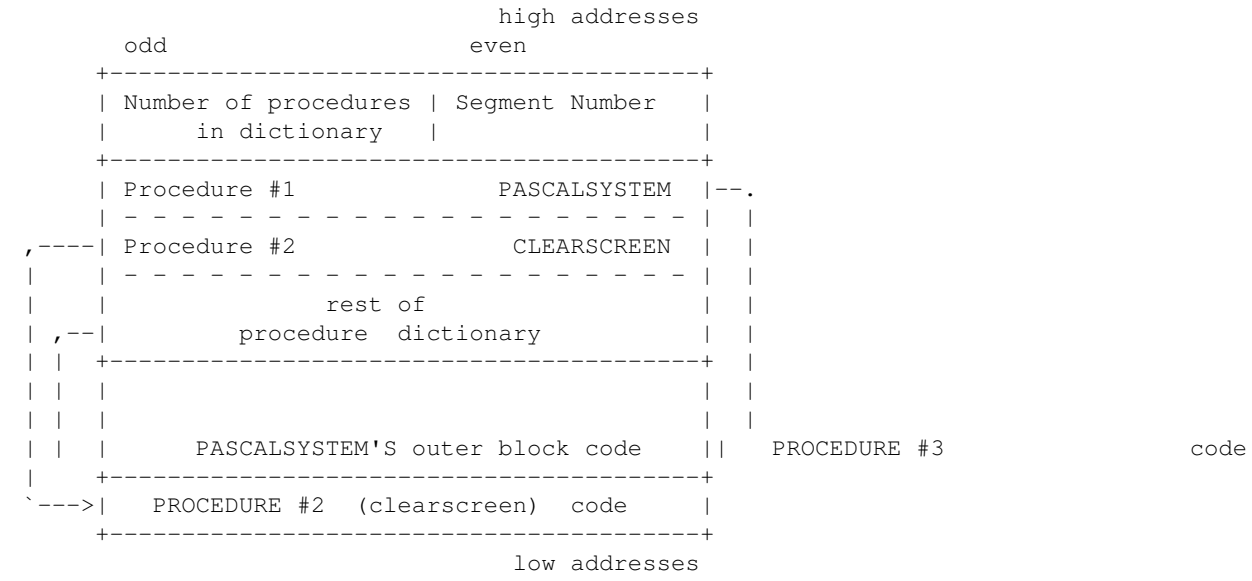


Figure 3.5.4. A Code Segment

A more detailed diagram of a single procedure code section is seen in figure 3.5.5. It consists of two parts: the procedure code itself in the lower portion of the section) and a table of attributes of the procedure. These attributes are:

LEX LEVEL

This odd byte is the depth of absolute lexical nesting for the procedure, (i.e. Lex Level (LL) Pascalsystem=−1, LL COMPILER or CLEARSCREEN=O, LL COMPINIT=l, etc.).

PROCEDURE NUMBER

This even byte refers to the number given in the procedure dictionary of the parent segment procedure. For example, the Procnun of CLEARSCREEN is 2. (see figure 3.5.4).

ENTER IC

This is a self−relative pointer to the first instruction to be executed far 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 parameter size is the number of bytes of parameters passed to a procedure from its caller.

DATA SEGMENT SIZE

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

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

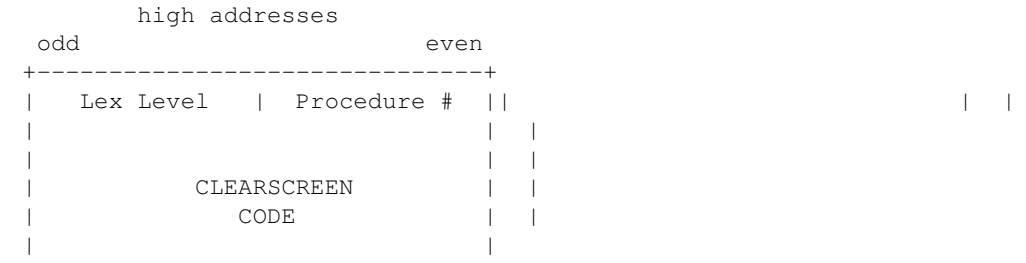
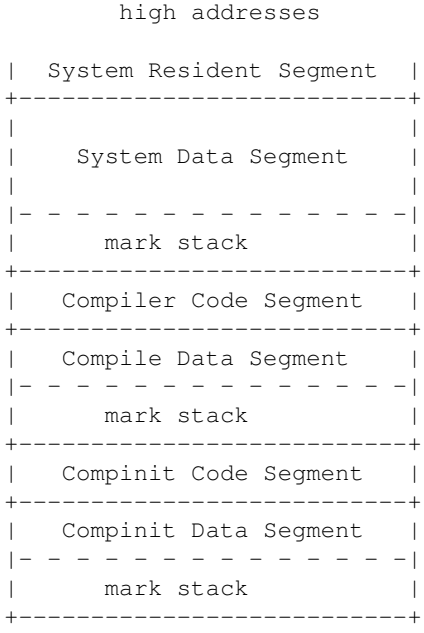


Figure 3.5.5. Procedure Code Section (of Clearscreen)



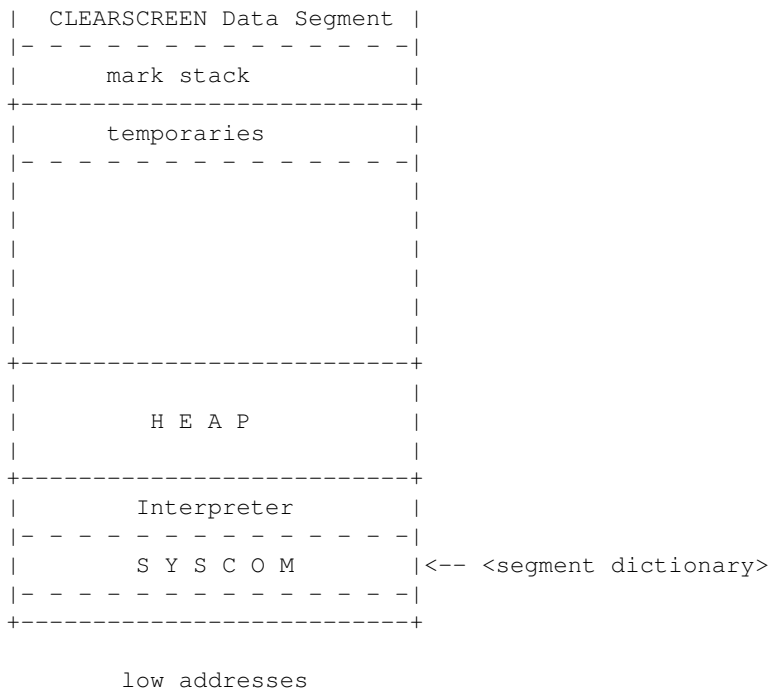

CLEARSCREEN Data Segment
mark stack
temporaries
H E A P
Interpreter
S Y S C O M
<-- <segment dictionary>
low addresses

Figure 3.5.6. System Memory During Clearscreen Execution

Figure 3.5.6 is a snapshot of' system memory during the execution of a call to procedure CLEARSCREEN frai, line C in CQ~1PINIT. The Pascal

~nterp−eter occupies the lowest area in memory. In it is the system conTnunications areaC 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 c~nmunication link between these t~ levels of the system. The Pascal heap is next in the memory layout; it grows toward high memory. The single stack growing down fran high memory is used for 3 types of items: 1) temporary storage needed during expression evaluation; 2) a data segment containing local variables and par~neters for each procedure activation; and 3) a code segment for each active segment procedure. (See figure 3.5.6)

Consider the status of operations just before C~1PINIT 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 SEQIEN~(SEG) variable:which points to the base of the procedure dictionary for the currently active segment procedure. For example, just before C~1PINIT is called, SEG points to the C~1PILER 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:~'~hich points to the collection of procedure attributes and ,jtznp table entries in the body of the current procedure code section,

heap.

and a N~.4 POINT~(NP)~.kiich points to the current top of the

When segment procedure C~4PINIT is called in line B, its code segment (jnclt.~ir~ all c~ipiler initialization procedures) is loaded on the stack. The C~4PINIT data segment is built on top of the stack. Figure 3.5.7 is a di~r~n of the data segment for C~1PINIT.

high addresses

; −−−−−a−−− − −−−−−−

Other CQIPINIT variables

low addresses

I, I

I I

I I

I I

I I

I I I I 1 I −~ I~

I I

I I

_l I

I, I – I ~ I

FIGURE 3.5.7. A DATA 5E~4ENT

In the upper portion of the data segment, space is allocated fc~r variables local to the new procedure. For ex~nple,C~1?INIT's data segment allocates space for integer variables I and ~J, as well as boolean BOOL.

I B~L

I

i

J

MSSP

MSIPC

MSSEG

MSDYN

MSSTAT MP −)~−−

rnarkstacx

Page 210

211

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 exanpie, the call to CC!IPINIT causes conditions in COMPILER just before the call to be stored in COMPINIT's markstack in the following manner:

MarkStack DYNanic link (MSDYN) <−− MP IPC(MSIPC) <−− IC SEGment Pointer(MSSEG) <−− SEG Jump TABle (MSJTAB) <−− JTAB " " Stack Pointer (SF) <−− SF

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, SB~, SF, 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 pints to the COMPINIT procedure dictionary, but to the Pascalsystem dictionary.

No code segment for CLEARSCREEN is added to the stack before the data segment since the code for CLEARSCREEN is already present in segment Pascalsystem. Its invocation causes only a data segment to be added to the stack. When CLEARSCREEN and INIT are completed, the COMPILER data segment will again be the top element on the stack.

Figure 3.5.8 is a detailed diagran of the stack during execution of an instruction in CLEARSCREEN, including appropriate pointers for static, dynanic, 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.

Page 211



to PASCALSYSTEN residont cede ligment

1o~ addresses

FIGURE 3.~.B ThE STACK DURING CLEM~CREEN

to PASCALSYSTEPI r~siden~ dmta s.~e.nt

a

LEO f'– in – high *ddresses ––– – JrAB ~c–pASCALS~STEPI IPC <–:od* sugment

~–

;

–

––––––~

CQ?IPILER code

–– –

segment

–I :

I

CQMPIi..ER data

s.gm.nt

.

I

–

I

8

,sarlstsck

I–

–

−−; 1

−

20

−

4

I Pointer to C3MFTNIT code

Pcjn:.r ~

ta−I

CCNPT~IT coda

oe~men;

CDMP INIT

data

Precedu's−. of CCIPINIT

CCMPINIT `,~si.ab1.s

PISSP

isxpc :−−−−−−−−−−−−−−−−−−−−~

~SSEG

(IS1JIAE

I− −−>:

I−~~ −)I −~ ~ :−−−~ ~<_I I 8 I _____ _____ _____ I I

i(−−~ 1:. II' III I I I I 4 4 4 I4I * I I I I I

* I I I I I I I

I I I I * p I * I I * I I I _____ S I I S

I S I I I I I I i'~SSTA I I I I, I I I I I I I S S I * I I *_I I I S I I * I I * I * S S I I

II ~ : : I I I S I I I * I I S I I * I I I * I I

: : −:

I I

I

MWVH

,1,.Iwat~n ~tac~

−−−a_−a−_a− I

CLEi~RSCRE~tJ var~ab1.s

r'~s:PC

C0I1~INIT

4 1

`i~m.nt *: : rSJTAB I CLEARS~~\

PiEDYN

?ISSTAT :C−−−I PIP `I −−1 evaluation stacg v I tp~ of SP I I NP H E A P too ~ I −

Pe.~e 212



Figure 3.5.9 illustrates a top−down process by showing the relationships anong diagrans 2 through 7.

code file figure 3.5.2

PAZCALSYSTEM figure 3.5.~4

CLEARSCREB~ −>~ figure 3.5.5: code detail proc. code −~ detail segment I dictionary figure 3.5.3 segment dictionary detail

system memory

figure 3.5.8 code segment −−−) figure 3.5.14 1 I I

I C~MPINIT

data segment −−−> figure 3.5.7 − data segment detail

FIGURE 3.5.9. RELATIONSHIP OF DOC~1ENT FIGURES

FIGURE 3.5.10. THE PROGRAM

PROGRAM PASCALSYST~1; VAR SYSCcM: SYSCcMREC; CH:CHAR;

Page 213



PROCEDURE CLEARSCREEN:FQRWA~;

SEQIENT PRCCEDURE USER PROGR~4M; BEGIN END; SEQNT PROCEDURE C~lPIIL~; VAR SY,OP:INTEG~; SYMCURSOR : INTEGER;

PROCEDURE fl~YMBOL; FORWARD;

S~1ENT PROCEDURE Cal?INIT; VAR I,J:INTEGER; BOOL: BOOLEAN; BEGIN

CLEARSCREEN; – ..INE C

INS'iMBOL;

END;

PROCEDURE INSYMBOL;

BEGIN ... END; PROCEDURE BLOCK; BEGIN ... END; BEGIN (*c~pIL~*) C~'4PINIT; LNE B INSYMBOL;

END; (*C~pILE9*)

SEGMENT PROCEDURE EDITOR; 3EGIN ... END;

PROCEDURE C LZARSCREEN BEGIN

WRITE(

END;

BEGIN (*PASC?J5YST~*) RE FEAT READ(CH); cASE CH ~ C:CQ'IPILER; – L1NEA

Page 21L4



E:EDITcR; U: USERPR~RAM

END(*CASE*) UNTIL CH `H' END.

Page 215



3.6. Byte Swapped

Version II.0, February 1979



Byte−swapping problems ocair when code generatad on one n~chine 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 mewry. Word quantities, such as integers, will be read and looked at in the sane way on both types of machines. However, byte quantities such as P−code or characters will be reversed within each word.

An example:

DEFINITION (A) (B) ls* 1P.s* flLS* 13*

VALUE(Hex) ! O~4 ! 07

B'X~TE 0 1

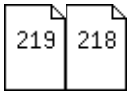
! 07 ! 0~4

0 1

C 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 ARRAYCO. .1] OF CHAR;) then Definition A would show a value of 0~4H and Definition B would show a value of 07H. Both definitions would show the value 0711 if the most significant byte were specified.

Byte−swappir~ 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.



**********,****** ********** * 0 0 P S * * Section 14.1 * ********** **********

Out Of Place Section

Page 219



3. Implementors' Guides

3.1.	Drawline	159
3.2.	File Formats	163
3.3.	Procedure Notes	165
3.3.2.	Linkage to Externally Compiled and Assembled Routines	167
3.3.3.	Long Integers	181
3.4.	Pseudo Machine Architecture	185
3.5.	Pseudo Machine Introduction	203
3.6.	Byte Swapped	217

4.2. Librarian

Version II.0, February 1979



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

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

Example

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

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

```
0– MOVETO   2398 4–     0  8–     0 12–     0
1– PSGRAPHI  864 5–     0  9–     0 13–     0
2–             0 6–     0 10–     0 14–     0
3–             0 7–     0 11–     0 15–     0
```

The following prompt line 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 tile. In this example, a separately compiled PASCAL UNIT called TURTLE is assigned 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 12–     0
```

1–	0	5–	0	9–	0	13–	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	12–	0
1– PSGRAPHI	864	5–	0	9–	0	13–	0
2– TURTLE	0	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 reponse 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`.



4.3. Setup — System Reconfiguration

Version II.0, March 1979



The UCSD Pascal Operating System keeps certain information about the user in a file called SYSTEM.MISCINFO. During each system initialization this file is read into memory, and from there it is accessed by many parts of the system, particularly (if the user has a terminal suitable for it) by the screen oriented editor.

Much of this information needs to be initially set up by the user to conform to his particular hardware configuration or his taste or convenience. Most of this information concerns the nature of the terminal and keyboard, although there are a few miscellaneous fields.

SETUP is run like any other compiled Pascal program, by entering the Command level of the system, typing X for eXecute and typing the filename SETUP followed by a carriage return. You should see the following (user input is in **bold** type):

```
Execute what file? SETUP
INITIALIZING.....................
.............................
SETUP: C(HANGE) T(EACH) H(ELP) Q(UIT) [C3]
```

If this does not happen it may be because the setup program is not on the disk. If so, the system will display the message

```
no file SETUP.CODE
```

If neither of the above happens, something is drastically wrong. Contact UCSD. Assuming all is well, continue.

All commands to the SETUP program are invoked by typing a single letter chosen from the prompt line.

```
SETUP: C(HANGE) T(EACH) H(ELP) Q(UIT)
```

Type ‘H’ to find out what the commands at this level do. The program is self teaching, so the rest of this document explains the information SETUP was designed to change.

SETUP does not tell the system how to do random access cursor addressing on the user's terminal (for those terminals which have this capability). To allow the system to use that feature, please refer to Section 4.7 of this document.

4.3.1. Miscellaneous Information



It is interesting to note that on all PDP−11 systems, the key which generates ASCII DC1 (or control−R); functions as an alpha−lock.

HAS CLOCK
: Values: TRUE, FALSE
A real time clock is available. A real time clock module, such as the DEC KW11, may be found on

many processors. It is assumed to be a line frequency (60 cycle) clock. If available it is used by the PASCAL system to optimize disk directory updates. See section 2.1.5 `TIME` intrinsic.

STUDENT

Values: TRUE, FALSE

If true, tells the system to simplify certain parts of the system for novice use. *E.g.*, an error detected while compiling sends student back to the editor without choice.

HAS 8510A

Values: TRUE, FALSE

The system is running on a Terak 8510a hardware configuration.

HAS BYTE FLIPPED MACHINE

Values: TRUE, FALSE

True if low order byte is in bits 0−7 of words on your processor (9900, 6800, GA440) and False if high order byte comes first (PDP11, 8080, 6502).

HAS WORD ORIENTED MACHINE

Values: TRUE, FALSE

True if sequential addresses address sequential 16 bit words, False if sequential addresses address sequential 8 bit bytes.

4.3.2. General Terminal Information

HAS SLOW TERMINAL

Values: TRUE, FALSE

When this field is true, the system issues abbreviated prompt lines and messages. Suggested setting: 600 baud and under – True, otherwise False.

HAS RANDOM CURSOR ADDRESSING

Values: TRUE, FALSE

Only applies to video terminals. See Section 4.7 in order to allow the system to make use of this feature.

HAS LOWER CASE

Values: TRUE, FALSE

SCREEN WIDTH

The number of characters per line of a terminal.

SCREEN HEIGHT

The number of lines per display screen of a video terminal. Set to 0 for a hard copy terminal or other terminal in which paging is not appropriate.



NON PRINTING CHARACTER

Values: Any printing character.

What should be displayed by the terminal to indicate the presence of a non−printing character. Recommended setting: ASCII “?”.

VERTICAL MOVE DELAY

The number of NUL characters to send after a vertical cursor move. Many types of terminals require a delay after certain cursor movements which enables the terminal to complete the movement before the next character is sent. This number of NUL characters will be sent after carriage returns.

ERASE TO END OF LINE

self explanatory

ERASE TO END OF SCREEN

self explanatory

MOVE CURSOR UP

self explanatory

4.3.3. Control Key Information

The user may choose which control keys suit his particular keyboard arrangement and his taste.

Some keyboards generate two codes when some single key is pressed. If that is the case for any of the keys mentioned here, it must be noted in the field `PREFIXED[`*field−name*`]` which has either the value TRUE or the value FALSE. The prefix for all such keys must be the same and must be noted in the field `LEAD IN FROM KEYBOARD`. This feature may also be used to access control functions with two−character sequences if a user's keyboard is unable to generate many control characters. As an example, suppose the user's keyboard had a vector pad which generated the value pairs ESC “U”, ESC “D”, ESC “L” and ESC “R” for the keys for Up−Arrow, Down−Arrow, Left−Arrow and Right−Arrow, respectively. Assume also that all other keys on the keyboard generate only single codes. Then the user would give the following fields the following values:

KEY FOR MOVING CURSOR UP	ASCII “U”
KEY FOR MOVING CURSOR DOWN	ASCII “D”
KEY FOR MOVING CURSOR LEFT	ASCII “L”
KEY FOR MOVING CURSOR RIGHT	ASCII “R”
LEAD IN KEY FOR KEYBOARD	ESC
PREFIXED[KEY FOR MOVING CURSOR UP]	TRUE
PREFIXED[KEY FOR MOVING CURSOR DOWN]	TRUE
PREFIXED[KEY FOR MOVING CURSOR LEFT]	TRUE
PREFIXED[KEY FOR MOVING CURSOR RIGHT]	TRUE

KEY FOR STOP
: Console output stop character. The STOP character is a toggle; when pressed, the key will cause output to the file ‘OUTPUT’ to cease. When the key is depressed again, the write to file ‘OUTPUT’ will resume where it left off. This function is very useful for reading data which is being displayed faster than one can read.
: Suggested setting: ASCII DC3

KEY FOR FLUSH



: Console output cancel character. Similar in concept and usage to the STOP key, the FLUSH key will cause output to the file ‘OUTPUT’ to go undisplayed until FLUSH is pressed again or the system writes to file ‘KEYBOARD’. Note that, unlike the STOP key, processing continues uninterrupted while output goes undisplayed.
: Suggested setting: ASCII ACK

KEY FOR BREAK
: Typing the character BREAK will cause the program currently executing to be terminated with a run−time error immediately.
: Suggested setting: Something difficult to hit accidentally.

KEY TO END FILE
: console end of file character. When reading from the files `KEYBOARD` or `INPUT` or the unit ‘`CONSOLE:`’, this key sets the Boolean function `EOF` to TRUE. See section 2.2.4 EOF intrinsic.
: Suggested setting: ASCII ETX

KEY TO DELETE CHARACTER

Each time you press this key one character is removed from the current line, until nothing is left on that line.
Suggested setting: ASCII BS

KEY TO DELETE LINE
Pressing LINE DELETE will cause the current line of input to be erased.
Suggested setting: ASCII DEL

The rest of this section contains information only of interest to users who are using video display terminals with a selective erase capability and may be safely ignored by users having any other kind of terminal, such as hard copy terminals or storage tube terminals.

KEY TO MOVE CURSOR UP
KEY TO MOVE CURSOR DOWN
KEY TO MOVE CURSOR LEFT
KEY TO MOVE CURSOR RIGHT
These keys are used by the screen oriented editor to control the basic motions of the cursor. If the keyboard has a vector pad, set these fields to the values it generates, otherwise, we suggest choosing 4 keys in the pattern of a vector pad and use the control codes which correspond to them, for example the keys ‘O’, ‘.’, ‘K’ and ‘;’ on most keyboards encircle an imaginary vector pad. You may wish to use a prefix character before such keys as described above.



EDITOR ESCAPE KEY
The key which, in the system screen oriented editor, is to be used to escape from commands, reversing any action taken.
Suggested setting: ASCII ESC

EDITOR ACCEPT KEY
The key which, in the system screen oriented editor, is to be used to accept commands, making permanent any action taken.
Suggested setting: ASCII ETX

4.3.4. Video Screen Control Characters

This section describes the characters which, went sent to the terminal by the computer, controls the terminals actions. You should consult the manual for your terminal to find the appropriate values. If a terminal does not have one of these characters, the field should be set to 0 unless otherwise directed.

Some screens require a two character sequence to exercise some of their functions. If the first character in all of these sequences is the same, it can be set as the value of the field `LEAD IN TO SCREEN` and for each *field−name* which requires that prefix, the user must set the field `PREFIX[`*field−name*`]` to TRUE. For example, suppose `ERASE TO END OF LINE` and `ERASE TO END CF SCREEN` were respectively performed by the sequences ESC “L” and ESC “S” but all the other screen controls were single characters. The user would then set the following fields to the following values:

LEAD IN TO SCREEN	ASCII ESC
ERASE TO END OF LINE	ASCII “L”
ERASE TO END CF SCREEN	ASCII “S”

PREFIXED[ERASE TO END OF SCREEN] TRUE
PREFIXED[ERASE TO END OF LINE] TRUE

ERASE TO END OF SCREEN
: The character which erases the screen from the current cursor position to the end of the screen.

ERASE TO END OF LINE
: The character which, when sent to the screen, erases all characters from the current cursor position to the end of the line the cursor is on.

ERASE LINE
: The character which, when sent to the screen, erases all the characters on the line the cursor is currently on.

ERASE SCREEN
: The character which, when sent to the screen, erases the entire screen.

BACKSPACE
: The character which, when sent to the screen, causes the cursor to move space to the left.

MOVE CURSOR HOME



: The character which moves your cursor to the upper left of the current page. **Important:**: If your terminal does not have such a character, set this field to CARRIAGE RETURN, ASCII mnemonic CR.

MOVE CURSOR UP
MOVE CURSOR LEFT
: The characters which move your cursor non−destructively one space in those directions.

4.3.5. Quit

The quit mode of SETUP gives many options: Memory update, which places the definitions in the memory cells which are appropriate. Disk update, which creates the file `NEW.MISCINFO`. Return, which takes the user back to setup, and Exit, which returns the user to the Pascal command level.

4.3.6. Quick Reference Summary

```
BACKSPACE
EDITOR ACCEPT KEY
EDITOR ESCAPE KEY
ERASE LINE
ERASE SCREEN
ERASE TO END OF LINE
ERASE TO END OF SCREEN
HAS 8510A
HAS BYTE FLIPPED MACHINE
HAS CLOCK
HAS LOWER CASE
HAS RANDOM CURSOR ADDRESSING
HAS SLOW TERMINAL
HAS WORD ORIENTED MACHINE
KEY FOR BREAK
KEY FOR FLUSH
KEY FOR STOP
KEY TO DELETE CHARACTER
KEY TO DELETE LINE
KEY TO END FILE
KEY TO MOVE CURSOR DOWN
```

```
KEY TO MOVE CURSOR LEFT
KEY TO MOVE CURS9R RIGHT
KEY TO MOVE CURSOR UP
LEAD IN FROM KEYBOARD
LEAD IN TO SCREEN
MOVE CURSOR HOME
MOVE CURSOR RIGHT
MOVE CURSOR UP
NON PRINTING CHARACTER
```




```
PREFIXED [DELETE CHARACTER]
PREFIXED [EDITOR ACCEPT KEY]
PREFIXED [EDITOR ESCAPE KEY]
PREFIXED [ERASE LINE]
PREFIXED [ERASE SCREEN]
PREFIXED [ERASE TO END OF LINE]
PREFIXED [ERASE TO END CF SCREEN]
PREFIXED [KEY FOR BREAK]
PREFIXED [KEY FOR FLUSH]
PREFIXED [KEY TO MOVE CURSOR DO?IN]
PREFIXED [KEY TO MOVE CURSOR LEFT]
PREFIXED [KEY TO MOVE CURSOR RIGHT]
PREFIXED [KEY TO MOVE CURSOR UP]
PREFIXED [KEY FOR STOP]
PREFIXED [KEY TO DELETE CHARACTER)
PREFIXED [KEY TO DELETE LINE)
PREFIXED [KEY TO END FILE]
PREFIXED [MOVE CURSOR HOME]
PREFIXED [MOVE CURSOR RIGHT]
PREFIXED [MOVE CURSOR UP]
PREFIXED [NON PRINTING CHARACTER]
SCREEN HEIGHT
SCREEN WIDTH
STUDENT
VERTICAL MOVE DELAY
```



4.4. Bootstrap Copier

Version I.5, September 1978



The bootstrap copier `BOOTER.CODE` asks for the unit number 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 disk name, 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).



4.5. Patch/Dump

Version I.5, September 1978



PATCH is a utility which was written as a personal piece of software, and has become part of the soul of the system. Even in the wonderful world of Pascal programming, it seems that the need to see disk blocks in the not so wonderful world of HEX remains. The usefulness of this proves itself over and over again. Usually this pertains to studying the output of a Pascal program which has created a file of some structured type, however the data in the output file just doesn't seem right. Patch comes to the rescue. Patch lets you see just exactly what bits are where, arid even lets you change them to be the way they should be.

On X(ecuting PATCH, the prompt line 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 prompt line changes with each command. The prompt line always reflects the commands available at any given time, and no more. The full prompt line 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 unit number. 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 the 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 prompt line 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 were possible, and hexadecimal values elsewhere. The prompt line 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 prompt line 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 prompt line:

```
Stuff for how many bytes:
```

Key a number from 0 to 512. Type carriage return to cause patch to accept the number, the prompt line 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 Patch write 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 'F' to PRINT, 'Q' to QUIT.

A( Input File
B( Begin Block 11
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 Number 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 Patch write 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:



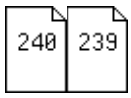

```
This procedure writes any number of blocks from an existing
file to a new file, unchanged. Simply specify the necessary
parameters Type 'P' to PUT, 'Q' to QUIT

I(nput File
S(tart Block
N(umber of Blocks

O(utput File
```

The protocol for changing the fields at this level is the same as that for the Patch write level. The Whole write 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 Patch write and Whole write levels remember their vital parameters across sessions (while remaining in Patch). The Console level will clear all memory of the session. The Patch write level paginates its output, after each block written, a form−feed is generated (specifically `PAGE(OUTPUTFILE)`).

4.6. RT11 to Pascal Conversion Kit

Version I.5, September 1978



The utility file labeled RT11TOEDIT is intended for use with RT−11 disks. It assumes the presence of an RT−11 directory spanning blocks 6–7. When the file is executed it asks the user to specify the Pascal system unit number of the vo1ume 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.

4.7. GotoXY Binder

Version I.5, September 1978



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

An example of a GOTOXY procedure for a relatively stupid terminal follows. More intelligent terminals will require less effort to have the proper cursor addressing happen. It is suggested that you might want to fill an array or string with the appropriate characters to cause your terminal to do its absolute addressing, and then UNITWRITE the stream all at once. This will improve the performance of the screen editor noticeably. An example of this for the Datamedia 1520 follows the example for the DECscope VT−50.

If the GOTOXY cursor addressing scheme for the terminal is not there, create one. The procedure may not be named GOTOXY because this identifier is predeclared at the “$U−” level of compilation.

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

Assumptions:

1. A screen terminal
2. A PASCAL system
3. The upper left−hand corner of the screen is X=0, Y=0.
4. GOTOXY corrects for bad input data.

See Section 2.1.2 for more information on GOTOXY.

Example:

The first example, for a DECscope VT−50, a terminal with very primitive cursor addressing, writes each character to the screen individually.

```
(*$U-,S+*)(* The pseudo comments inform the compiler of the correct
             state to be in for compiling this little routine. *)
PROGRAM DUMMY;
```




```
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-5O, GOTOXY needs to be implemented by:
     send the cursor home, 0,0 *)
  WRITE(CHR(27), 'H');
  (* While TAB is meaningful, use it to move the cursor. *)
  WHILE X > 8 DO
    BEGIN
      WRITE(CHR(9));
      X := X - 8;
    END;
  (* Finish off what portion of the x coordinate could not be absorbed
     with TAB characters. *)
  WHILE X > 0 DO
  BEGIN
    WRITE(CHR(27), 'C');
    X := X - 1
  END;
  (* send line-feeds to access the y coordinate *)
  WHILE Y > 0 DO
  BEGIN
    WRITE(CHR(10));
    Y := Y - 1
  END

BEGIN
  (* This dummy body of the operating system is needed to keep the
     Pascal compiler happy about having complete programs to compile.
     The method used for 'binding' the GOTOXY procedure is somewhat
     unclean, and only the code for the above procedure is used by
     the binder to add to the SYSTEM.PASCAL file. *)
END.
```

The next example, for a Datamedia 1520, shows how to collect all of the characters into a single array of characters and write it all to the terminal in a single system call.

```
PROGRAM DUMMY;

PROCEDURE ITSGOTOXY(X, Y: INTEGER);
VAR
    T: PACKED ARRAY[O..2] OF CHAR;
BEGIN
  T[0] := CHR(30); (* RS is Datamedia's absolute cursor address flag *)
```




```
  (* Set appropriate character for x coordinate. *)
  IF X < 0 THEN T[1] := CHR(32)
  ELSE
    IF X > 79 THEN T[1] := CHR(32 + 79)
    ELSE
      T[1] := CHR(X + 32);
  (* Set appropriate character for y coordinate. *)
  IF Y < 0 THEN T[2] := CHR(32)
  ELSE
    IF Y > 23 THEN T[2] := CHR(32 + 23)
```

```
    ELSE
      T[2] := CHR(32 + Y);
  (* Send the cursor where it belongs... WHAPPO! *)
  UNITWRITE(1, T, 3)
END;

BEGIN
  (* Same comment applies. *)
END.
```



4.8. Duplicate Directory

Version I.5, September 1978

COPYDUPDIR



This program will copy the duplicate directory into the primary directory location. If the disk is not currently maintaining a current directory the program will tell you so. To use this program e(x)ecute COPYDUPDIR. The program will ask for the drive in which the copy is to take place (4 or 5). If no duplicate directory is found it will tell you after you indicate the drive unit. If the duplicate is found then it will ask you if you are sure you want to destroy the directory in blocks 2–5. A ‘Y’ will execute the copy, any other character will abort the program.

MARKDUPDIR

This program will mark a disk that is currently not maintaining a duplicate directory so that it will. Caution must be exercised to be sure that blocks 6–9 are free for use. If they are not one must rearrange the files as to make them free. One can tell if they are 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>        14             6
SYSTEM.PASCAL   31 30-Aug-78 10 Codefile
```

Both of the above cases indicate disks that have not been marked. Below is the directory of a properly marked disk.




```
SYSTEM.PASCAL   31 30-Aug-78 10 Codefile
```

To execute this program e(X)ecute MARKDUPDIR. The program will ask you which unit contains the disk to be marked (4 or 5). The program will check to see if it thinks that the blocks 6–9 are free. If the program doesn't think so it will ask you if you are sure they are free? Typing ‘Y’ will execute the mark, any other character will abort the program. Be sure that the space is free before marking it as a duplicate directory.

4.9. P–Code Disassembler

Version I.5, September 1978



The disassembler reads a standard UCSD code file and outputs symbolic pseudo–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 and optimizing programs, and provides a source of further information regarding some of subtleties of our implementation of Pascal. All statistics gathered are collected by making a pass through the code file instead of collecting them while the code file is actually running.

4.9.1. Disassembly

The Disassembler reads a code file that has been generated by the UCSD Pascal Compiler. If a program USES a UNIT the disassembly will include the UNIT only if the code file has been linked. Assembly routines linked into a Pascal host will never be included in the disassembly.

The Disassembler is invoked by executing `DISASM.I5` and requires the file `OPCODES.I5` to be on the system disk. The Disassembler will first prompt for an input code file, the suffix `.CODE` being assigned 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 t: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:=O;
 8 1 1:C  5   TOMORROW:=FALSE;
 9 1 1:C  8   REPEAT
1O 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 I 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             A803
     1    1     5(005):    SLDC         0             00
     1    1     6(006):    SRO          4             AB04
     1    1     B(OOB):    SLDO         3             EA
     1    1     9(009):    SLOC         1             01
     1    1    10(OOA):    ADI                        82
     1    1    11(008):    SRO          3             AB03
     1    1    13(000):    LOD          1  3          B60103
     1    1    16(010):    LCA         42 'Disassembly - a step backwards...'
     1    1    60(03C):    SLDC         0             00
     1    1    61(030):    CXP     WRITESTR           C00013
     1    1    64(040):    CSP     IOCHECK            9E00
     1    1    66(042):    LOD          1  3          B60103
     1    1    69(045):    CXP     WRITELN            C00016
     1    1    72(048):    CSP     IOCHECK            9E00
     1    1    74(04A):    SLDO         4             E3
     1    1    75(043):    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 mnemonics is given in section 3.4. The actual code that exists in the file is given in hex in the rightmost column. The parameters to CXPs and CSPs 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 (XJPs). The block number and byte offset of the start of the procedure are given relative to the start of the code file. Thus this procedure starts at block 1, offset 0 of the code file. The segment dictionary resides in block 0 for all code files.

4.9.2. Data Segment Reference Statistics

The fourth prompt the Disassenbier 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 fran optimizing such a small program but the general idea can still be presented. By using the compiled listing shown in figure 1 one can match offsets to variables as such:

variable	*offset*
I	3
TOMORROW	4
COMMENT	5

Now by using the figures in figure 3 one can see that offset 3 or the variable I occurs most frequently and thus deserves it's position. This same idea carried out on a large program may result in substantial size savings. Notice that offset 6 never occurs and thus is not included in the statistics in figure 3.

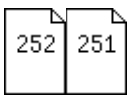
The prompt for the output file for these statistics occurs after the disassembly has been completed. If you elect to collect these statistics you will be taken into the Segment and Procedure Guides as described in the previous section except that the pranpt requests the selection of a data segment on which to collect statistics. In the Procedure Guide, “L” gives a listing of all the procedures in the selected segment by number, lex level, and data segment size. After the selection of a data segment, processing continues, as described in the previous section, from the point after the data segment question.

4.9.3. Opcode, Procedure Call, and Jump Statistics

These statistics are collected as an aid in optimizing the architecture of P−Code and although they are interesting to look at they are of no real use to the typical user. For this reason they will be described only superficially.

Each opcode is given with a complete breakdown of which bit was most significant for each operand on any given occurrence of the opcode. These are presented in terms of totals and percentages of the number of occurrences of the opcode. In addition a histogram of the opcode occurrence as a percentage of the total number of opcodes disassembled runs along the right−hand 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 junps. 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.



4.10. Library Map

Version 1.5, September 1978

The program LIBMAP produces a map of a library (or code) file and lists the linker information maintained for each segment of the file. In the case of segments ~dnich are Pascal Units the map file will also contain the interface section of the Unit. See section 3.3.2 for greater detail.



The program first prompts for a library file name. As in the linker, this may be an asterisk to indicate ‘*SYSTEM.LIBRARY’. The “.CODE” suffix may be suppressed by appending a period to the full file name.

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.

Example

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)

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

4. Utility Programs

4.2. Librarian 221
4.3. Setup — System Reconfiguration 225
4.4. Bootstrap Copier 233
4.5. Patch/Dump 235
4.6. RT11 To Pascal Conversion Kit 239
4.7. GotoXY Procedure Binder 241
4.8. Duplicate Directory 245
4.9. P–Code Disassembler 247
4.10. Library Map 253

5. Tables



<un&,~n~d ~v~t1e~er)

C

<id~y~tU~i,g–)

~unsI. yrted consto.yit)

I

L~~J

Page 255



<constorit)

<ur~igr~d v~umber>

–~n~L~L~er 1

Page 256



<tLgtd ii.gt>

type LcJeI,tL(tier ~F

~~o–E~ ~

<sirip~te type>

~ t~~e lderttLfLerj~__~.~~

~st~ntj ~st~nt~ J

r

Page 257



<~o~tor>

_– –

_____–.

L~1

~ –~

– {~pr ~s I a,, j

(t.err~)

Page 258



<expr~sst on)

<pc~ro~eter *ttst)

Page 259



<st~terient>

vo.rtobte LdentLcter

Page 260



1~ c~~)

Page 26~t



<c~rtpL ~tLDn>

_____ <procedure>

_____ PjtdentLflerF~ ~{P~r3J1eter tLstf ~

po.ro.i~'tet.er `ttst

: type ~ent~.cLer

Page 262



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

Version 1.5 September 1978

0 System error FATAL

1 Invalid index, value out of range (XINVNDX)

2 No segment, bad code file (XNOPRCC)

3 Procedure not present at exit ti.ne (XNOEXIT) 14 Stack overflow (XSTKOVR)

5 Integer overflow (XIWIOVR)

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)

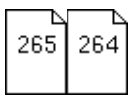
13 String too long (XS2LONG)

114 Halt, Break~int (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)

Page 263



******~***** ********** * TABLE 2 * * IORESULTS * ********** **********

Version 1.5 ~ptember 1978

0 No error

1 Bad Block, Parity error (CRC)

2 Bad Unit Number

3 Bad Mode, Illegal operation 14 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 nane

8 No room, insufficient space

9 No unit, No such voltxne on line

10 No file, No such file on vohzne

11 L~plicate file

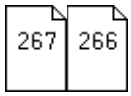
12 Not closed, attempt to open an open file

13 Not open, attempt to access a closed file

114 Bad format, error in reading real or integer

15 Ring buffer overflow

Page 265



********** ,****,********* * TABLE 3 * * UNITNUMBERS * *******I*** **********

Version 1.5 Septenber 1978

NUMBER 1~LUME NAME

0 <empty>

1 CONSOLE

2 SYSTERM

3 GRAPHIC

floppyC

5 floppyl

6 PRINTER

7 R~1IN

8 R~O~T~

9 blockl

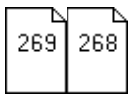
10 block2

11 block3

12 b1ock~4

Devices 9 − 12 are block−structured devices, in most cases (RK−05).

Page 267



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

23: `Irnp1~entation' expected 2L1: `Unit' expected

50: Error in constant 51: `: :` expected 52: `THEN' expected 53: `UNTIL' expected 5~: `tO' expected 55: `TO' or `DO~1NTO' expected in for stat~nent 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 1O~l: Undeclared identifier

Page 271



105: sign not allowed 106: Nunber expected 107: Incanpatible subrange types 108: File not allowed here 109: Type must not be real 110: <tagfield> type must be scalar or subrange lii: Incanpatible with <tagfield> part 112: Index type must not be rca]. 113: Index type must be a scalar or a subrange 1114: 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: W~satisified forward reference 118: Forward reference type identifier in variable declaration 119: Re−specified parans not OK for a forward declared procedure 120: Function result type must be scalar, subrange or pointer 121: File value paraneter not allowed 122: A forward declared function's result type can't be re−specified 123: Missing result type in function declaration 12~4: F−format for reals only 125: Error in type of standard procedure parameter 126: Nunber of par~neters 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 c~nparison not allowed 1314: 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 c~npatible 138: Type of variable is not array 139: Index type is not c~i,patible with the declaration 1140: Type of variable is not record 1141: Type of variable must be file or pointer 1112: 111eg31 parameter solution 143: Illegal type of loop control variable 11414: Illegal type of expression 1145: Type conflict 1146: Assignment of files not allowed 1147: Label type inc~patible with selecting expression 1148: Subrarige bounds must be scalar 1149: Index type must be integer 150: Assignment to standard function is not allowed

151: Assignment to formal function is not allowed 152: Mo such field in this record 153: Type error in read 1514: Actual parameter must be a variable 155: Control variable cannot be formal or non−local

Page 272



156: ~4iltidefined 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 1614: Substition of standard proc/func not allowed 165: Multidefiried label 166: Multideclared label 157: Undeclared label 168: Undefined label 169: Error in base set 170: Value parameter expected 171: Standard file was re−declared 172: Undeclared external file 17)4: Pascal function or procedure expected 182: Nested units not allowed 183: External declaration not allowed at this nesting level 1814: External declaration not allowed in interface section 785: 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 19)4: 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 2014: 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 25~4: 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 assiriged is out of bounds 3014: Element expression out of range

Page 273



398: Imp1e~n~tation restriction 399: Imp1~nentation restriction

1400: Illegal character in text 1401: Lk~expected end of input A402: Error in writing code file, not enough roan 1403: Error in reading include file 1404: Error in writing list file, not enough room 1405: Call not allowed in separate procedure 1406: Include tile not legal

Page 274



********** ********** * TABLE 6 * * ASS~BLER SYNTAX ERRORS * ********** **********

Version 1.5 September 1978 This section lists all the general errors found in the ERRORS file, specific machine errors are found in the sections below dealing with machine specifics.

1: Undefined label 2: Operand out of range 3: Must have procedure name 4: Number of parameters expected 5: Extra garbage on line 6: Imput line over 80 characters 7: Not enough ifs 8: Must be declared in ASECT before use 9: Identifier previously declared 10: Improper format 11: EQU expected 12: Must EQU before use if not to a label 13: Macro identifier expected 14: Word addressed machine 15: Backward ORG not allowed 16: Indentifier expected 17: Constant expected 18: Invalid structure 19: Extra special symbol 20: Branch too far 21: Variable not PC relative 22: Illegal macro parameter index 23: Not enough macro parameters 24: Operand not absolute 25: Illegal use of special symbols 26: Ill–formed expression 27: Not enough operands 28: Cannot handle this relative 29: Constant overflow 30: Illegal decimal constant 31: Illegal octal constant 32: Illegal binary constant 33: Invalid key word 3)4: 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 & con~ents 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

Page 275



44: No local labels in ASECT 45: Expected key word 46: String expected 47: Bad block, parity error (crc) 48: Bad unit nunber 149: 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 roan, insufficient space 55: No unit, no such volunn on–line 56: No file, no such file on voliinri 57: D.~plicate file 58: Not closed, attempt to open an open file 59: Not open, attempt to ~cess 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 ur~efined labels

Z80 Based machines

For constants, Hex is the default type, a `B' defines binary cx. bOlOS a `.` defines decimal cx. 5674.

Location Counter (LC) = $

All reserved words may not be used for any other purpose such as an identifier. For example, the reserved word "C" currently is being used as a register and in a condition code, therefore it may not be used for any other purpose (this is contrary to usual Zilog assembly language, but is restricted in the UCSD assembler).

Specific error messages;

76: Incorrect operand format 77: Close paren ")" expected 78: Corrna "," expected 79: Plus "+" expected 80: Open paren "(" expected 81: Stack pointer "SF" expected 82: "HI.." expected 83: Illegal "CC" condition code 8~: Register "C" expected 85: Register "R" expected 86: Register "A" expected

Page 276



PDP11 Base:1 machines:

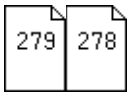
For constants, Octal is the default type for both input and output, a `H' defines hexadecimal ex. 056H a `.` defines decimal ex. 5~46. a `B' defines binary ex. 1001B

Location Cotnter (LC) = *

Specific error messages:

76: Closing paren ")" expected 77: Register expected 78: Too many special symbols 79: Unrecognizable operand 80: Register reference only 81: First operand must be a register 82: Comma expected 83: Unimplimented instruction 8L4: Must branch backwards to label

Page 277



********** **********,** * TABLE 7 * * M~ericari Standard Code for Information Interchange * **********
****

Version 1.5 Sept~ber 1978

0

000

00

NUL

32

0~40

20

SF

64

100

140

96

1~40

60

`

1

001

01

SQ-I

33041

21

!

65

101

141

A

97

1141

61

a

2

002

02

STX

314

0142

22

66

102

42

B

98

1142

62

b

3

003

03

ETX

35

0143

23

II

67

103

143

C

99

1143

63

c

14

0014

014

ECT

36

01414

24

$

68

10~4

1414

D

100

11414

614

d

5

005

05

ENQ

37

0145

25

%

69

105

145

E

101

1145

65

e

6

006

06

ACK

38

0146

26

&

70

106

146

F

102

1146

66

f

7

007

07

BEL

39

0~47

27

`

71

107

147

G

103

1147

67

g

8

010

08

BS

40

050

28

(

72

110

148

1−1

10~4

150

68

h

9

011

09

H~

~41

051

29

)

73

111

149

I

105

151

69

1

10

012

OA

LI

142

052

2A

*

714

112

14A

J

106

152

6A

j

11

013

06

VT

143

053

26

+

75

113

11B

K

107

153

66

k

12

0114

OC

FF

1414

0514

2C

,

76

1114

~4C

L

108

1511

6C

1

13

015

OD

CR

145

055

2D

–

77

115

lID

M

10.9

155

6D

m

1~4

016

OE

SO

46

056

2E

.

78

116

14E

N

110

156

6E

n

15

017

OF

SI

147

057

2F

1

89

117

~4F

0

111

157

6F

o

16

020

10

DLE

~48

060

30

0

80

120

50

P

112

160

7C

p

17

021

11

DC1

249

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

0214

14

DCII

52

0614

314

14

814

1214

514

T

116

16'4

714

t

21

025

15

NAK

53

065

35

5

85

125

55

U

117

165

75

u

22

026

16

SYN

514

066

36

6

86

126

56

V

118

166

76

V

23

027

17

ETh

55

067

37

7

87

127

57

W

119

167

77

w

2~4

030

18

CAN

56

070

38

8

88

130

56

X

120

17C

78

x

25

031

19

57

071

39

9

89

131

59

1

121

171

79

y

26

032

1A

SUB

58

072

3A

:

90

132

5A

Z

122

172

7~

z

27

033

lB

ESC

59

073

36

;

91

133

SB

1

123

173

76

(

28

0314

1C

FS

60

0714

3C

<

92

1314

SC

\

1214

1714

7C

29

035

1D

OS

61

075

3D

93

135

5D

3

125

175

7D

)

30

036

1E

RS

62

076

3E

>

914

136

SE

126

176

7E

31

037

iF

US

63

077

3F

?

95

137

SF

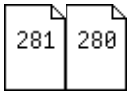
127

177

7F

DEL

Page 279



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

* TPIBLE 8 * * P–MACHINE OP–CODES * ********** **********

Version 11.0 February 1979

000000 SLDC 0

100101 SLDC 1

126 176 7E SLDC 126 127 177 7F SLDC 127 128 200 80 ABI 171 253 AB SRO 214 326 D6 XIT 129 201 81 ABR 172 2514 AC XJP 215 327 D7 NO? 130 202 82 ADI 173 255 AD RNP 216 330 D8 SLDL 1 131 203 83 ADR 1714 256 AE CI? 217 331 D9 SLDL 2 132 2014 814 AND 175 257 AF EQU 218 332 DA SLDL 3 133 205 85 DIF 176 260 BO GEQ 219 333 DB SLDL LI 1314 206 86 DVI 177 261 31 GR'r 220 3314 DC SLDL 5 135 207 87 DVR 178 262 B2 LDA 221 335 DD SLDL 6 136 210 88 CHK 179 263 B3 LDC 222 336 DE SLDL. 7 137 211 89 FLO 180 26~4 B'4 LEQ 223 337 OF SLDL 8 138 212 8A FLT 181 265 35 LES 2214 3140 E0 SLDL 9 139 213 88 INN 182 266 B6 LOD 225 3141 El SLDL 10 1140 2114 8C INT 183 267 B7 NEQ 226 3142 E2 SLDL 11 1141 215 8D IOR 18'4 270 B8 STR 227 3143 E3 SLDL 12 1~42 216 BE MOD 185 271 B9 UJP 228 31414 ELI SLDL 13 1J43 217 8F MPI 186 272 BA LDP 229 3145 E5 SLDL 114 1)44 220 90 MPR 187 273 BB Si'? 230 3146 E6 SLDL 15 1145 221 91 NGI 188 2714 BC LEM 231 3147 E7 SLDL 16 1146 222 92 NGR 189 275 BD Sill 232 350 E8 SLDO 1 1147 223 93 NOT 190 276 BE LDB 233 351 E9 SLDO 2 1148 224 914 SRS 191 277 BF STB 2314 352 EA SLDO 3 1149 225 95 SBI 192 300 CO IX? 235 353 EB SL~J 14 150 226 96 SBR 193 301 Cl RB? 236 3514 EC SLW 5 151 227 97 SGS 1914 302 C2 CBP 237 355 ED SLDO 6 152 230 98 SQl 195 303 C3 EQUI 238 356 EE SLCIJ 7 153 231 99 SQR 196 30)4 CLI GEQI 239 357 EF SLDO 8 1514 232 9A STU 197 305 C5 GRTI 2140 360 FO SLDO 9 155 233 93 I)~ 198 306 C6 LLA 2141 361 Fl SLDO 10 156 2314 9C UNI 199 307 C7 LDCI 2142 362 F2 SL~ 11 157 235 9D 200 310 C8 L.EQI 2143 363 F3 SLDO 12 158 236 9E Cs? 201 311 C9 LESI 24~4 3614 N SL~J 13 159 237 9F LDCN 202 312 CA LDL 2145 365 F5 SLDO 114 160 2140 AO ADJ 203 313 CB NEQI 2146 366 F6 SLW 15 161 2141 Al FJP 20J4 31'! CC STL 2147 367 F7 SLDO 16 162 2142 A2 INC 205 315 CD CX? 248 370 F8 SIND 0 163 2143 A3 IND 206 316 CE CLP 2149 371 F9 SIND 1 164 2144 A4 IXA 207 317 CF CC? 250 372 FA SIND 2 165 2145 A5 LAO 208 320 DO LPA 251 373 FB SIND 3 166 2146 A6 LSA 209 321 Dl 252 3714 FC SIND `14

Page 281



167 2L17 A7 210 322 D2 253 ~75 F~ SIND 5 168 250 AS MOV 211 323 D3 WJ 25~4 376 FE SIND 6

169 251 A9 LD0 212 32L~ DL~ NFJ 255 377 FF SIND 7 170 252 AA SPIS 213 325 P5 8F~ 170 252 AA SAS 213 325 D5 BFT

Page 282

A. Index



********** ********* * Appendix A * * Index * ********** *********

ARRAY, 115 FUNCTION, 1014 ASSB~1BLER 4, 97, 98, 111 GENERAL ERRORS, 275 BAD BL~CK SCAN, 25 GET, 12, 1214 BANISH, 59 GOTO, 79, 86, 89, 1140 BASIC, 85 GOTOXY, 127, 156, 226, 2141 BLCCK, 115 GRAPHICS, 159 ELCCKN1R~1BER, 115 HALT, 127, 156 BLOCKREAD, 122, 138, 155 HEAP, 1314 BUlKS, 30 I~EARCH, 156 BLOCKWRITE, 122, 138, 155 IMPL~1ENTATION, 167 BOOTSTRAP 233 INCLUDE, 79, 98, 112 BCWLES, 1 INDENTATION CODE, 163 CASE STAT~1ENTS, 133 INDEX, 115 CHANGE, 18 INITIALIZE DISKS, 26 CHARACTER, 115 INPUT, 136, 147 CLOSE,– – 123, 1~47, 155 INSERT, 314, 39, 55, 69, 118, 156 COMMENTS, 133 INTERACTIVE, 1147 COMPILED LISTING, 81 INTERFACE, 167 COMPILER, 3, 77, 85 INTRINSICS, 155 CONCAT, 117, 155 10–ERRORS, 1214, 263, 265 CONDITIONAL ASSaIBLY, 108 IORESULT, 79, 1214, 156, 186 CONTROL CHARACTERS, 63 ~ 55 COPY, 55, 118 KEYBOARD 136, 1147 CP/M, 5 KRUNCH, 27 CURSOR, 31, 37, 66, 68 L2 EDITOR 57 DATE, 24 LENGTH, 117, 152, 156 DEBUGGER, II, 75, 78 DELETE, 35, 141, 42, 55, 56, 69, 118, 155 DESTINATION, 115 DIRECTIVES, 102 LIBRARY, 173 DIRECTORY, 15, 17 LINKER, 14, 91, 173 DISK ERROR, 25 LIST DIRECTORY, 15, 17 DISK SIZE, 30 L~K, 123 DISK SPACE, 27 L~, 127 DLE, 163 LONG INTEGERS, 118, 181 DRAWLINE, 159 MACRC, 101 EDITOR, 2, 31 MACROS, 71, 107 EOF, 123, 136, 140 MAKE, 28 EOLN, 123, 136, 1140, 1~47 MARK, 127, 156 EXAMINE, 26 MARKERS, 38, 149, 55, 5~ EXCHANGE, 69 M~1AVAIL, 128, 156 EXECUTE, 3 M~1ORY ALLCCATION, 1~14 EXIT, 1140, 155 MB~1ORY MANAG~ENT, 127 EXPRESSION, 115 MOVELEFT, 131, 156 EXTENDED LIST, 17 MOVERIGHT, 131, 156 EXTERNAL, 91, 100, 173 NB4, 114, 136 FILE, 121, 1214, 1146 NEXT, 59 FILEID, 115 NORMAL, 123 FILENAMES, 7, 9, 32 NUMBER, 115 FILER, 2, 3, 7 OUTPUT, 136, 147 FILES, 137 PACK, 1145 FILLCHAR, 132, 11414, 156 PACKED ARRAYS, 1142 FIND, 1414, 145, 55, 68 PACKED –RECORDS, 11414 FORWARD, 47, 173 PACKED VARIABLES, 142 PAGE, 125

Page 283



PASCAL, I TITLE, 116 PDP–11, 97 TOKEN, PDP11, 5, 188 TRANSFER, 21 P05, 117, 156 TREESEARCH, 157 PR~IX, 25 TRUNC, 162 PROCEDURE, lOLl UNIT, 63, 914, 167, lEE PROGRAM HEA~ING3, 1146 UNITBUSY, 122, 157 PSEUDO CD~fl1ENTS, 78 UNITCLEAR, 122 PSEUDO–OPS, 102 UNITNLI1BER, 116, 267 PURGE, 123 UNITREA~, 121, 157 FUr, 1214 UNIT~AIT, 122, 157 RQ'RcFTEN, 127, 156 UNITWRITE, 121, 157 QUIET, 82 UNPACK, 1145 QUIT, 14, 52, 55, 66 UNTCLEAR, 157 RADAR, 159 USE LIBRARY, 83 RANGECHECK, 82 USES, REAG, 124, 1246, 153 VCLID, 116 READLN, 1146 VOLUIE, 25 RELBLDCK, 115 VOLUIE NAMES, 7 RELEASE, 127, 156 VOLUIE NUMBERS, 267 RE~CVE, 20 VCLU1ES, 15 REPLACE, 4414, 46, 56 WKAT, 15 RESET, 121, 147, 148, 156 WILDCARIS, 10 RESTRICTIONS, 154 WCRD PROCESSING, 39, 148, ~, 55 REWRITE, 121, 1147, 1448, 156 WCR~ILE, 2, 8, 33, 36, 63, 77, ~3, ~ RT–11, 239 WRITE, 1214, 153 3 WRITELN, 153 13 YALOE SU~?1ARY, 73 SCAN, 131 Z8C, 5, 97, 188 SCREEN, 116 ZERC, 28 SCREEN CCNTRCL, 5, 82, 127 SCREENCONTROL, 225, 2141 SEEK, 125, 138, 156 SE~NT PROCEDURE, 1149, 165 SET, 149 SETS, 150 SYPLVARIABLE, 115 SIZE, 116 SIECF, 127, 1414, 156 S3UBCE, 116 STE, 118, 156, 182 STRING, 86, 116, 117 STRIN05, 151 &IAPPING, 82 SYNTAX ERRORS, 271 SYSCOM, 77 SYST~1 Ct~'1PiLATICN, 63 SYST~1.C~4PILER, 85 SYSTDI.LIBRARY, 14, 81, 83, 91, 1O~4 SYST~1.WRK.COt~E, 3, 36, 77, 93, 98 TEXT, 1146, 163 T2IE, 127, 157

Page 284