PSCOPE-86 HIGH-LEVEL PROGRAM DEBUGGER USER'S GUIDE (SUPPORTING THE iRMX™-86 OPERATING SYSTEM)

Order Number: 165496-001

PSCOPE-86 HIGH-LEVEL PROGRAM DEBUGGER USER'S GUIDE (SUPPORTING THE iRMX™-86 OPERATING SYSTEM)

Order Number: 165496-001

REV.	REVISION HISTORY	DATE
-001	Original issue.	8/84

PREFACE

The *PSCOPE-86 High-Level Program Debugger User's Guide* contains the following:

- An introduction to the PSCOPE high-level program debugger.
- An introductory PSCOPE debugging session.
- A description of PSCOPE's internal, screen-oriented editor.
- Descriptions and examples of PSCOPE command language.
- Instructions for loading and executing user programs.
- An introduction to debug objects and symbols, as well as an explanation of the commands used to manipulate debug objects.
- A description of PSCOPE utility commands.
- PSCOPE error messages.
- Instructions for configuring non-Intel terminals to use PSCOPE.
- PSCOPE syntax.
- PSCOPE reserved keywords.

Manual Organization

This manual contains 11 chapters and 9 appendixes, as follows:

Chapter 1 defines PSCOPE and describes the program development process and the host system execution environment. Chapter 1 also details PSCOPE's major functions and provides an introductory sample session.

Chapter 2 explains how to invoke the debugger and discusses each of the invocation controls. Chapter 2 also describes invocation error messages, how to enter commands from the keyboard, and PSCOPE's internal editor.

Chapter 3 describes PSCOPE command lines and is an overview of tokens, symbolic references, and symbol object types. Chapter 3 defines the operands and operators used in expressions and explains expression types, stepping, and the GO command.

Chapter 4 details how to load programs and control execution using the GO command.

Chapter 5 describes how to reference objects in a program and how to display and modify program objects. Additionally, Chapter 5 discusses fully qualified and partially qualified symbol references.

Chapter 6 describes the four control constructs used by the PSCOPE command language: REPEAT, COUNT, IF, and DO.

Chapter 7 describes the command syntax necessary for defining, displaying, modifying, saving, and removing debug objects.

Chapter 8 describes what debug procedures are and how to define, use, display, save, and remove them.

Chapter 9 explains how to use code patches in a user's program.

Chapter 10 describes and explains the syntax for each of the utility commands and string functions available with PSCOPE.

Chapter 11 describes how to use breakpoints and tracepoints to control and monitor program execution.

Appendix A is a numerically ordered list of the PSCOPE error messages.

Appendix B provides the codes necessary to configure PSCOPE to run on non-Intel terminals.

Appendix C provides further information about using PSCOPE on the Series III development system.

Appendix D provides further information about using PSCOPE on the Series IV development system.

Appendix E contains the program upon which the examples in this manual are based.

Appendix F lists the lexical conventions which PSCOPE recognizes.

Appendix G lists the PSCOPE reserved keywords.

Appendix H lists PSCOPE commands and the pages on which they are discussed in this manual.

Appendix I provides further information about using PSCOPE on the iRMX™ operating system.

Related Publications

The following publications contain further information on the high-level languages PSCOPE supports:

Pascal-86 User's Guide, order number 121539

PL/M-86 User's Guide for 8086-Based Development Systems, order number 121636

FORTRAN-86 User's Guide, order number 121570

The following publications contain further information on the Series III:

Intellec ® Series III Microprocessor Development System Product Overview, order number 121575

Intellec ® Series III Microprocessor Development System Console Operating Instructions, order number 121609

Intellec® Series III Microprocessor Development System Programmer's Reference Manual, order number 121618

The following publications contain further information on the Series IV:

Intellec® Series IV Microcomupter Development System Overview, order number 121752

Intellec® Series IV Operating and Programming Guide, order number 121753

Intellec® Series IV ISIS-IV User's Guide, order number 121880

Notational Conventions

This manual adheres to the following conventions when describing PSCOPE command syntax.

Convention	**Meaning**
UPPERCASE WORDS	PSCOPE keywords. You must enter these words exactly as they appear, except that you can use either uppercase or lowercase.
italics	Grammar symbols for which you must substitute a value. These symbols are in italic face type.
a b c	You must enter the symbols a, b, and c in exactly the order specified.
[a b]	The items between the brackets are optional.
[a b]*	The items between the brackets are optional and may be repeated zero or more times.
a|b	Enter either the symbol a or the symbol b.
a ::= b c	Replace the symbol a with the symbol b followed by c.
punctuation	You must enter punctuation other than ellipses (...), brackets ([ ]), and braces ({ }) entered as shown. For example, you must enter all the punctuation shown in the following command: SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')
filename	A *filename* is a valid name for the part of a *pathname* that names a file.
pathname	A *pathname* uniquely specifies a file and consists of a *directory-name* and a *filename.*
directory-name	A *directory-name* is that portion of a *pathname* which locates the file by identifying the device and the directory containing the *filename.*
system-id	A *system-id* is a generic label placed on sample listings where an operating system-dependent name would actually be printed.

CNTL	CNTL denotes the terminal's control key. For example, CNTL-C means enter C while pressing the control key.
apostrophe	If your terminal has two apostrophe (or single quotes) symbols, determine which one PSCOPE accepts in command syntax.

CONTENTS

	PAGE
CHAPTER 1 INTRODUCTION	
What PSCOPE Is	1-1
What PSCOPE Can Do	1-1
The Program Development Process	1-1
Host System Execution Environment	1-2
Compiling or Assembling the Source Code	1-2
Compiling Under the ISIS Operating System	1-3
Invoking the Pascal Compiler	1-3
Invoking the PL/M-86 Compiler	1-3
Compiling Under the iRMX™-86 Operating System	1-3
Invoking the Pascal Compiler	1-3
Invoking the PL/M-86 Compiler	1-3
Linking the Object Code	1-3
Linking Under the ISIS Operating System	1-4
Invoking the Linker for a Pascal Program	1-4
Invoking the Linker for a PL/M-86 Program	1-4
Linking Under the iRMX™-86 Operating System	1-4
Invoking the Linker for a Pascal Program	1-5
Invoking the Linker for a PL/M-86 Program	1-5
Major Functions	1-5
Introductory Sample Session	1-6
A Debug Session with the Sample Pascal Program	1-6
A Debug Session with the Sample PL/M Program	1-14
CHAPTER 2 USING THE DEBUGGER	
Invoking the Debugger	2-1
The CRT|NOCRT Control	2-1
The MACRO|NOMACRO Control	2-2
The NOSUBMIT|SUBMIT Control	2-3
The VSTBUFFER Control	2-4
Invocation Error Messages	2-4
Using the Debugger	2-5
Terminating a Session	2-6
Command Entry	2-6
Line-editing Keys	2-6
Syntax Errors	2-7
The Internal Screen-Oriented Editor	2-7
Entering the Internal Editor	2-8
Exiting the Internal Editor	2-8
Internal Editor Display	2-9
Internal Editor Commands	2-9
Menu Commands	2-11
BLOCK Command (B)	2-11
DELETE Command (D)	2-12
GET Command (G)	2-12
INSERT Command (I)	2-12
QUIT Command (Q)	2-13
VIEW Command (V)	2-13
XCHANGE Command (X)	2-13
The VIEW Command	2-13
CHAPTER 3 COMMAND LANGUAGE AND EXPRESSIONS	
Tokens	3-1
Delimiters	3-1
Names	3-2
Names Format	3-2
Referencing Names	3-2
Line Numbers	3-3
Numeric Constants	3-3
Integers	3-3
Floating Point Numbers	3-4
Character String Constants	3-4
Operators	3-5
Comments	3-6
Types of Symbol Objects	3-7
Compiler/Assembler Type vs. PSCOPE Type Names	3-7
Expressions	3-8
Operands	3-8
Numeric Constants	3-9
String Constants	3-9
Program Symbol References	3-9
Machine Register References	3-9
Memory References with Explicit Typing	3-9
Line Number References	3-10
Debug Variable References	3-10
Debug Procedure Calls and Parameter References	3-11
Operators	3-11
Type Conversions	3-12
Type Conversions for Expressions	3-13
Type Conversions for Assignments	3-13
CHAPTER 4 LOADING AND EXECUTING PROGRAMS	
The LOAD Command	4-1
The GO Command	4-2
The LSTEP and PSTEP Commands	4-3
The ISTEP Command	4-4
CHAPTER 5 EXAMINING AND MODIFYING PROGRAM SYMBOLS	
Program Symbol References	5-1
Current Name Scope	5-1
Fully Qualified References	5-1
Partially Qualified References	5-3

PAGE
Display Program Symbol 5-3
Change Program Symbol 5-5
Change 8086/8088 Flags 5-5
Change 8086/8088 Registers 5-7
The REGS Command 5-8
Change 8087 Registers 5-8
Read and Write I/O Ports 5-9
Change Name Scope 5-10
Active Function 5-11
Display Memory 5-11
Modify Memory 5-12
The Single Line Assembler/Disassembler 5-13
The Disassembler 5-14
The Single Line Assembler (SLA) 5-15
RETFAR 5-17

CHAPTER 6
CONTROL CONSTRUCTS
The REPEAT and COUNT Constructs 6-1
The IF Construct 6-2
The DO Construct 6-3

CHAPTER 7
DEBUG OBJECT MANIPULATION COMMANDS
Debug Objects 7-1
Memory Type Debug Objects 7-2
Debug Type Debug Objects 7-2
The DEFINE Command 7-2
The DISPLAY Command 7-3
The MODIFY Command 7-5
The REMOVE Command 7-5
The PUT and APPEND Commands 7-6

CHAPTER 8
DEBUG PROCEDURES
DEFINE Debug Procedure 8-1
Debug Procedure Calls 8-2
Debug Procedure Return 8-2
Accessing Debug Procedure Parameters 8-3

CHAPTER 9
CODE PATCHES
Defining a Patch 9-1
Displaying a Patch 9-2
Removing a Patch 9-3

CHAPTER 10
UTILITY COMMANDS AND STRING FUNCTIONS
The EXIT Command 10-1
The DIRectory Command 10-2
The CALLSTACK Command 10-4
The HELP Command 10-5
The LIST and NOLIST Commands 10-6
The INCLUDE Command 10-6
The EVAL Command 10-7
The BASE Command 10-9

PAGE
INPUTMODE 10-10
The WRITE Command 10-12
The STRING Functions
(SUBSTR, CONCAT, STRLEN, and CI) 10-13
The SELECTOR$OF and OFFSET$OF
Functions 10-15

CHAPTER 11
ADVANCED EXECUTION AND TRACE COMMANDS
Breaking and Tracing 11-1
Break Registers (BRKREG) 11-1
Trace Registers (TRCREG) 11-3
The GO Command 11-4
Exception Trapping 11-6

APPENDIX A
ERROR MESSAGES
Classes of Errors A-1
Help A-1
Error Messages A-1

APPENDIX B
CONFIGURING PSCOPE FOR NON-INTEL TERMINALS
Configuration Commands B-1
Tested Configurations B-4
Adds Regent Model 200 .... B-5
Adds Veiwpoint 3A Plus B-6
Beehive Mini-Bee B-7
DEC VT52 .... B-8
DEC VT100 .... B-9
Hazeltine 1420 .... B-10
Hazeltine 1510 .... B-11
Intel Series-IIIE B-13
Lear Siegler ADM-3A B-14
Televideo 910 Plus B-15
Televideo 925 and 950 .... B-16
Zentec B-17

APPENDIX C
ADDITIONAL INFORMATION FOR SERIES III USERS
Operation of the Series III C-1
Program Development Process C-1
Hardware and Software Required C-2
User Programs Supported C-3
System Resources Used C-3
Memory C-3
File Requirements C-3
Other Resources Required C-4
Invocation Line C-4

APPENDIX D
ADDITIONAL INFORMATION FOR SERIES IV USERS
Operation of the Series IV D-1
Program Development Process D-1

PAGE

Hardware and Software Required D-2
User Programs Supported D-3
  System Resources Used D-3
  Memory D-3
  File Requirements D-3
  Other Resources Required D-4
Invocation Line D-4

APPENDIX E
SAMPLE PROGRAM LISTING

APPENDIX F
PSCOPE GRAMMAR
PSCOPE Grammar F-1

APPENDIX G
RESERVED KEYWORDS
PSCOPE Keywords G-1
PSCOPE Operators and Delimiters G-2

APPENDIX H
PSCOPE COMMAND INDEX

APPENDIX I
PSCOPE UNDER THE iRMX™ OPERATING SYSTEM
Linking I-1
Invocation I-1
Multitasking Support I-2
8087 Support I-2
Other Resources Required I-2

INDEX

TABLES

TABLE	TITLE	PAGE
3-1	Special Character Delimiters	3-1
3-2	Names Format	3-2
3-3	Elements of Integer Constants	3-3
3-4	Special Character Operators	3-5
3-5	Standard Symbol Object Types	3-6
3-6	Compiler/Assembler Type vs. PSCOPE Type Names	3-7
3-7	Precedence of Operators (Highest to Lowest)	3-12
5-1	Default Display Formats	5-4
B-1	Configuration Commands	B-2
B-2	ADDS Regent Model 200 Configuration	B-5
B-3	ADDS Viewpoint 3A Plus Configuration	B-6
B-4	Beehive Mini-Bee Configuration	B-7
B-5	DEC VT52 Configuration	B-8
B-6	DEC VT100 Configuration	B-9
B-7	Hazeltine 1420 Configuration	B-10
B-8	Hazeltine 1510 Configuration (Tilde Lead-in)	B-11
B-9	Hazeltine 1510 Configuration (ESC Lead-in)	B-12
B-10	Intel Series-IIIE Configuration	B-13
B-11	Lear Siegler ADM-3A Configuration	B-14
B-12	Televideo 910 Plus Configuration	B-15
B-13	Televideo 925 and 950 Configuration	B-16
B-14	Zentec Configuration	B-17

FIGURES

FIGURE	TITLE	PAGE
1-1	Generalized Program Development Process	1-2
1-2	Listing of Pager in Pascal-86	1-7
1-3	Listing of Pager in PL/M-86	1-17
2-1	Editor Display	2-9
C-1	Series III Program Development Process	C-2
D-1	Series IV Program Development Process	D-2

CHAPTER 1 INTRODUCTION

This chapter introduces the high-level language debugger called PSCOPE-86. It contains an overview of PSCOPE-86's capabilities and describes the preparation of a user program.

What PSCOPE Is

PSCOPE-86 is an interactive, symbolic debugger for object modules produced by the following compilers:

Pascal-86	(version 2.0 or later)
PL/M-86	(version 2.0 or later)
FORTRAN-86	(version 2.0 or later)
ASM-86	(version 2.0 or later)

Both PSCOPE and the program being debugged reside in the microcomputer system's memory (which is expandable to 1M byte). PSCOPE runs under the ISIS operating system, the iNDX operating system, and the iRMX^{TM}-86 operating system (release 5 or later).

What PSCOPE Can Do

With PSCOPE, you can examine and modify a program's execution to find software logic errors. With PSCOPE commands, you can do the following:

- Set breakpoints.
- Single-step through high-level language statements, assembly language instructions, functions, or procedures.
- Patch user code at either high-level language or assembly language level.
- Display and modify program memory and 8086/8088 registers. When PSCOPE-86 runs under the iRMX-86 operating system, you can also display and modify 8087 registers.
- Create and edit debug procedures, save these debug procedures, and recall them in future debug sessions.

The Program Development Process

PSCOPE is part of your microcomputer system's development software. Figure 1-1 shows how PSCOPE fits into a program development process.

Typical program development consists of the following steps:

1. Write the source code with the host system editor.
2. Compile the source code. This results in relocatable object code.
3. Correct any compile-time or assemble-time errors. Recompile or reassemble.

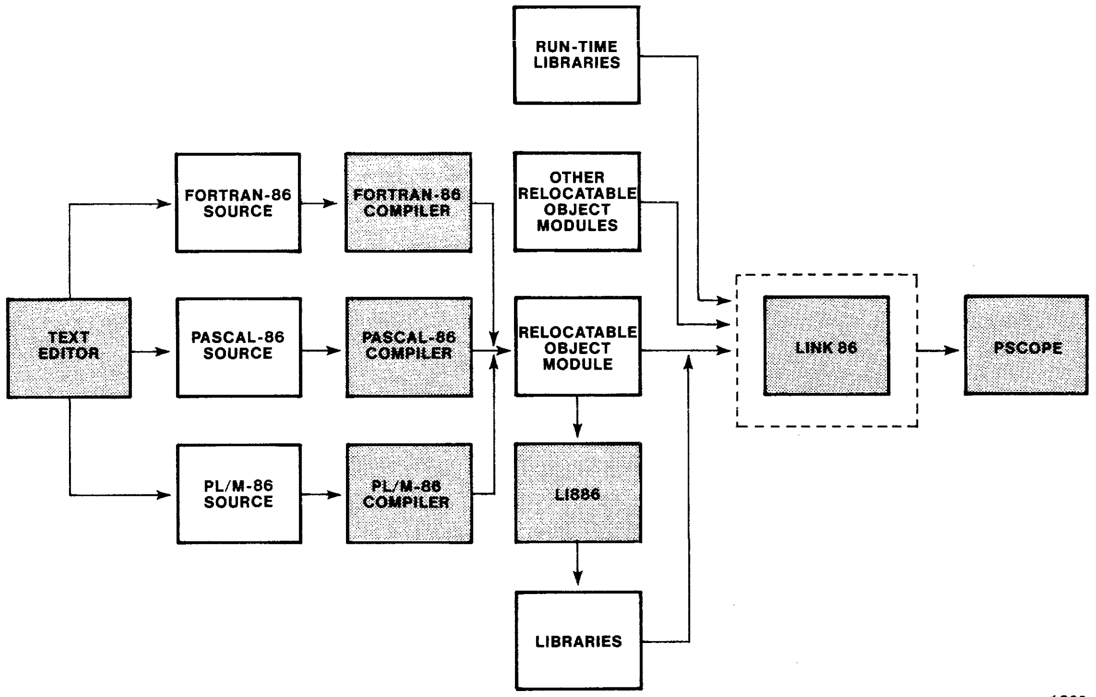

Figure 1-1 Generalized Program Development Process

4. Link the object code file to selected library files with LINK86, using the BIND option. This creates a loadable version of the program.

5. Execute, test, and debug the loadable object file using PSCOPE.

Host System Execution Environment

Appendixes B, C, D, and I contain information about the host system's execution environment, including related manuals, required system hardware and software, maximum user program size, and host system resources used by the debugger (such as memory and open files).

Compiling or Assembling the Source Code

When compiling or assembling the source code, include the following controls:

1. The DEBUG control. This control instructs the compiler or assembler to place symbol information in the object file.

2. The TYPE control. This control instructs the compiler or assembler to place type information in the object file. Symbol and type information is useful when debugging a user program with PSCOPE.

3. With a compiler, the OPTIMIZE(0) control. This control instructs the compiler to turn off object code optimization.

Compiling Under the ISIS Operating System

Assume that the source code file is called *pager.src* and you are running on an Intellec® Series III development system under the ISIS operating system.

Invoking the Pascal Compiler

Assume that the Pascal compiler is on disk drive one and that the source code is on disk drive zero. The following command will invoke the Pascal compiler.

```
-run:f1 :pasc86 pager.src debug type optimize(0)
```

Note that TYPE is a default option for the Pascal-86 compiler.

Invoking the PL/M-86 Compiler

Assume that the PL/M-86 compiler is on disk drive one and that the source code is on disk drive zero. The following command will invoke the PL/M-86 compiler.

```
-run:f1 :plm86 pager.src debug type optimize(0)
```

Note that TYPE is a default option for the PL/M-86 compiler.

Compiling Under the iRMX™-86 Operating System

Assume that the source code file is called *pager.src* and that you are running on an 86/3xx system under the iRMX-86 operating system.

Choose either the COMPACT or the LARGE option. The iRMX-86 operating system does not support the SMALL or MEDIUM option.

Invoking the Pascal Compiler

Assume that the pathname of the Pascal compiler is /lang/pasc86 and that the pathname of the source code is /user/world/prog/pager.src. The following command will invoke the Pascal compiler.

```
-/lang/pasc86 /user/world/prog/pager.src compact debug type
optimize(0)
```

Invoking the PL/M-86 Compiler

Assume that the pathname of the PL/M-86 compiler is /lang/plm86 and that the pathname of the source code is /user/world/prog/pager.src. Here's how to invoke the PL/M-86 compiler.

```
-/lang/plm86 /user/world/prog/pager.src. compact debug type
optimize(0)
```

Linking the Object Code

PSCOPE does not support overlays. Do not link overlay files into the final debug load module.

Be sure to use the library files supplied with the operating system. The library files supplied with the ISIS operating system have the same names and perform the same functions as those supplied with the iRMX-86 operating system, but they differ internally.

PSCOPE requires that the user program be in load-time-locatable (LTL) code. This means that code and data addresses are assigned by the system loader. Use the BIND control when invoking LINK86.

Linking Under the ISIS Operating System

Assume that the object code file is called *pager.obj* and you are running on an Intellec Series III development system under the ISIS operating system. Assume that LINK86 and the library files are on disk drive one and that *pager.obj* is on disk drive zero.

When the program you plan to debug with PSCOPE-86 does real arithmetic, you have the choice of using the 8087 chip or the 8087 software emulator.

Invoking the Linker for a Pascal Program

The following command will invoke the linker for a Pascal program.

```
run-:f1:link86 pager.obj,   :f1:p86rn0.lib, &
                            :f1:p86rn1.lib, &
                            :f1:p86rn2.lib, &
                            :f1:p86rn3.lib, &
                            :f1:87null.lib, &
                            :f1:large.lib to pager.86 bind
```

This example assumes that your microcomputer system does not contain the 8087 chip. Because the Pascal example in this chapter does not do real arithmetic, e8087.lib and e8087 are not required; 87null.lib is required to resolve external references in p86rn2.lib and p86rn3.lib.

If your microcomputer system contains the 8087 chip, 8087.lib takes the place of e8087.lib and e8087.

Invoking the Linker for a PL/M-86 Program

Here's how to invoke the linker for a PL/M-86 program.

```
-run:f1:link86 pager.obj,   :f1:plm86.lib, &
                            :f1:compac.lib to pager.86 bind
```

This example assumes that you chose the COMPAC option when compiling the PL/M-86 program. The default is SMALL.

Linking Under the iRMX™-86 Operating System

Assume that you are running on an 86/3xx system under the iRMX-86 operating system.

When running under the iRMX-86 operating system (release 5 or later) and invoking LINK86, include the following two options during the final link.

1. **SEGSIZE(STACK(+2048))**	This option increases the stack size by 2048 bytes.
2. **MEMPOOL(+25000,0FF000H)**	LINK86 reserves 25000 bytes for dynamic memory allocation.

When the program you plan to debug with PSCOPE-86 does real arithmetic, your microcomputer system must contain the 8087 chip. The iRMX-86 operating system does not support the 8087 software emulator.

Invoking the Linker for a Pascal Program

Assume that the pathname of link86 is /lang/link86 and that the pathname of the object file is /user/world/prog/pager.obj. The following command will invoke the linker for a Pascal program.

```
-/lang/link86  /user/world/prog/pager.obj, &
               /lang/p86rn0.lib, &
               /lang/p86rn1.lib, &
               /lang/p86rn2.lib, &
               /lang/p86rn3.lib, &
               /lang/87null.lib, &
               /lang/large.lib to /user/world/prog/pager.86 &
                segsize(stack(+2048)) mempool(+25000,0FF000H)
                bind
```

If your program does real arithmetic, your microcomputer system must contain the 8087 chip, and you must link your program with 8087.lib. Although the Pascal example in this chapter does not do real arithmetic, 87null.lib is required to resolve external references in p86rn2.lib and p86rn3.lib.

Invoking the Linker for a PL/M-86 Program

Assume that the pathname of link86 is /lang/link86 and that the pathname of the object file is /user/world/prog/pager.obj. The following command will invoke the linker for a PL/M-86 program.

```
-/lang/link86  /user/world/prog/pager.obj, &
               /lang/plm86.lib, &
               /lang/compac.lib to /user/world/prog/pager.86 &
                segsize(stack(+2048)) mempool(+25000,0FF000H)
                bind
```

Major Functions

The following list briefly describes PSCOPE's major functions and references the chapters containing more detailed information.

- Internal editor (Chapter 2). The internal, screen-oriented editor lets you edit commands, debug procedures, and patches from the keyboard.
- Single-stepping (Chapter 4). PSCOPE allows single-stepping through assembly language instructions, high-level language statements, and procedures.
- Control blocks (Chapter 6). You can use PSCOPE command language conditional and repetitive control constructs to build up blocks of debugger commands.

- Debug objects (Chapter 7). You can define and manipulate various types of debug objects, such as debug variables, debug procedures, and LITERALLYs (a form of command macro).
- Debug procedures (Chapter 8). You can define and edit debug procedures. Debug procedures consist of PSCOPE-86 commands.
- Code patches (Chapter 9). PSCOPE lets you add and delete code at the statement or instruction level without permanently changing your program. Patches made in PSCOPE are not permanent but they can be saved. Your program will not run with those patches outside of PSCOPE. If you load another program, PSCOPE deletes any previously defined patches.
- Single-line assembler and disassembler (Chapter 9). With these features you can modify and display assembly language instructions.
- Utility commands (Chapter 10). With utility commands, you can perform such functions as obtaining on-line help, recording all or part of a debug session in a log file, and executing command files.
- Breakpoints and tracepoints (Chapter 11). By setting breakpoints and tracepoints, you can control and monitor debugging.

Introductory Sample Session

This section contains two sample programs, one in Pascal-86 and one in PL/M-86. Both programs are called *pager.src*, and both perform the same function.

The program reads text from a file called *txtin*, formats the text, and writes it to a file called *txtout*. The program inserts 10 spaces at the beginning of each line, thus creating a left margin, inserts page breaks, writes a heading for each page, and numbers the pages. The program may also double-space.

A Debug Session With the Sample Pascal Program

Figure 1-2 is the list file for the sample Pascal program.

```
SERIES-III Pascal-86, V3.0
Source File: PAGER.SRC
Object File: PAGER.OBJ
Controls Specified: XREF, CODE, DEBUG, OPTIMIZE(0), TYPE.

STMT  LINE  NESTING      SOURCE TEXT: PAGER.SRC
  1     1    0  0        program pager(input,output);
  2     2    0  0        const   blank=' ';
  3     3    0  0        var     textin,textout          :text;
  4     4    0  0                ch                      :char;
  5     5    0  0                leftmargin,i,linenumber :integer;
  6     6    0  0                linesend,pagenumber     :integer;
  7     7    0  0                double                  :boolean;
  8     8    0  0        procedure init(var leftmargin,linesend:integer;
  8     9    1  0                    var double:boolean;
                                     var textin:text);
```

(figure continues)

```
   9   12   1   0       BEGIN      (*init*)
   9   13   1   1       leftmargin:=10;
  10   14   1   1       linesend:=50;
  11   15   1   1       double:=false;
  12   16   1   1       writeln('leftmargin   = ',leftmargin:2);
  13   17   1   1       writeln('lines/page   = ',linesend:2);
  14   18   1   1       writeln('double       = ',double);
  15   19   1   1       writeln
                        END        (*init*);

  16   22   0   0       $eject


SERIES-III Pascal-86,

STMT  LINE  NESTING     SOURCE TEXT: PAGER.SRC

   16   24   0   1       reset(textin,'txtin');
   17   25   0   1       rewrite(textout,'txtout');
   18   26   0   1       pagenumber:=1;
   19   27   0   1       linenumber:=1;

   20   29   0   1       init(leftmargin,linesend,double,textin);

   21   31   0   1       while eof(textin)=false do
   22   32   0   1         begin
   22   33   0   2         writeln(textout);
   23   34   0   2         writeln(textout);
   24   35   0   2         writeln(textout,'                                   ',
                                                                  Intel Corporation
 ',pagenumber:4);
   25   37   0   2         writeln(textout);

   26   39   0   2         repeat
   26   40   0   3           for i:=leftmargin down to 1 do
   27   41   0   3             write(textout,blank);
   28   42   0   3           while eoln(textin)=false do
   29   43   0   3             BEGIN
   29   44   0   4             read(textin,ch);
   30   45   0   4             write(textout,ch)
                               END;
   32   47   0   3           if double=true then
   33   48   0   3             BEGIN
   33   49   0   4             writeln(textout);
   34   50   0   4             writeln(textout);
   35   51   0   4             linenumber:=linenumber+2
                               END
   36   53   0   3           else
   37   54   0   3             BEGIN
   37   55   0   4             writeln(textout);
   38   56   0   4             linenumber:=linenumber+1
                               END;
   40   58   0   3           readln(textin)
                           until (linenumber=linesend) or (eof(textin)=true);

   42   61   0   2         page(textout);
   43   62   0   2         writeln('page = ',pagenumber:4);
   44   63   0   2         pagenumber:=pagenumber+1;
   45   64   0   2         linenumber:=1
                           end;
   47   66   0   1       writeln;
   48   67   0   1       writeln('end of file on textin encountered')
                         END.       (*main*)
```

Figure 1-2 Listing of Pager in Pascal-86

The sample Pascal program has a bug in it. Statement #41 should read as follows:

```
until (linenumber>linesend) or (eof(textin)=true);
```

rather than the following:

```
until (linenumber=linesend) or (eof(textin)=true);
```

The program works with the equal sign if you choose the single-space option. The loop does not terminate if you choose the double-space option and the program variable *linesend* is an even number.

What follows is a sample debug session. The boxed text at the top explains the action of the PSCOPE dialogue following the boxed text.

Invoke PSCOPE. Set up a list file called *pager.log*. Load the program *pager.86*. Define a debug variable called *begin* and set it equal to the current CS:IP, represented by a dollar sign ($).

Define a debug procedure called *again*. This sets $ and namescope to *begin*. This debug procedure is useful after you have executed your program once in PSCOPE and wish to execute again.

Define a debug procedure called *fixlinenumber*. This debug procedure checks if the program variable *linenumber* is greater than 50 and if so, sets it to 50.

Define a break register called *line41*. If you execute your program with this break register, execution stops at the end of the loop just before the terminating check. If you now call the debug procedure *fixlinenumber*, the loop terminates as intended.

Save the defined debug objects. Write them to the disk in a file called *pager.mac*. You can call up this file and use these debug objects in later debug session.

```
*list pager.log

*load pager.86
*
*define pointer begin=$
*
*define proc again=do
.*$=begin
.*namescope=$
.*end
*
*define proc fixlinenumber = do
.*if linenumber >50 then
..*linenumber=50
..*end
.*end
*
*define brkreg line41=#41
*
*put pager.mac debug
```

List the currently defined debug objects with the DIR command. Also list the program variables and procedures.

Display the current value of $, the starting memory location.

Begin program execution. Break just before the execution of statement #41.

Notice that the program itself writes initial variables to the screen. Such output is not included in a list file. Notice that the program variable *double* is false. The program only fails when *double* is true.

```
*dir debug
LINE41  . .  brkreg
BEGIN   . .  pointer
FIXLNUMBER proc
AGAIN   . .  proc
*
*dir
DIR of :PAGER
PQ_OUTPUT ............ TEXT (file)
PQ_INPUT .............. TEXT (file)
TEXTOUT ............... TEXT (file)
TEXTIN ................ TEXT (file)
CH .................... char
LINENUMBER ............ integer
I ..................... integer
LEFTMARGIN ............ integer
PAGENUMBER ........... integer
LINESEND .............. integer
DOUBLE ................ boolean
INIT .................. procedure
     LEFTMARGIN ............ integer
     LINESEND ............... integer
     DOUBLE ................. boolean
     TEXTIN ................. TEXT (file)
*
*$
1C88H:0159H
*go til #41
leftmargin = 10
lines/page = 50
double = F

[Break at #41]
```

Display the current value of the program variable *linenumber*. Change it to three.

Display the address of *linenumber*. The period (.) signifies "the address of." Identify what symbolic variable is stored at that address with the EVAL command. The symbol *ds* represents the data segment register.

Display the data segment register. Because the default radix is decimal, the resulting number does not correspond with the selector displayed when .linenumber was entered. Change the current radix to hexadecimal and redisplay the data segment register. Then return the current radix to decimal.

Determine what program variables are integers. Apply the INTEGER memory template command to .linenumber. Notice that you get both the address of *linenumber* and the value stored there.

```
*linenumber
+2
*linenumber=3
*.linenumber
1CC3H:0024H
*
*eval ds:0024h symbol
:PAGER.LINENUMBER
*
*ds
7363
*base=10h
*ds
1CC3
*base=10t
*
*dir integer
DIR of :PAGER
LINENUMBER
I
LEFTMARGIN
PAGENUMBER
LINESEND
INIT.LEFTMARGIN
INIT.LINESEND
*
*integer .linenumber
1CC3H:0024H      +3
```

Display the value of *double*. Apply the BOOLEAN memory template command to .double. Set *double* to true and again apply the BOOLEAN memory template command.

Resume execution and break again at statement #41. Display the program variables *linenumber* and *double*. Display the break register *line41*. You intended this break register to call the debug procedure *fixlinenumber*, but you neglected to include that call.

```
*double
FALSE
*boolean .double
1CC3H:002EH FALSE
*double=true
*boolean .double
1CC3H:002EH TRUE
*
*go til #41
[Break at #41]
*linenumber
+5
*double
TRUE
*
*brkreg line41
define brkreg LINE41 =#41
```

Edit the break register *line41*. Press the i key (for insert). The menu on the last line of the screen changes to [insert]. Move the cursor to the end of the line and type *call fixlinenumber*. Press the ESC key. The main menu returns.

Exit the editor by pressing the q key (for quit). The menu changes to the following:

```
Abort   Execute
```

Press the e key (for execute). The PSCOPE prompt returns, and PSCOPE redefines the break register *line41*.

```
*edit line41
define brkreg LINE41 =#41

----
Block    Delete    Get    Insert     Quit    View    Xchange

define brkreg LINE41 =#41

----
[insert]

define brkreg LINE41 =#41 call fixlinenumber

----
Abort    Execute


*define brkreg LINE41 =#41 call fixlinenumber
```

Display the break register *line41*. Begin execution with this break register. The warning message indicates that your debug procedure has problems. A debug procedure called by a break register must return a boolean value. True means break; false means execute the procedure, but resume the program execution.

Obtain on-line help with the HELP command. The asterisk in brackets after the warning message indicates that additional on-line information is available.

Edit the debug procedure. Change it so that it displays the current value of *linenumber* and breaks when *linenumber* is greater than 50. This example skips the editing session because it is similar to that with the break register *line41*.

```
*brkreg line41
define brkreg LINE41 =#41 call fixlinenumber
*
*go using line41
WARNING #514: Invalid return type from PROC called at breakpoint. [*]
[Break at #41]
*help e514
====
E514
====
```

The debugger procedure called at the break or trace point returned a value with an invalid type, or had no return value. The return value must be a BYTE, WORD, DWORD, BOOLEAN, or INTEGER (including LONG/SHORT). A return value of TRUE is manufactured, causing the associated break or trace to be done.

```
*
*edit fixlinenumber
*define proc FIXLNUMBER = do
.*write 'linenumber= ',linenumber
.*if linenumber >50 then
..*linenumber=50
..*write 'linenumber= ',linenumber
..*return true
..*else return false
..*end
.*end
```

Resume execution. The GO command assumes the last go specification. Typing GO is the same as typing GO USING line41. The debug procedure *fixlinenumber* displays the line numbers as they increment. Notice *linenumber* go from 51 to 50. Then the break occurs.

Step through three high-level language statements. The program writes the program variable *pagenumber* to the screen. Because the program itself writes to the screen, this output does not appear in the list file.

Step through one more statement. Display *pagenumber* and notice that it has incremented. Step through another statement and notice that the program has returned to the beginning of the loop, ready to take care of the second page.

```
*go
linenumber=   +9
linenumber=   +11
linenumber=   +13
linenumber=   +15
linenumber=   +17
linenumber=   +19
linenumber=   +21
linenumber=   +23
linenumber=   +25
linenumber=   +27
linenumber=   +29
linenumber=   +31
linenumber=   +33
linenumber=   +35
linenumber=   +37
linenumber=   +39
linenumber=   +41
linenumber=   +43
linenumber=   +45
linenumber=   +47
linenumber=   +49
linenumber=   +51
linenumber=   +50
[Break at #41]
```

```
*
*lstep
[Step at :PAGER#42]
*lstep
[Step at :PAGER#43]
*lstep
page = 1
[Step at :PAGER#44]
*lstep
[Step at :PAGER#45]
*pagenumber
+2
*lstep
[Step at :PAGER#21]
```

Resume execution using the break register *line41*. Step through some additional statements and display *pagenumber* again.

```
*go using line41
linenumber=   +3
linenumber=   +5
linenumber=   +7
linenumber=   +9
linenumber=   +11
linenumber=   +13
linenumber=   +15
linenumber=   +17
linenumber=   +19
linenumber=   +21
linenumber=   +23
linenumber=   +25
linenumber=   +27
linenumber=   +29
linenumber=   +31
linenumber=   +33
linenumber=   +35
linenumber=   +37
linenumber=   +39
linenumber=   +41
linenumber=   +43
linenumber=   +45
linenumber=   +47
linenumber=   +49
linenumber=   +51
linenumber=   +50
[Break at #41]
*
*lstep;lstep;lstep
[Step at :PAGER#42]
[Step at :PAGER#43]
page = 2
[Step at :PAGER#44]
*lstep
[Step at :PAGER#45]
*integer.pagenumber
1CC3H:002AH     +3
```

Resume execution using the break register *line41*. The program encounters an end-of-file before *linenumber* exceeds 50.

Close the list file. Exit PSCOPE.

```
*go using line41
linenumber=  +3
linenumber=  +5
linenumber=  +7
linenumber=  +9
linenumber=  +11
linenumber=  +13
linenumber=  +15
linenumber=  +17
linenumber=  +19
linenumber=  +21
page = 3

end of file on textin encountered

EXCEPTION: Program call to DQ$Exit
[Stop at location 2178H:0030H]
*nolist
*exit
PSCOPE terminated
-
```

A Debug Session with the Sample PL/M Program

Figure 1-3 is the list file for the PL/M-86 program.

```
SERIES-III PL/M-86 V2.3 COMPILATION OF MODULE PAGER
OBJECT MODULE PLACED IN PAGER.OBJ
COMPILER INVOKED BY: :F1:PLM86.86 PAGER.SRC OPTIMIZE(0) COMPACT DEBUG
                          TYPE XREF CODE

  1       pager:
          do;

  2   1   declare connection          literally 'word';
  3   1   declare (conin,conout)      connection;
  4   1   declare (bytes$read,err)    word;
  5   1   declare compl               word;
  6   1   declare boolean             literally 'byte';
  7   1   declare true                literally '0ffh';
  8   1   declare false               literally '00h';
  9   1   declare blank               byte data(32);
 10   1   declare lfcr(2)             byte data(10,13);
 11   1   declare ff                  byte data(12);
 12   1   declare (ch,i)              byte;
 13   1   declare      (textin,textout)
          connection;

 14   1   declare (leftmargin,linesend,linenumber,pagenumber) word;
 15   1   declare double              boolean;


          $eject
          /*The external procedure declarations follow.*/
```

(figure continues)

```
16  1   dq$attachprocedure (path$p,excep$p) connection external;
        :
17  2     declare path$p   pointer,
                  excep$p pointer;
18  2     end;

19  1   dq$detacprocedure (conn,excep$p) external;
        h:
20  2     declare conn      connection,
                  excep$p pointer;
21  2     end;

22  1   dq$creat procedure (path$p,excep$p) connection external;
        e:
23  2     declare path$p   pointer,
                  excep$p pointer;
24  2     end;

25  1   dq$open: procedure (conn,access,num$buf,excep$p) external;
26  2     declare conn      connection,
                  access   byte,
                  num$buf byte,
                  excep$p pointer;
27  2     end;

28  1   dq$close:procedure (conn,excep$p) external;
29  2     declare conn      connection,
                  excep$p pointer;
30  2     end;

31  1   dq$read:  procedure (conn,buf$p,count,excep$p) word external;
32  2     declare conn      connection,
                  buf$p     pointer,
                  count     word,
                  excep$p pointer;
33  2     end;

34  1   dq$write: procedure (conn,buf$p,count,excep$p) external;
35  2     declare conn      connection,
                  buf$p     pointer,
                  count     word,
                  excep$p pointer;
36  2     end;

37  1   dq$exit:   procedure (completion$code) external;
38  2     declare completion$code word;
39  2     end;

        $eject
        /*The local procedures follow.*/

40  1   numout:
        procedure (value,where);
41  2     declare  value       word;
42  2     declare  where       connection;
43  2     declare  digits(*)   byte data('0123456789');
44  2     declare  chars(5)    byte;
45  2     declare  i           byte;

46  2     do i=1 to 5;
47  3       chars(5-i)=digits(value mod 10);
48  3       value=value/10;
49  3     end;
50  2     i=0;
51  2     do while chars(i)='0' and i<4;
52  3       chars(i)=' ';
53  3       i=i+1;
54  3     end;
55  2     call dq$write(where,@chars,5,@err);
56  2     call dq$write(where,@lfcr,2,@err);
57  2   end numout;
```

(figure continues)

```
              $eject
              /*Now begins the executable code*/

  58    1     leftmargin=10;
  59    1     linesend=50;
  60    1     double=true;

              /*In the interest of space, initial variables are assigned in the code. A more useful program
              would read leftmargin, linesend, and double as input.*/


  61    1     pagenumber=1;
  62    1     linenumber=1;

  63    1     conout=dq$create(@(4,':CO:'),@err);
  64    1     conin =dq$attach(@(4,':CI:'),@err);

  65    1     call dq$open(conin,1,0,@err);
  66    1     call dq$open(conout,2,0,@err);

  67    1     call dq$write(conout,@('leftmargin= '),12,@err);
  68    1     call numout(leftmargin,conout);

  69    1     call dq$write(conout,@('lines/page= '),12,@err);
  70    1     call numout(linesend,conout);

  71    1     call dq$write(conout,@('double = '),12,@err);
  72    1     if double=true then
  73    1       do;
  74    2       call dq$write(conout,@(' true'),5,@err);
  75    2       call dq$write(conout,@lfcr,2,@err);
  76    2       end;
  77    1     else
                do;
  78    2       call dq$write(conout,@(' false'),6,@err);
  79    2       call dq$write(conout,@lfcr,2,@err);
  80    2       end;


              /*Attach the file txtin and create the file txtout; open them.*/

  81    1     textout=dq$create(@(6,'txtout'),@err);
  82    1     textin=dq$attach(@(5,'txtin'),@err);

  83    1     call dq$open(textin,1,0,@err);
  84    1     call dq$open(textout,2,2,@err);


              $eject
              /*Output to file textout.*/

  85    1     bytes$read=1;
  86    1     do while bytes$read< >0;

              /*Output the heading.*/

  87    2     do i=1 to 50;
  88    3       call dq$write(textout,@blank,1,@err);
  89    3     end;
  90    2     call dq$write(textout,@('Intel Corporation '),21,@err);
  91    2     call numout(pagenumber,textout);
  92    2     call dq$write(textout,@lfcr,2,@err);
  93    2     call dq$write(textout,@lfcr,2,@err);

              /*Output the page.*/

  94    2     do while linenumber< =linesend and bytes$read< >0;
  95    3       do i=1 to leftmargin;
  96    4           call dq$write(textout,@blank,1,@err);
  97    4       end;
```

(figure continues)

```
 98  3      bytes$read=dq$read(textin,@ch,1,@err);
 99  3      do while ch <> 13 and bytes$read <>0;
100  4          call dq$write(textout,@ch,1,@err);
101  4          bytes$read=dq$read(textin,@ch,1,@err);
102  4      end;
103  3      bytes$read=dq$read(textin,@ch,1,@err);

104  3      if double=true then
105  3      do;
106  4          call dq$write(textout,@lfcr,2,@err);
107  4          call dq$write(textout,@lfcr,2,@err);
108  4          linenumber=linenumber+2;
109  4      end;
110  3      else
            do;
111  4          call dq$write(textout,@lfcr,2,@err);
112  4          linenumber=linenumber+1;
113  4      end;
114  3    end;

115  2    call dq$write(textout,@ff,1,@err);
116  2    call dq$write(conout,@('page= '),6,@err);
117  2    call numout(pagenumber,conout);
118  2    pagenumber=pagenumber+1;
119  2    linenumber=1;
120  2    end;

            $eject
            /*Close and detach files.*/

121  1    call dq$close(conin,@err);
122  1    call dq$close(conout,@err);
123  1    call dq$close(textin,@err);
124  1    call dq$close(textout,@err);

125  1    call dq$detach(conin,@err);
126  1    call dq$detach(textin,@err);
127  1    call dq$detach(textout,@err);

128  1    halt;
129  1    call dq$exit(compl);
130  1    end;
```

Figure 1-3 Listing of Pager in PL/M-86

This program does not contain any obvious errors. In the interest of space, the code does not check the status returned after the file handling system calls. It is good programming practice to check this status and enter an error routine if the status is non-zero.

Invoke PSCOPE, load the program *pager.86.* Define a debug procedure called *lnumber.* This debug procedure is intended to be called by a trace register. It prints out the current line number and returns a value of true.

Define a trace register called *line115.* Statement #115 of the PL/M program writes a form feed at the end of a page in the file *txtout.* Printing out *linenumber* at this time displays its value when the loop is exited.

Display the current value of CS:IP.

```
*load pager.86
*
*define proc LNUMBER =do
.*write 'linenumber= ',linenumber
.*return true
.*end
*define trcreg LINE115 =#115 call lnumber
*$
1CC6H:0000H
```

Begin execution with the trace register *line115*. Exit PSCOPE.

```
*go using line115
leftmargin= 10
lines/page= 50
double= true
linenumber= 51
[At #115]
page= 1
linenumber= 51
[At #115]
page= 2
linenumber= 23
[At #115]
page= 3

EXCEPTION: Program call to DQ$Exit
[Stop at location 1CC6H:04AEH]
*exit
PSCOPE terminated
-
```

CHAPTER 2 USING THE DEBUGGER

This chapter describes how to use the debugger, including the following:

- Invoking the debugger using initialization and configuration controls.
- Entering commands from the keyboard.
- Editing command lines with the line editor or the internal editor.
- Using the menu commands.

Invoking the Debugger

Invoke the debugger by entering the following invocation line (notational conventions are defined in the Preface):

[RUN] [*directory-name*] PSCOPE [*controls*]*

Where:

directory-name	is the host system's file path name.
controls	is any of the following invocation controls. The first control listed in each pair is the default. The controls can be specified in any order.

CRT|NOCRT
MACRO|NOMACRO
NOSUBMIT|SUBMIT
VSTBUFFER (*decimal-number*)

Each control is described in the following sections.

The CRT|NOCRT Control

CRT specifies a file which defines CRT characteristics that describe the control sequences for communicating with the terminal. The form of this file is described in Appendix B.

If you do not enter either CRT or NOCRT, the debugger looks for the CRT file PSCOPE.CRT in the same directory from which the debugger was invoked. If PSCOPE does not find the file PSCOPE.CRT, the debugger assumes that the keyboard and CRT control sequences are the same as those on a standard Series III or Series IV development system.

If you specify CRT without a file name, the default file (PSCOPE.CRT) must exist; otherwise, PSCOPE displays an error message.

If you specify NOCRT, the debugger does not look for a CRT file. It assumes a Series III- or Series IV-based machine.

If you rename the debugger file and invoke it with the new name, the debugger looks for a CRT file with the new name.

Syntax

```
CRT [(filename)]
NOCRT
```

Abbreviation

CR|NOCR

Default

PSCOPE.CRT

Example

Following are some examples of the CRT|NOCRT control commands.

```
RUN PSCOPE
run pscope crt(1510t.crt)
RUN PSCOPE NOCRT
```

The MACRO|NOMACRO Control

MACRO specifies a file containing debugger commands to be executed during debugger initialization. You create MACRO files containing command definitions useful to a particular application, such as abbreviations or debugger procedures which will be used over several debug sessions.

If you do not enter either MACRO or NOMACRO, the debugger looks for the file PSCOPE.MAC in the same directory from which the debugger was invoked.

The PSCOPE.MAC file can contain any number of debugger commands to be executed on initialization. For example, the MAC file can automatically define abbreviations using the LITERALLY command (discussed in Chapter 7).

If you specify MACRO without a file name, the default macro file must already exist.

If you specify NOMACRO, the debugger does not look for a macro file.

If you rename the debugger and invoke it with the new name, the debugger automatically looks for a MAC file with the new name.

A macro file is not required.

Syntax

```
MACRO [(filename)]
NOMACRO
```

Abbreviation

MR | NOMR

Default

PSCOPE.MAC

Example

Here are some sample MACRO | NOMACRO control commands.

```
run pscope macro(procs.mac)
RUN PSCOPE NOMR
```

The NOSUBMIT | SUBMIT Control

SUBMIT indicates that PSCOPE is to operate inside of a SUBMIT file. If you specify SUBMIT, you must use the standard system line editor rather than PSCOPE's extended line editor. Using the standard system line editor ensures that SUBMIT file commands are echoed properly to the system terminal.

Syntax

```
NOSUBMIT
SUBMIT
```

Abbreviation

NOSM | SM

Default

NOSUBMIT

Example

The following illustrates how to use the NOSUBMIT | SUBMIT control.

```
run pscope sm
RUN PSCOPE NOSM
```

The VSTBUFFER Control

PSCOPE employs a virtual symbol table. Only a portion of the user-program symbol table need reside in physical memory at any one time. The entire symbol table resides on disk.

VSTBUFFER specifies the amount of physical memory to be used for the virtual symbol table. Replace *decimal-number* with the number of kilobytes that you want reserved for the virtual symbol table buffer. The parameter *decimal-number* must be a number in the range 5 through 61.

The larger the virtual symbol table buffer is, the less time PSCOPE must spend manipulating the symbol table. On the other hand, a small virtual symbol table buffer frees up more memory for use by the user program and debug objects.

Syntax

```
VSTBUFFER(decimal-number)
```

Where:

decimal-number	is a number in the range 5 through 61, specifying the amount of physical memory in kilobytes to be used for the virtual symbol table.

Abbreviation

```
VSTB(decimal-number)
```

Default

VSTB(5)

Invocation Error Messages

You can make three types of errors when entering the invocation line:

- An unrecognized control
- A control missing a required parameter
- A control with an invalid parameter

When an invocation error occurs, PSCOPE displays an error message, followed by the operating system prompt. You can then enter a corrected invocation line.

PSCOPE displays the following error message when you enter an unrecognized control character:

```
UNKNOWN CONTROL: control
```

PSCOPE displays the following error message when you do not include the required parameter with the invocation line:

```
PSCOPE OPTION ERROR
     OPTION:  control
     ERROR:  message
PSCOPE terminated
```

PSCOPE displays the following error message when you enter an invalid parameter:

```
PSCOPE I/O ERROR
     FILE:file-type
     NAME:filename
     ERROR:message
PSCOPE terminated
```

Where:

file-type	is CRT or MAC.
filename	is the name of your file.
message	is the error message that identifies the problem.

Using the Debugger

After you correctly enter the invocation line, the debugger clears the screen and displays the following sign-on message:

system-id PSCOPE-86, V*x.y*

The *system-id* identifies the host system. On the Series III, the host system-id is SERIES III.

In V*x.y*, *x* is the debugger version number, and *y* is the change number. After signing on, the debugger prompts for commands with an asterisk (*). User software can then be loaded, executed, tested, and debugged by entering the commands described in subsequent chapters. After completing an operation in response to a command, the debugger prompts for new input.

You can perform any of the following operations when the debugger prompt is displayed:

- Enter a command from the keyboard.
- Enter commands from an external file.
- Re-execute the last command (CNTL-E).
- Enter the internal editor to create debug objects or edit commands.
- Suspend or cancel debugger terminal output.

While the debugger is executing, you can interrupt operation as follows:

- Suspend terminal output by entering CNTL-S and resume terminal output by entering CNTL-Q.

- Cancel the command in progress by entering CNTL-C.

You can create a file containing debugger commands, then use the INCLUDE command described in Chapter 10, to enter the commands from that file.

You can record the debugging session by using the LIST command (described in Chapter 10). The LIST command sends all debugger terminal output to the specified file, either on disk or hard copy. The file includes PSCOPE prompts, user input lines, PSCOPE output, and error messages. It does not, however, contain output from the program being debugged.

Terminating a Session

Enter the following EXIT command to exit the debugger:

```
EXIT
```

The debugger responds with the following message:

```
PSCOPE terminated
```

The debugger closes all open files and returns to the operating system.

Command Entry

The debugger prompt (*) indicates that the system is in command-entry mode. In this mode, the debugger places characters entered at the terminal in a command buffer until the end of a complete command coincides with the end of a command line. At this point, the debugger executes all commands in the buffer in the order in which they were read. (Legal command line length is virtually unlimited, depending on the amount of workspace available.)

You can continue commands which will not fit on one line on subsequent lines. The continuation flag, an ampersand (&) at the end of the line to be continued, is optional because the debugger can recognize the end of a completed command in most cases. (PSCOPE issues a double asterisk (**) prompt if it needs more command input.) The exception is when a command has an optional parameter that is placed on the next line. In this case, the debugger executes the partial command (since it is complete) unless you include the continuation flag.

Continuation flags let you specify a multi-line sequence of commands before they are executed (without using the DO construct explained in Chapter 6).

You must separate commands with a semicolon when you specify more than one command on a line.

Line-Editing Keys

You can edit command input in line-oriented mode using the following line editing functions:

RUBOUT key	Deletes the character to the left of the cursor.

CNTL-F or DEL CHAR (RUB OUT)	Deletes the character at the cursor.
CNTL-X	Deletes all characters to the left of the cursor.
CNTL-A	Deletes the character at the cursor and all characters to the right of the cursor.
CNTL-Z or CLEAR LINE	Deletes the entire current line.
Left Arrow key	Moves the cursor one character to the left.
Right Arrow key	Moves the cursor one character to the right.
HOME key	Operates with the left or right arrow key. Moves the cursor to the beginning of the line if pressed after the left arrow key. Moves the cursor to the end of the line if pressed after the right arrow key.
CNTL-C	Cancels the command in progress.
ESC key	Enters PSCOPE's internal editor.

Syntax Errors

A syntax error is an error in the command's format. When the debugger finds a syntax error, it displays the following message:

```
↑Syntax error
```

The arrow (↑) is aligned under the portion of the command line containing the error. If the error is located near the right edge of the screen, the message takes the following form:

```
Syntax error ↑
```

You can correct commands in which errors were detected by pressing ESC to invoke the internal editor and using the appropriate editor commands. You can then execute the corrected command after exiting from the editor.

The Internal Screen-Oriented Editor

PSCOPE contains a built-in editor for modifying debug procedures, LITERALLYs, patches, debug registers, the most recent command, and the most recent GO command. For example, use the editor to modify the specification of a break register.

The *edit-item* may be the name of a debug procedure, the name of a debug register, the name of a LITERALLY definition, the keyword PATCH followed by the patch's starting address, or the keyword GO (for editing the current GO specification).

Without an argument, the EDIT command is invoked with an empty buffer. Pressing the ESC key invokes the editor with the last command as the edit item.

The internal editor is screen-oriented. Figure 2-1 shows a typical menu. If you do not have an Intel terminal, you must include a CRT configuration file when you invoke PSCOPE. This file (called PSCOPE.CRT) contains the I/O sequences for cursor control. You can also use this file to reprogram special keys such as the direction arrows, ESC, and HOME keys to map their functions to other keys.

Because the internal editor is interactive and screen-oriented, you cannot use it if you specified the SUBMIT control when you invoked PSCOPE. Nor can you use the VIEW command when PSCOPE is running under SUBMIT.

Entering the Internal Editor

You can invoke the internal editor in the following two ways:

- By pressing the ESC key either at the debugger prompt or during command entry.

  If you press ESC immediately after the debugger prompt, the last command is recalled for editing.

  If you press ESC during command entry, you can edit the entire command being entered.

- By entering the EDIT command immediately after the debugger prompt.

Syntax

EDIT [*edit-item*]

Where:

edit-item is one of the following:

debug-symbol	is the name of an existing debug object of one of the debug types (not memory or user types) specified in Table 3-5. If you specified *debug-symbol*, its definition (the command that defined the debug object) is brought up for editing.
PATCH *address*	
GO	

If you specify PATCH, the corresponding debugger program PATCH is made available for editing.

If you specify GO, the text of the last GO command is brought up for editing.

If you do not specify anything, the editor invokes with an empty edit space.

Exiting the Internal Editor

After exiting the internal editor (using the QUIT command), you can either execute or ignore the command(s) you just edited.

Internal Editor Display

As shown in Figure 2-1, the internal editor uses the screen's two bottom lines for the edit message line and the menu option line. The remainder of the screen is the edit display area.

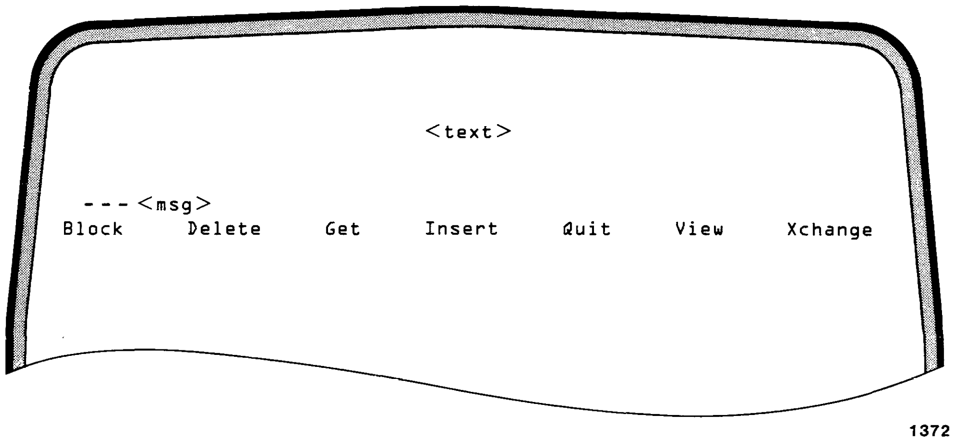

Figure 2-1 Editor Display

The internal editor displays a maximum of 79 columns of text but supports longer lines. For lines exceeding 79 characters, an exclamation point (!) is displayed as the last character to indicate that more text exists beyond the end of the display. When you move the cursor logically beyond the display, it remains physically in the right-most position in the line. All edit functions can act on text existing beyond the display area, but the display is not affected.

The internal editor displays the printable ASCII characters (20h to 7Eh). It displays unprintable characters as question marks (?). The internal editor considers the carriage return (CR) and the linefeed (LF) characters printable characters. The CR/LF combination acts as a single character called return. A tab is interpreted as four spaces.

Internal Editor Commands

The internal editor uses the cursor control keys and editor display options for command input.

Cursor Control Keys. The following cursor control keys control movement within the text being edited:

- Up arrow
- Down arrow
- Left arrow
- Right arrow
- HOME
- RETURN

The cursor control keys operate as described in the following sections.

Up Arrow Key. Pressing the up arrow key moves the cursor up one line from its current column position. If the cursor is already in the top line of the screen, pressing the up arrow key moves the cursor to the preceding line and displays the text with that line positioned six lines from the top of the screen (standard Intel terminals). The up arrow has no effect if the cursor is on the first line of the text being edited.

Down Arrow Key. Pressing the down arrow key moves the cursor one line down from its current column position. If the cursor is already in the last line on the screen, the text scrolls up one line. The down arrow key has no effect if the cursor is in the last line of text.

Left Arrow Key. Pressing the left arrow key moves the cursor one character to the left. If the cursor is at the beginning of a line, the cursor moves to the carriage return at the end of the preceding line. If the cursor is at the beginning of the screen, pressing the left arrow key moves the cursor to the end of the preceding line and displays the text with that line positioned six lines from the top of the screen (standard Intel terminals). The left arrow key has no effect if the cursor is at the beginning of the text.

Right Arrow Key. Pressing the right arrow key moves the cursor one character to the right. If the cursor is at the last character of the last line on the screen, the screen scrolls up one line. The right arrow key has no effect if the cursor is at the end of the text.

HOME Key. The HOME key is used with the directional cursor keys, as follows:

- Pressing HOME after pressing an up or down arrow key displays the previous or next screen of text, respectively. The cursor remains in the same relative location on the new page.
- Pressing HOME after pressing a left or right arrow key moves the cursor to the beginning or end of the line, respectively.

You can press HOME any number of times after pressing a directional key.

RETURN Key. Pressing the RETURN key when the editor is expecting a command moves the cursor to the beginning of the next line of text. If the cursor is in the last line of the text display area, the text is scrolled up one line. If the cursor is at the end of the text, pressing the RETURN key has no effect.

Delete Keys. You can use the following delete keys when the editor is at the command level or in the insert or exchange mode:

- RUBOUT key
  Deletes the character to the left of the cursor.
- CNTL-F
  Deletes the character at the cursor.

- CHAR DEL
  Deletes the character at the cursor.
- CNTL-X
  Deletes all characters to the left of the cursor.
- CNTL-A
  Deletes the character at the cursor and all characters to the right of the cursor.
- CNTL-Z
  Deletes the current line.
- CLEAR LINE
  Deletes the current line.

Menu Commands

The menu command provides the following command options:

- Block
- Delete
- Get
- Insert
- Quit
- View
- Xchange

To select each menu command, enter the first letter of the command (either lowercase or uppercase). The editor beeps if you give it an unexpected command character.

Enter CNTL-C to abort the menu command in progress. The editor ignores the CNTL-C if it is waiting for a command.

The menu always indicates which options are available. Some menu commands lead to sub-menus (for example, Quit and Block).

BLOCK Command (B)

The BLOCK command lets you mark off a block of text and place it in the block buffer for later retrieval (with the GET command).

To place text into the buffer, move the cursor to the first character in the block of text desired and press B. The editor displays the following sub-menu:

```
Buffer Delete
```

Move the cursor just beyond the character at the end of the block to be delimited and again enter B. (The beginning and ending characters of the block being delimited are marked with an at sign (@).) The text is now in the buffer.

Use the left, right, up and down arrow keys and the HOME and RETURN keys to move the cursor to the end of the block.

Note that the block buffer holds only one block of text. If you execute a block or delete command before retrieving the contents of the block buffer with the GET command, the original contents are overwritten by the new block of text.

DELETE Command (D)

The DELETE command lets you mark off a block of text, place it in the block buffer for later retrieval (with the GET command), and then delete it.

To place text in the buffer, move the cursor to the first character in the block of text desired and press D. The editor displays the following sub-menu:

```
Buffer Delete
```

Move the cursor just beyond the character at the end of the block to be delimited and again enter D. (The beginning and ending characters of the block being delimited are marked with an at sign (@).)

Use the left, right, up and down arrow keys and the HOME and RETURN keys to move the cursor to the end of the block.

Note that the block buffer holds only one block of text. If you execute a block or delete command before retrieving the contents of the block buffer with the GET command, the original contents are overwritten by the new block of text.

GET Command (G)

The GET command retrieves the contents of the block buffer (see the block and delete commands) and inserts it at the current cursor location. The block buffer is initially the null string.

You can move text from one part of the file to another by doing one of the following:

- Placing the text to be inserted in the block buffer with the delete command.
- Moving the cursor (with the cursor control keys) to where you want the text inserted.
- Entering the GET command.

INSERT Command (I)

The INSERT command puts the editor into insert mode, which is indicated by [insert] on the menu line. Each character you enter is then inserted at the cursor until you press ESC. The display echoes the new text as you insert each character.

In insert mode, you can use the cursor control keys (left, right, up and down arrow, HOME and RETURN) and the delete keys. Pressing RETURN after inserting text in the last line scrolls the text up one line.

If the cursor is positioned beyond the end of a line when entering text in insert mode, the cursor moves to the point immediately before the end (or carriage return) of the current line, and the insertion begins beyond the line.

QUIT Command (Q)

The QUIT command lets you exit the editor and either pass or not pass a command back to the debugger. The editor displays the following sub-menu when you enter Q:

```
Abort   Execute
```

Enter A (abort) to exit the editor without passing a command back to the debugger.

Enter E (execute) to exit the editor and process the edited text as a command.

VIEW Command (V)

The VIEW command redisplays the screen with the line containing the cursor positioned in the middle of the screen, unless centering places the beginning of the text below the top of the screen.

XCHANGE Command (X)

The XCHANGE command puts the editor into exchange mode, indicated by [exchange] on the menu line. The XCHANGE command lets you replace the character at the cursor, one for one. The cursor moves one character to the right each time you replace a character.

Press ESC to end the exchange mode.

While in exchange mode, you can use any of the cursor control keys to move the cursor. Characters you enter beyond the end of a line are inserted before the RETURN in that line.

The VIEW Command

The VIEW command allows you to examine files without exiting PSCOPE. This is a PSCOPE command; it is distinct from the view command that belongs to the PSCOPE internal editor.

Syntax

VIEW *pathname*

Where:

pathname is the fully-qualified name of the text file you want to examine.

Description

The VIEW command is menu driven. After you issue the command, the first 23 lines of the specified file appear on the screen. The last two lines of the screen contain the VIEW menu as shown in the following example:

```
----- VIEW :f1:pager.src
Again  Find  -find  Jump  Quit  Roll  Set  View
```

Choose a menu item by entering its first letter (VIEW is not case-sensitive). For example, entering Q chooses Quit, which terminates the VIEW command and returns you to the PSCOPE prompt.

Again — Repeats the last command option. For example, if you just searched for the text string "reset", entering A searches for the next occurrence of that text string.

Find — Searches for a text string in the forward direction. After you enter the F, VIEW requests a text string. Enter the text string and terminate it with an ESC. Find remembers the specified text string. If you enter another F, you can search for the next occurrence of the string by just entering an ESC.

-find — Searches for a text string in the reverse direction. After you enter the hyphen (-), VIEW requests a text string. Enter the text string and terminate it with an ESC.

Jump — Moves the cursor to the start or the end of the file. After you enter the J, VIEW presents the following new menu:

```
-----
Start   End
```

Choose S for the start of the file and E for the end of the file.

Quit — Returns you to the PSCOPE prompt.

Roll — Moves the cursor to the last line on the screen, then moves the screen down through the file one line at a time. You can stop rolling with CNTL-S and resume rolling with CNTL-Q. Terminate the roll with an ESC.

Set — Determines the way VIEW displays the file. After you enter the S, VIEW presents the following new menu:

```
Leftcol    Tabs    Viewrow
```

Choose one item by entering its first letter.

Leftcol is zero by default; zero identifies the first column. You can change the first column displayed on the screen. This is useful when your file contains rows longer than the screen display. When leftcol is other than zero, column zero contains an exclamation point (!).

Tab is four spaces by default. You can alter the tab settings in your file display. This alteration is only for the display. Your file is not edited.

Viewrow is line five by default. The rows on the screen are labeled 0 through 22. Setting viewrow to a row number moves the row the cursor is on to that row number.

CHAPTER 3 COMMAND LANGUAGE AND EXPRESSIONS

This chapter describes PSCOPE command line format. It gives an overview of tokens and symbol object types. It defines the operands and operators you can use in expressions and explains the rules for combining different expression types.

Tokens

Tokens are the smallest meaningful units in a command line. Each token belongs to one of the following classes:

- Delimiters
- Names
- Line numbers
- Numeric constants
- Character string constants
- Operators
- Comments

Delimiters

A delimiter is a character or a pair of characters that separate or mark the beginning or end of a token. The debugger recognizes delimiters for names, lines, commands, strings, range specifications, modules, lists, and comments. Table 3-1 lists the delimiters.

Table 3-1 Special Character Delimiters

Character	Description	Function
	Space	Blank separator
	Tab	Blank separator
<cr>	Carriage return	Line terminator
&	Ampersand	Continuation line indicator
;	Semicolon	Command separator
'	Apostrophe	String delimiter
.	Dot	Compound name separator
"	Quotation marks	User symbol flag
:	Colon	Module name prefix
,	Comma	List separator
/*	Slash asterisk	Start-of-comment delimiter
*/	Asterisk slash	End-of-commment delimiter

Names

There are three types of names:

- Keywords
  Keywords are reserved elements of the debugger command language. Keywords have special meaning within the debugger language and, therefore, cannot be used in other ways. For example, a keyword cannot be used as a debug symbol. Appendix G contains a complete list of PSCOPE keywords.

- Program symbols
  The compiler includes program symbol information in your object file when you compile your source module with the compiler DEBUG control. The debugger enters the program symbols into its symbol table when you load your object code.

- Debug symbols
  Debug symbols are any symbols defined by the user in a debug session. Chapters 7 and 8 describe debug symbols and their use.

Names Format

A name is a sequence of letters, digits, underscores, at-signs, question marks, or dollar signs, of which the first character must be a letter, underscore, at-sign, or question mark. This format is summarized in Table 3-2.

Table 3-2 Names Format

Valid First Character	Valid Remaining Characters	Description
A-Z	-Z	Letters
@	@	At-sign
?	?	Question mark
	0-9	Digits
	$	Dollar sign*
_	_	Underscore

*Embedded dollar signs are ignored by the debugger and may be used to improve the readability of a name.

To convert Pascal labels (which are numbers in the source code) to the name format, the compiler removes leading zeros from each label and attaches a leading at sign (@) to the label. For example, the label '9999' in module 'DC' is as follows:

```
:DC.@9999
```

The debugger accepts names of unlimited length; however, it uses only the first 40 characters. The debugger interprets uppercase and lowercase characters as the same character (i.e., b and B are interpreted as the same character).

Referencing Names

Names have the following precedence: command keywords, debug symbols, program symbols. The debugger checks symbols it encounters to see if they are keywords. If a symbol is not a keyword, the debugger checks to see if it is the name of a debugger object. If the symbol is not the name of a debugger object, the debugger assumes that the symbol is the name of a program object.

A debug symbol must not duplicate a keyword. A debug symbol is referenced by entering its name.

A program symbol name may duplicate a keyword or debug symbol if you precede it with quotation marks ("), as shown in the following example:

```
FOO + "Line
```

In all cases, you can reference a program symbol by using a fully qualified name. A fully qualified name is a compound name, where each level of the name is specified beginning with the module name, including any names of enclosing procedures, and ending with the symbol name. For example, use the following reference for the variable "TEST" in procedure "GETSCORE" of module "SYSTEM":

```
:SYSTEM.GETSCORE.TEST
```

Alternatively, you can use a partially qualified reference depending on where you are in the program. A partially qualified reference lets you abbreviate the reference by omitting leading parts of the reference, such as the module name. This method is explained further in Chapter 5.

Line Numbers

The compiler produces line numbers. The following format is for a line number reference.

[:*module-name*] #*line-number*

For example:

```
:MOD1#23
```
or
```
#23
```

Note that module names must begin with a colon (:), as shown in the first example.

Numeric Constants

Numeric constants are integers or floating point numbers.

Integers

An integer constant is a number consisting of one or more digits and an optional one-character suffix that identifies the number base. The suffix is not required if the integer's number base corresponds to the current default base set with the BASE command. (Note that a 0 (zero) must precede hexadecimal numbers beginning with the letters A through F to distinguish them from names.)

You can enter alphabetic hexadecimal digits A through F and base suffixes (Y, T, H, K) in either uppercase or lowercase.

Table 3-3 summarizes integer constants.

Table 3-3 Elements of Integer Constants

Number Base	Valid Digits	Suffix	Example
Binary (base 2)	0,1	Y	11110011Y
Decimal (base 10)	0-9	T	243T
Hexadecimal (base 16)	0-9,A-F	H	0F3H
Decimal multiple of 1024T	0-9	K	4K

Floating Point Numbers

Floating point numbers are decimal numbers consisting of a significand (expressed in one or more digits), a decimal point, one or more additional digits, and an optional exponent. The exponent consists of the letter E followed by a signed integer value. For example, the following decimal value:

0.24 x 10 −2

May be expressed as follows:

0.0024

or

0.24E −2

Use the following guidelines when working with floating point numbers:

- You must use the decimal point in a floating point number to distinguish the E as a scale factor (e.g., 44.0E30); otherwise, E might look like a digit in a hexadecimal number (e.g., 44E30).
- Digits must appear on both sides of the decimal point. For example, 0.85E2 and 85.0E2 are acceptable, but .85E2 is not because there is no digit before the decimal point.
- A floating point number must have a value in the following range: 64-bit mantissa, 15-bit exponent, and a sign bit, for a total of 80 bits.

Floating point (real) arithmetic used in PSCOPE conforms to the proposed IEEE standard for binary floating point arithmetic. This standard specifies internal data representations, normalization, rounding modes, and error handling. All real arithmetic performed by Intel microprocessors and software (except VSP software) conforms to this standard. (*The 8086 Family User's Manual* explains the proposed IEEE floating point standard.)

Character String Constants

The term *string* in a command format means a sequence of one or more ASCII printing characters enclosed in delimiters of apostrophes. Examples are as follows:

'ABCDE'
'Testing 1 2 3'
'X'
'THIS IS A STRING'
'This is a string'

To enter a literal apostrophe inside a string, use two apostrophes to distinguish the literal apostrophe from those used as delimiters.

For example, the following:

'WHAT''S UP?'

Is stored as follows:

WHAT'S UP?

The debugger accepts strings of up to 254 characters, not counting the enclosing delimiters. You can extend strings over more than one line; the debugger concatenates (links) adjacent strings into a single string. You can separate adjacent strings with spaces, tabs, or carriage returns.

When a string value is stored in memory, the value is the one-byte ASCII value of each character. If the string has more than one character, the debugger stores the subsequent ASCII values in consecutive locations.

Operators

The command language contains tokens that serve as operators. Table 3-4 lists the special character operators that the debugger recognizes.

Table 3-4 Special Character Operators

Operator	Description	Function
*	Multiply sign	L Multiplication
−	Minus sign	Negation or subtraction
+	Plus sign	Identity or addition
/	Slash	Division
= =	Double equal signs	Equality
<>	Angle brackets	Inequality
>	Angle bracket	Greater than
<	Angle bracket	Less than
> =	Bracket,equals	Greater than or equal to
< =	Bracket,equals	Less than or equal to
.	Dot	Address of (prefix operator)
( )	Parenthesis	Bracketing
[ ]	Square brackets	Array indexing
=	Equal sign	Assignment
:	Colon	Pointer constructor

In addition to the special character operators shown in Table 3-4, PSCOPE also supports several keyword operators. OR, XOR, AND, and NOT are the conventional Boolean operators. MOD is the conventional remainder (or modulo) operator (as defined in Pascal), extended to also work with real numbers.

Comments

The debugger ignores characters enclosed by the comment delimiters, /* and */. The following comments would be ignored by PSCOPE:

```
/* THIS PROGRAM WAS DEBUGGED WITH PSCOPE */

/* THIS COMMENT IS
SPREAD OVER TWO LINES. */
```

Types of Symbol Objects

All objects referred to by debug or program symbols have an associated type. Symbols are divided into three types. The first two types, memory and debug, are basic types whose names and definitions are determined by PSCOPE. The third type, referred to as user types, consists of user-defined symbols. PSCOPE obtains this type of information from the debug information in the load module of a program. Table 3-5 lists the standard types recognized by the debugger.

Table 3-5 Standard Symbol Object Types

Symbol Type	Object Type	Definition
Memory	BOOLEAN	TRUE or FALSE
	CHAR	String of ASCII character(s)
	POINTER	Pointer value
	BYTE	Unsigned 8-bit quantity
	WORD	Unsigned 16-bit quantity
	DWORD	Unsigned 32-bit quantity
	SELECTOR	Unsigned 16-bit quantity
	ADDRESS	Unsigned 16-bit quantity
	SHORTINT	Signed 8-bit quantity
	INTEGER	Signed 16-bit quantity
	LONGINT	Signed 32-bit quantity
	EXTINT	Signed 64-bit quantity
	BCD	Signed 18-digit binary coded decimal number
	REAL	32-bit floating point number
	LONGREAL	64-bit floating point number
	TEMPREAL	80-bit floating point number
Debug	PROC	Debug procedure
	LITERALLY	String macro
	BRKREG	Break register
	TRCREG	Trace register
	PATCH	Debug patch code
User	ARRAY	Array
	RECORD	Pascal record or PL/M structure
	PROCEDURE	User program procedure or function
	LABEL	User program label
	LINE	User program line number
	FILE	User file
	MODULE	User program module
	ENUMERATION	User-defined PASCAL enumerated type

Compiler/Assembler Type vs. PSCOPE Type Names

PSCOPE does not always identify a program variable as the same memory type as the user program. For example, PSCOPE considers an ASM-86 program variable of type DWORD to be a program variable of type POINTER. Table 3-6 lists the memory type differences between ASM-86, PL/M-86, Pascal-86, FORTRAN-86, and PSCOPE.

Table 3-6 Compiler/Assembler Type vs. PSCOPE Type Names

ASM-86	PSCOPE for ASM-86	PL/M-86	PSCOPE for PL/M-86
BYTE	BYTE	BYTE	BYTE
WORD	WORD	WORD	WORD
		INTEGER	INTEGER
		SELECTOR	SELECTOR
POINTER	POINTER	POINTER (small)	ADDRESS
		POINTER (compact or large)	POINTER
DWORD	POINTER	DWORD	DWORD
		REAL	REAL
QWORD	LONGREAL		
TBYTE	TEMPREAL		
STRUC	RECORD	STRUCTURE	RECORD
RECORD			
RFIELD			
STRUC ARRAY	ARRAY OR RECORD	STRUCTURE ARRAY	ARRAY OF RECORD

Pascal-86	PSCOPE for Pascal-86	FORTRAN-86	PSCOPE for FORTRAN-86
BOOLEAN	BOOLEAN	LOGICAL*1	BOOLEAN
CHAR	CHAR	INTEGER*1	SHORTINT
		CHARACTER*n	CHAR
WORD	WORD	LOGICAL*2	WORD
INTEGER	INTEGER	INTEGER*2	INTEGER
LONGINT	LONGINT	LOGICAL*4	DWORD
REAL	REAL	REAL*4	REAL
		INTEGER*4	LONGINT
LONGREAL	LONGREAL	REAL*8	LONGREAL
		DOUBLE PRECISION	
TEMPREAL	TEMPREAL	TEMPREAL	TEMPREAL
RECORD	RECORD		
ARRAY	ARRAY		
FILE	FILE		
SET	SET		

Pascal-86 uses POINTER types to allocate, access, and de-allocate dynamic variables. Pascal POINTERs are not analagous to PL/M-86 POINTERs.

Expressions

Expressions can be used as command arguments to specify numeric, Boolean, or string values.

Expressions can be one of the following:

- A single number, constant, or symbolic reference. Example areas follow:

0	(Number without explicit suffix)
100H	(Hexadecimal numeric constant)
'A'	(One-character string constant)
X	(Symbolic reference yielding a value)

- A formula applying operators and functions to numbers, constants, and symbolic references as operands.

  The debugger performs the calculation, using parenthetical and operator precedence and left-to-right order to determine the sequence of operations.

Examples of expressions are as follows:

```
2 + 3
174 / 4
0100H + 00FH
2 * (6 + 4)
.BUFFER + 2
:MOD1.SAM + 21
```

To evaluate and display the value of an expression, enter the expression. An expression evaluates to a type component and a value component. The rules for determining expression types and values are discussed later in this chapter.

Operands

You can use the following types of operands in expressions:

- Numeric constants
- String constants
- Program symbol references
- Machine register references
- Memory references with explicit typing
- Line number references
- Debug variable references
- Debug procedure calls
- Debug procedure parameters
- Debug built-in function calls

Numeric Constants

Numeric constants can be integers or floating point numbers, described previously in this chapter.

String Constants

You can use character strings as arithmetic values in expressions, as follows:

- A one-character string, which has a byte value corresponding to the character's ASCII representation. For example, the string constant 'A' has the value 41H.
- Longer string constants (up to 254 characters), which you may also use as parameters of debug procedures or as arguments to built-in string functions.

Program Symbol References

When you enter a program symbol reference as an operand, its value is obtained from the debugger symbol table and used in the associated expression.

To reference the value of a program symbol, use the following format:

symbolic-reference

Where:

symbolic-reference	is a fully or partially qualified reference as described in Chapter 5.

To reference the address of a program symbol, prefix *symbolic-reference* with the dot operator as shown in the following example:

.symbolic-reference

Machine Register References

You can reference the 8086 registers symbolically, within expressions, just like variables.

The registers are as follows:

AH	AL	AX	BH	BL	BP
BX	CH	CL	CS	CX	DH
DI	DL	DS	DX	ES	FH
FL	FLAG	IP	SI	SP	SS

Memory References with Explicit Typing

To reference a memory location and interpret it as a particular type of object in an expression, use the following format:

memory-type location

Where:

memory-type	is one of the object types shown in Table 3-5.
location	is an expression that evaluates to a pointer. The pointer must refer to a single valid address.

The memory reference format uses *memory-type* to interpret the area of memory pointed to by the address expression. The following example:

```
BYTE .FOO
```

interprets the first byte at the address of FOO as a byte regardless of the type of FOO.

Other examples of memory references are as follows:

```
BYTE (.buffer + bufindex)
WORD DS:22H
X + (INTEGER .ABLE)
(LONGREAL TEST) MOD 5
```

Line Number References

When you use a line number reference in an expression, you get the address of the first instruction generated by the compiler for the source line number. In other words, you are referencing a program location through the line number.

If different modules (each with their own statement numbers) are linked, you must sometimes specify a module name in the line number reference, as follows:

[:*module*] #*line-number*

You must use fully qualified line references (those with a module specified) when referring to a line number that is not in the current default module (determined by the current name scope). You can use partially qualified references (those without a module name) when the line reference is in the current default module.

The statement number must be a decimal integer. Examples of line number references are as follows:

```
#45
:TEST#1
#23
```

Debug Variable References

After defining a debug variable (described in Chapter 7), you can use its value as an operand within an expression. PSCOPE also includes the following predefined variables:

Name	Object Type	Use
BASE	BYTE	Current default numeric base
NAMESCOPE	POINTER	Starting point for program symbol lookup
$	POINTER	Value of CS:IP

Debug Procedure Calls and Parameter References

After defining a debug procedure within the debugger (described in Chapter 8), you can call that procedure from within an expression and have it return a value.

PSCOPE includes several built-in functions, as follows:

Name	Use
SUBSTR	Substring selection (Chapter 10)
CONCAT	String concatenation (Chapter 10)
STRLEN	String length (Chapter 10)
CI	Console character input (Chapter 10)
ACTIVE	Testing for program symbol accessibility (Chapter 5)
SELECTOR$OF	Segment portion of pointer (Chapter 10)
OFFSET$OF	Offset portion of pointer (Chapter 10)

Operators

An expression can contain any combination of unary and binary operators.

The debugger recognizes five groups of operators: dereference operators, pointer selector operators (pointer selection uses built-in functions), arithmetic operators, memory-type operators, relational operators, and logical operators.

Table 3-7 shows the operators in each group in descending order (from highest to lowest precedence). In the table, all operators apply to both real and integer operands unless otherwise noted. All operations are binary unless specified as unary operations.

Table 3-7 Precedence of Operators (Highest to Lowest)

Group	Operator	Operation	Precedence
Arithmetic	+	Unary plus	1
	−	Unary minus (2's complement)	
	*	Multiplication	2
	/	Division	
	MOD	Integer remainder	
	+	Addition	3
	−	Subtraction	
Relational	= =	Is equal to	4
	>	Is greater than	
	<	Is less than	
	>=	Is greater than or equal to	
	<=	Is less than or equal to	
	<>	Is not equal to	
Logical	NOT	1's complement	5
	AND	Logical AND	6
	OR	Logical OR	7
	XOR	Exclusive OR	
Memory Type	BOOLEAN	TRUE or FALSE value	8
	CHAR	8-bit ASCII character	
	POINTER	Pointer value	
	BYTE	Unsigned 8-bit quantity	
	WORD	Unsigned 16-bit quantity	
	ADDRESS	Unsigned 16-bit quantity	
	SELECTOR	Unsigned 16-bit quantity	
	DWORD	Unsigned 32-bit quantity	
	SHORTINT	Signed 8-bit quantity	
	INTEGER	Signed 16-bit quantity	
	LONGINT	Signed 32-bit quantity	
	EXTINT	Signed 64-bit quantity	
	BCD	Signed 18-digit binary coded decimal number	
	REAL	32-bit floating point number	
	LONGREAL	64-bit floating point number	
	TEMPREAL	80-bit floating point number	

Type Conversions

PSCOPE automatically converts values from one type to another during expression evaluation and during modify commands (described in Chapter 7).

The following type classification is useful for describing the type conversions performed by PSCOPE. The numbers in parentheses are the number of bytes used to store a value of that type (precision). The type in each class considered to have the maximum precision for that class is listed at the bottom of each column.

Unsigned	**Signed**	**Real**
BYTE (1)	SHORTINT (1)	REAL (4)
WORD (2)	INTEGER (2)	LONGREAL (8)
ADDRESS (2)	LONGINT (4)	EXTINT (8)
SELECTOR (2)		BCD (10)
DWORD (4)		TEMPREAL (10)

Note that Pascal enumeration types are treated as unsigned types of the smallest precision necessary to hold the ordinal representation of that type.

Type Conversions for Expressions

The automatic type conversions that PSCOPE performs during expression evaluation are dictated by the following two things:

- The type of value expected by an operator or function.
- Not losing any significant portion of a value when performing an operation.

To accomplish these objectives, PSCOPE performs the following type conversions during expression evaluation:

1. Each value used by the operation is extended to the maximum precision for that type class.
2. Each value is converted to the type required by the operation.
3. The operation is performed, and the resulting value is left in its maximum precision and passed on as a value to other operations in the expression (if any).

Type Conversions for Assignments

The type of the source value and the type of the target variable dictate the automatic type conversions PSCOPE performs during assignment. In general, the type conversion proceeds as follows:

1. The source value is extended to the maximum precision for its type class.
2. The resulting value is converted to the maximum precision type of the type class of the target variable.
3. The resulting value is truncated to the exact precision required for the target variable type.

There are a few minor exceptions to these rules when non-numeric types are involved (CHAR, POINTER, and BOOLEAN). In these cases, conversions between some types may not be allowed or are handled as special cases. However, even for these types, the automatic conversions performed by PSCOPE extend those provided by Pascal and PL/M.

CHAPTER 4 LOADING AND EXECUTING PROGRAMS

This chapter describes how to load programs and control execution by setting breakpoints with the GO command and by stepping through the program.

The LOAD Command

The LOAD command loads the program file you want to debug. With the LOAD command, you can specify that certain debugging information not be loaded, that the 8087 emulator be linked with the program file (ISIS and iNDX only), or that 8087 chip support be included (iRMX-86 only).

Syntax

LOAD *file* [*load-Option*]* [CONTROLS *command-tail*]

Where:

file	is the name of the program file you want to debug. This name may be a complete pathname.
load-option	is one of the following:
E8087	informs PSCOPE that your program uses the 8087 software emulator. Select this option if your program performs real arithmetic and PSCOPE is running under the ISIS or iNDX operating systems. (*The 8087 Support Library Reference Manual* contains information on the 8087 emulator and numeric support.)
CH8087	informs PSCOPE that your program uses the iSBC® 337 MULTIMODULE™ (the 8087 hardware). Select this option if your program performs real arithmetic and PSCOPE is running under the iRMX-86 operating system.
NOLINES	specifies that line number information is not loaded from the program file.
NOSYMBOLS	specifies that symbol information is not loaded from the program file.
command-tail	is any arbitrary text expected by your program. For example, if your program expects parameters on its invocation line, (a PL/M-86 program that uses the DQ$GET$ARGUMENT system call), those parameters would appear here.

Description

The load command loads the specified *file* into the microcomputer system memory. The debug information in *file* is processed to produce PSCOPE's symbol table of information about the loaded program. If necessary, PSCOPE sends part of this symbol table to disk, using a temporary work file.

The controls command lets you specify information your program may need to execute.

You can extend the invocation line as you would any command line.

The object file must conform to the 8086 object module formats. The object program must be position independent code (PIC) or load time locatable (LTL), with absolute segments for interrupt vectors only. Because they do not use absolute addressing, PIC and LTL object programs are less likely to cause conflicts with PSCOPE's address base. PSCOPE issues an error message if you try to load a program which is neither PIC nor LTL.

The loaded object file must contain information initializing the CS, IP, SS, and DS 8086 registers. If these registers are not initialized during loading, PSCOPE displays an error message.

PSCOPE removes all previously set break registers (BRKREGs), trace registers (TRCREGs), and patches when you load a program, even if the load is unsuccessful.

Example

The following example uses the LOAD command.

```
*LOAD dc.86
```

The GO Command

The GO command transfers execution control to the loaded program.

Syntax (simplified)

GO [TIL *expression* [, *expression*]*]

or

GO FOREVER

Where:

expression	is a symbolic expression specifying an address in the program code where you want a breakpoint.

Description

The GO command transfers control from PSCOPE to the program under debug and specifies the conditions under which the user program stops executing and transfers control back to PSCOPE.

TIL lets you specify any number of breakpoints. Breakpoints are stopping points at specific addresses in your program.

GO FOREVER specifies that your program will be executed without breakpoints.

If you do not specify neither TIL nor FOREVER, control passes to your program using the same breakpoints as those used by the previous GO command (if any).

During program execution, you can interrupt execution any time by entering CNTL-C. Note that entering CNTL-C while your program is executing the UDI primitive DQ$Read from :CI: causes an end of file condition on :CI:. The sample program DC (found in Appendix E) is included on the PSCOPE disk and shows one way of avoiding this problem in its procedure GET_LINE.

Note that the GO command always executes at least one instruction, so you can set a breakpoint at the current execution point.

Chapter 11 explains how to use the GO command with break and trace registers.

Example

Each of the following examples is based on the sample program DC found in Appendix E.

The following example executes the program using two breakpoints, one set at a specific statement and the other set at the address of a procedure.

```
*go til :dc#26, :dc.gettoken
```

The following example executes using the previous breakpoints:

```
*go
```

The following example executes using no breakpoints at all:

```
*go forever
```

The LSTEP and PSTEP Commands

The LSTEP and PSTEP commands let you single-step through the program by executing numbered (high-level) language statements.

Syntax

```
LSTEP
```

or

```
PSTEP
```

Description

PSTEP and LSTEP are PSCOPE's source-level statement stepping commands. PSTEP treats a procedure or function as a single statement, executing it entirely before returning control to you at the next statement. LSTEP steps through procedures and functions one statement at a time.

IF you enter PSTEP or LSTEP from a normal command level, PSCOPE returns the following message after executing the statement:

[Step at *line-number*]

Where:

line-number	refers to the current execution point, after the step is complete.

PSCOPE does not print a message if either command is issued from a nested PSCOPE command (DO, IF, COUNT or REPEAT).

PSTEP and LSTEP must be entered when the execution point is at the start of a statement. Note that any code patches are executed when stepping, but that no other user-set breakpoints are active (run-time exceptions, however, are still trapped).

Note also that both PSTEP and LSTEP require line information; stepping from a location with line information into one without it causes execution to continue until a known line is reached.

Example

The following example uses the PSTEP command.

```
*pstep
```

The ISTEP Command

The ISTEP command allows you to single-step your program through assembly language instructions.

Syntax

```
ISTEP
```

Description

The ISTEP command executes one assembly language instruction, displays the next assembly language instruction to be executed, then halts. When stepping through an intruction that alters a segment register the step will execute two instructions.

CHAPTER 5 EXAMINING AND MODIFYING PROGRAM SYMBOLS

This chapter describes how to reference objects (variables, procedures, etc.) in a program you have loaded and how to display and modify program objects. It explains the concept of current name scope (CNS) and how CNS allows abbreviated (partially qualified) references to program symbols. The following commands are explained in this chapter:

- Program symbol references
  Display program symbol
  Change program symbol
  Change 8086/8088 flags
  Change 8086/8088 registers
  The REGS command
  Change 8087 registers
  Change name scope
  Active function

- Memory manipulation commands
  Display memory
  Modify memory
  The single line assembler/disassembler

Program Symbol References

Program symbols are produced by the compiler (when you specify the DEBUG option) and loaded into the debugger symbol table with the LOAD command (described in Chapter 4).

Current Name Scope

The current name scope (CNS) is the set of symbols accessible from a specific location in the program, as defined by the compiler. This program location is called the debug cursor and changes as the program execution point changes. You can also change the debug cursor with the NAMESCOPE command (described later in this chapter).

In a Pascal procedure, the scope of local variables is the procedure they are defined in; outside that scope, the symbols have either no meaning or an entirely different meaning. To illustrate, suppose you have two variables of the same name in two different procedures. In this case, specifying the variable name alone is not sufficient. You must also specify the procedure in which the variable is found.

References to program symbols can be fully or partially qualified, as explained in the following section.

Fully Qualified References

A fully qualified reference always begins with a module name. It also specifies the name of the procedure (or procedures) containing the referenced symbol, including the names of all procedures enclosing the symbol.

Syntax

:module-name [*.procedure-symbol*]* *.symbol-reference*

Where:

module.name	is the name of the load module.
procedure-symbol	is the name of the procedure.
symbol-reference	is one of the following:

variable-symbol	is a program symbol that specifies a program variable.

variable-name [*qualifier*]*

variable-name	is a variable name.
qualifier	is one of the following:

left-bracket expr [, *expr*]* *right-bracket*

left-bracket and *right-bracket* are the characters [and] and specify array indexing.

expr	is an expression used to index an array variable.
.field-symbol	specifies a field within a record (structure) variable.
pointer	is the character ↑and indicates Pascal pointer dereferencing.

Description

For fully qualified references, *procedure-symbols* must be direct references to procedure names; procedure variables are not allowed.

To illustrate, assume you want to reference a parameter named C contained in the function DIGIT in the procedure GET_TOKEN of module DC. Your fully qualified reference to the variable C is the following:

```
:dc.get_token.digit.c
```

A fully qualified reference establishes a path from the module level of the program down to the desired symbol.

For example, to reference the variable named VARIABLE_INDEX of the procedure FACTOR, you must specify both the variable and the procedure containing it. Specifying the following:

```
:dc.variable_index
```

does not work. Because PSCOPE does not know which procedure contains the variable VARIABLE_INDEX, it assumes that VARIABLE_INDEX is either a variable or a procedure declared at the main level of DC. Specifying the following:

```
:dc.factor.variable_index
```

establishes a path for PSCOPE to follow to the desired variable.

Structures nested within other structures (or records within records) are referenced in the same way (outer to inner levels), thus establishing a path that specifies all enclosing structures.

Partially Qualified References

A partially qualified reference omits some or all of the leading part of a fully qualified reference, depending on the current name scope (CNS).

Using the previous fully qualified reference example (:dc.get_token.digit.c), you can use varying degrees of partially qualified references, depending upon the current name scope. For example, if the CNS is within the procedure GET_TOKEN, the partially qualified reference DIGIT.C is sufficient. If, however, the debug cursor is at the main level of module DC, the partially qualified reference should be as follows:

```
get_token.digit.c
```

The fully qualified reference is required when the debug cursor is in a module other than DC.

The following examples illustrate fully qualified references (FQR) and partially qualified references (PQR):

FQR	PQR	Required CNS for PQR
:dc.get_token.digit.c	c	:dc.get_token.digit
	digit.c	:dc.get_token
:dc.term.term_value	term_value	:dc.term
	term.term_value	:dc
dc.@9999	@9999	:dc

Note that changing the name scope or using a more qualified reference lets you reference symbols outside of the symbols scope. This is useful for operations like setting breakpoints, patches, etc. However, referencing symbols outside of their scope may not let you examine the value of some local variables because the values are undefined outside their scope.

Display Program Symbol

You can obtain the value of the program symbol, like the value of an expression, by entering the name of the program object whose value you want.

Syntax

symbol-name

Where:

symbol-name	is either a fully qualified or a partially qualified symbolic reference to a program symbol.

Description

Entering *symbol-name* yields a typed value. The format of the symbol value displayed depends on the referenced symbol type. Table 5-1 describes default display formats.

Example

The following example references a CHAR variable C (with a value of a) in the sample program DC (found in Appendix E):

```
*:dc.get_token.digit.c
a
*c
a
```

The following example displays the value of a field CLASS in the record T. Note that class is of type enumeration:

```
*t.class
3
```

The following example references an element of an array within a record. Note that the components of the array are of type CHAR:

```
*buffer.str[1]
a
```

Table 5-1 shows the default display formats for predefined program symbol types. These formats are used by the display program symbol, unformatted WRITE, and display memory commands.

Table 5-1 Default Display Formats

Predefined Symbol Types	
Type	**Display**
BOOLEAN	Byte value displays FALSE if low order bit is 0 or TRUE if low order bit is 1.
BYTE	Unsigned 8-bit quantity in current base.
CHAR	8-bit ASCII character.
WORD	Unsigned 16-bit quantity in current base.
DWORD	Unsigned 32-bit quantity in current base.
POINTER	Pair of words as nnnn:nnnn (always hex).
ADDRESS	Unsigned 16-bit quantity in current base.
SELECTOR	Unsigned 16-bit quantity in current base.
SHORTINT	Signed 8-bit quantity in current base.
INTEGER	Signed 16-bit quantity in current base.
LONGINT	Signed 32-bit quantity in current base.
EXTINT	Signed 64-bit quantity in current base.
BCD	Signed 18-digit quantity in current base.
REAL	32-bit quantity in floating point notation, (always decimal).
LONGREAL	64-bit quantity in floating point notation, (always decimal).
TEMPREAL	80-bit quantity in floating point notation,(always decimal).
ENUMERATION	Elements displayed as ordinal number.
ARRAY	Array components that are predefined symbol types are displayed as described previously. PSCOPE does not display entire arrays or array components that are not predefined types.
PROCEDURE	Entry point address.
LABEL	Address.
FILE	No display.
MODULE	Address.
RECORD	Record fields that are predefined symbol types are displayed as described previously. PSCOPE does not display entire records or fields that are not pre-defined types.

Change Program Symbol

This section shows you how to change the value of a program symbol.

Syntax

symbol-name = new-value

Where:

symbol-name	is the name of a program symbol.
new-value	is the new value for *symbol-name*. The *new-value* can be an expression that evaluates to the correct type for the assignment. The expression can contain program or debug symbol references, constants, strings, etc.

Description

The new value must yield a typed value that matches the type of the program symbol referenced, or the new value can be forced to match under the type coercion rules given in Chapter 3.

The debugger displays an error message if the change value yields the wrong type.

Example

The following example illustrates how to change the value of a program symbol.

```
*buffer.str[buffer.index] = 'x'
*buffer.index = buffer.index + 1
*term.factor_1_value = 0
*:dc.variable_table['a'] = -23
```

Change 8086/8088 Flags

PSCOPE allows you to display and modify 8086/8088 flags.

Syntax

$$\left\{\begin{array}{l}\text{FLAG}\\ \text{FH}\\ \text{FL}\\ \textit{8086/8088-flag}\end{array}\right\}\ [=\textit{expression}]$$

Where:

expression	resolves to a 16-bit number to be loaded into the FLAG word.
8086-8088-flag	is a symbol representing one of the 8086/8088 flags.
FLAG	represents the flag word.

FH	represents the upper (most significant) byte of the flag word.
FL	represents the lower (least significant) byte of the flag word.

8086-8088 Flag	Description	Type
OFL	Overflow flag	BOOLEAN
DFL	Direction flag	BOOLEAN
IFL	Interrupt flag	BOOLEAN
TFL	Trap flag	BOOLEAN
SFL	Sign flag	BOOLEAN
ZFL	Zero flag	BOOLEAN
AFL	Auxiliary flag	BOOLEAN
PFL	Parity flag	BOOLEAN
CFL	Carry flag	BOOLEAN

Interpret the FLAG word as follows:

15 0

X	X	X	X	OFL	DFL	IFL	TFL	SFL	ZFL	X	AFL	X	PFL	X	CFL

where X represents a don't-care bit.

Description

The FLAG word displays in the current radix. The flag names are pseudo-variables and may be used within an expression.

These pseudo-variables only affect the copy of the 8086/8088 flags used by the program being debugged. These pseudo-variables do not affect the copy of the 8086/8088 flags used by PSCOPE.

Example

To display the 8086/8088 flags, enter the following:

```
*BASE=10Y;BASE   /*Setting the current radix to binary*/
BINARY
*FLAG
1111000010000110
```

To set the overflow flag (OFL) enter the following:

```
*OFL=TRUE
*OFL
TRUE
```

Another way of setting the overflow flag is as follows:

```
*FLAG=FLAG OR 0100H
```

Change 8086/8088 Registers

PSCOPE allows you to display and modify 8086/8088 registers.

Syntax

8086-8088-register [= *expression*]

Where:

expression	resolves to a number to be loaded into the specified 8086/8088 register.
8086/8088-register	is a symbol representing one of the 8086/8088 registers.

8086-8088 Register	Description	Type
AX	Accumulator register	WORD
AH	Accumulator high byte	BYTE
AL	Accumulator low byte	BYTE
BX	B register	WORD
BH	B register high byte	BYTE
BL	B register low byte	BYTE
CX	C register	WORD
CH	C register high byte	BYTE
CL	C register low byte	BYTE
DX	D register	WORD
DH	D register high byte	BYTE
DL	D register low byte	BYTE
CS	Code segment register	WORD
DS	Data segment register	WORD
ES	Extra segment register	WORD
SS	Stack segment register	WORD
SP	Stack pointer	WORD
BP	Base pointer	WORD
IP	Instruction pointer	WORD
DI	Destination index	WORD
SI	Source index	WORD

Description

These pseudo-variables only affect the copy of the 8086/8088 flags used by the program being debugged. These pseudo-variables do not affect the copy of the 8086/8088 flags used by PSCOPE.

Example

The following example illustrates how to load the AX register with 12ABH:

```
*AX=12ABH
*AX
12AB
```

The REGS Command

The REGS command displays the 8086/8088 registers.

Syntax

```
REGS
```

Description

The REGS command displays the registers in hexadecimal regardless of what the current radix is. Flags are identified by mnemonic only when they are set. If no flags are set, the word *none* appears.

The registers displayed are the four data registers (AX, BX, CX, and DX), the four segment registers (CS, DS, SS, and ES), the instruction pointer (IP), the base pointer (BP), the stack pointer (SP), and the two index registers (SI and DI).

Example

To display the 8086/8088 registers enter the following command:

```
*REGS
AX=0004H       BX=0005H       CX=0000H       DX=0002H
CS=5588H       DS=0188H       SS=0104H       ES=0000H
IP=46C7H       BP=0634H       SP=0104H       SI=0830H
DI=03A2H
FLAGS: ZFL PFL
```

Change 8087 Registers

When running under the iRMX-86 operating system, PSCOPE requires that your microcomputer system contain the iSBC 337 MULTIMODULE (the 8087 hardware). The MULTIMODULE resides on the 86/30 card. When loading your program, use the CH8087 option.

When running under the ISIS or iNDX operating systems, PSCOPE requires that your program be linked with the 8087 software emulator. When loading your program, use the E8087 option.

When running under the iRMX-86 operating system, PSCOPE allows you to display and modify 8087 registers.

Syntax

8087-register [=*expression*]

Where:

expression	resolves to a number to be loaded into an 8087 register.
8087-register	is a symbol representing one of the 8087 registers.

8087 Register	Description	Type
STO	Internal stack register 0	TEMPREAL
ST1	Internal stack register 1	TEMPREAL
ST2	Internal stack register 2	TEMPREAL
ST3	Internal stack register 3	TEMPREAL
ST4	Internal stack register 4	TEMPREAL
ST5	Internal stack register 5	TEMPREAL
ST6	Internal stack register 6	TEMPREAL
ST7	Internal stack register 7	TEMPREAL
FSW	Status word	WORD
FCW	Control word	WORD
FIA	Instruction address	DWORD
FDA	Data address	DWORD
FIO	Instruction	WORD

Example

To display the ST4 register enter the following command:

```
*ST4
+2.3596874320856382E+00001
```

Read and Write I/O Ports

This section explains how to read and write the I/O ports. The 8086/8088 I/O space contains byte-wide ports and word-wide ports. The port addresses are from 0000H to FFFFH.

Syntax

$\begin{bmatrix} \text{PORT} \\ \text{WPORT} \end{bmatrix}$ (*expr-for-port-number*)[=*expr*]

Where:

PORT	identifies a byte-wide port.
WPORT	identifies a word-wide port.
expr-for-port-number	represents the port number in the current radix.
expr	represents the data in the current radix. If the value takes up more than one byte, PSCOPE truncates the data to a byte.

Example

The following example writes 10 to the port 100. The H identifies 199 as a hex number. The T identifies the 100 as a decimal number. The value 199H is written to a byte-wide port.

```
*PORT(100T)=10H
```

The following example writes 199 to the port 100. The H identifies the 199 as a hex number. The T identifies the 100 as a decimal number. The value 199H is written to a word-wide port.

```
*WPORT(100T)=199H
```

The following example displays the word-wide port 100. PSCOPE always displays the port value in hexadecimal, regardless of what the current radix is.

```
*PORT(100T)
199
```

Note that accessing ports on the microcomputer system may cause undesirable results. Consult the user documentation for your microcomputer system before you access any of its ports.

Change Name Scope

This section explains how to move the debug cursor to a new location, thus changing the current name scope (CNS).

Syntax

NAMESCOPE [= *expression*]

Where:

expression is the location to which the debug cursor is to be reset.

Description

If you do not specify an *expression*, PSCOPE displays the address of the current name scope. Changing the current name scope affects the set of symbols to which the debugger has access. However, changing the current name scope does not activate symbols that are not already active.

Note that the dollar sign ($) is a predefined symbol equivalent to the current execution point. Entering the following:

```
NAMESCOPE = $
```

returns the current name scope to the current execution point.

Resuming program execution (using GO, LSTEP, ISTEP, or PSTEP) automatically resets the debug cursor to the current execution point.

Example

The following example illustrates how changing the current name scope affects the lookup of a partially qualified reference to the procedure DIGIT in the sample program DC (found in Appendix E):

```
*get_token.digit
1C50H:02D2H
*digit
DIGIT
ERROR #12: Symbol not known in current context.
*namescope = get_token
*digit
1C50H:02D2H
```

Active Function

The ACTIVE function determines if a program object is active at the point where execution was suspended. ACTIVE will tell you if a stack-based variable is currently allocated and accessible.

ACTIVE (*symbolic-reference)*

Where:

symbolic-reference is any program symbol, fully or partially qualified.

Description

ACTIVE is a Boolean function which indicates whether a *symbolic-reference* refers to a program object that can be displayed or modified at the current execution point. Statically allocated variables are always active. Dynamically allocated (stack-based) variables are active if they are available in the current (top) stack frame. ACTIVE returns a TRUE if *symbolic-reference* is active and a FALSE if it is not.

Example

```
*ACTIVE (:dc.factor.expression_value)
TRUE
*if ACTIVE(op) then write 'op = ',op
.* else write '-op not active-'
.*endif
-op not active-
```

Display Memory

This section explains how to display the contents of one or more memory locations.

Syntax

memory-type start-address [length-specifier]

Where:

memory-type	is one of the predefined memory types listed in Table 5-1.
start-address	is the address of the first location in memory to be displayed, expressed with an expression. Recall that you can obtain the address of a symbol by prefixing the symbol with the dot operator.
length-specifier	is one of the following:

LENGTH *expression*	specifies the number of adjacent objects of the specified memory type to be accessed.
TO *end-address*	specifies the last address of a range of memory to be displayed.

Description

You can specify portions of memory you want displayed using *memory-type* (indicates the memory type to be used in displaying that portion of memory). The memory address is displayed first (as a pointer value), followed by the value(s) at that location.

Example

The following example illustrates using the display memory command.

```
*char .buffer.str[1] to .buffer.str[buffer."length]
1CCBH:0018H '1+2+3+4+5+'
```

The following example references the CHAR variable c, (in the sample program DC found in Appendix E) as a BYTE. Displaying memory as BYTE values also displays them as CHARs, as shown here:

```
*byte .c
1CCBH:0074H     97                                     'a'
```

Modify Memory

This section explains how to modify the contents of one or more memory locations.

Syntax

mem-type start-addr [length-specifier] = chg-value

Where:

mem-type	is one of the memory types shown in Table 5-1.
start-addr	is the first memory address to be displayed.

length-specifier is one of the following:

LENGTH *expression*	specifies the number of adjacent objects of the specified memory type to be displayed.
TO *end-addr*	is the last address of a range of memory to be displayed.

chg-value is the new value to which the contents of the specified memory locations are to be set; *chg-value* is one of the following:

expression [,*expression*]*

mem-type start-addr [*length-specifier*]

Description

The change value must be of a type that either matches the memory type or can be forced to match under the type coercion rules given in Chapter 3. The debugger displays an error message if the change value is the wrong type.

Example

The following example initializes the first five values in the array "variable_table":

```
*integer .variable_table = 2,4,6,8,10
```

The following example initializes the entire contents of the array variable_table to 0 and 1, with 0 in the first location and every other location thereafter, and 1 in the second location and every other location thereafter:

```
*integer .variable_table length 26 = 0,1
```

The following example uses the 8086 registers to reference the WORD value on top of the 8086 stack and set it to 0:

```
*word ss:sp = 0
```

The Single Line Assembler/Disassembler

You can modify and display memory as 8086/8088/8087 mnemonics.

Syntax

Assignment

$\begin{bmatrix}\text{SASM}\\ \text{ASM}\end{bmatrix}$ *address* = *'assembler-mnemonic'* [,*'assembler-mnemonic'*]*

Display

ASM *start-address* [*length-specifier*]

Where:

SASM and ASM	are keywords identifying the single-line assembler. SASM and ASM are equivalent. SASM is included to make the PSCOPE syntax familiar to I^{2}ICE^{TM} users.
start-address	is the address of the first location in memory to be displayed.
length-specifier	is one of the following:
LENGTH *expression*	specifies the number of instructions to disassemble.
TO *end-address*	specifies the last address of a range of memory to be displayed.
address	is a single address or an expression that resolves to a single address.
assembler-mnemonic	is an 8086/8088/8087 instruction.

Description

With the disassembler, you can display memory locations as 8086/8088/8087 mnemonics.

With the assembler, you can load memory locations with 8086/8088/8087 instructions.

The Disassembler

PSCOPE interprets the *start-address* as the beginning of an assembly language instruction. If you give a *start-address* that is not the beginning of an instruction, PSCOPE still interprets the memory location as an instruction.

A single address displays the instruction beginning at that address. A range of addresses (*start-address* TO *end-address*) displays all instructions that start within that range. To specify an exact number of instructions to be displayed, use the form, *start-address* LENGTH *number-of-instructions.*

The following example displays the assembly language instructions making up statement #16.

```
*ASM :pager#16 TO :pager#17
063CH:01F9H FF361C00    PUSH ES:WORD PTR [SI]
063CH:01FDH 9A00000000  CAll 0:0
063CH:0202H 3C00        CMP  AL,0H
063CH:0204H 7403        JE   $+5H
063CH:0206H E95401      JMP  $+0000AH
063CH:0209H FF361400    PUSH ES:WORD PTR [SI]
```

Interpret the display this way. The first entry is the starting address where the instruction resides; the second entry is the hexadecimal representation of the instruction (displayed in hex, regardless of what the current radix is); the third entry is the mnemonic representation of the instruction.

If you did not specify a partition, you would only get the assembly language instruction at the start address, as in the following example:

```
*ASM :pager#16
01F9  FF361C00     PUSH ES:WORD PTR [SI]
```

The Single Line Assembler (SLA)

You specify the instructions with assembler mnemonics. The SLA does not accept all the possible forms of the 8086 instructions (as described in the Assembler Operators section).

Assembler Directives. The SLA does not support assembler directives. For example, you cannot replace *assembler-mnemonic* with MY_VAR DB ?. What you put in for *assembler-mnemonic* must actually generate code.

Assembler Operators. The SLA does not recognize all the possible assembler operators. Consider the instruction MOV AL,BYTE PTR [BX]. This instruction would be an incorrect form for the SLA because the SLA does not recognize PTR. You can still put that instruction into memory with the SLA, but you must code it as MOV AL,BYTE [BX]. In some cases, what is a correct form for the SLA is an incorrect form for ASM-86.

The assembler type operators recognized by the SLA are the following:

BYTE	Specifies a number that takes up one byte. The corresponding PSCOPE memory type is BYTE.
DWORD	Specifies a number that takes up four bytes. The corresponding PSCOPE memory type is POINTER.
FAR	Specifies that both the CS and the IP take part in a JMP or CALL.
QWORD	Specifies a number that takes up eight bytes. The corresponding PSCOPE memory type is LONGREAL.
TBYTE	Specifies a number that takes up 10 bytes. The corresponding PSCOPE memory type is TEMPREAL.
WORD	Specifies a 16-bit unsigned number. The corresponding PSCOPE memory type is WORD.
segment override prefixes	Specifies that an operand is to be taken from a non-default segment.

Absolute Addresses. Unlike ASM-86, the SLA allows you to specify an absolute address within an instruction. For example, the SLA recognizes the instruction JMP 12:34. This instruction is a direct-far jump. ASM-86 would require that you use a label or jump indirectly through a register.

Like ASM-86, the SLA accepts a symbol, but the SLA requires a fully-qualified symbolic reference. For example, to jump to a label within the same module and procedure, you must code JMP :mod.proc.label. To load BX with a program variable, code MOV BX, .:mod.proc.var.

Remember that PSCOPE itself consists of load-time-locatable code. So does the program that you are debugging. The result is that the absolute addresses corresponding to program locations may differ from application to application. Be careful only to change instructions that exist in locations assigned to your program. Do not change instructions in any locations used by PSCOPE. The consequence of this restriction is that the SLA is good for replacement patches, not for insertion patches.

Also, when constructing a replacement patch, be careful not to leave part of an instruction hanging. You may need to pad your replacement with NOPs.

Jumps and Calls. The SLA expects the control transfer instruction to obey slightly different mnemonic conventions than ASM-86. Consider the five kinds of jumps: direct-short, direct-near, indirect-near, direct-far, and indirect-far.

The SLA does not produce a direct-short jump. This is not a problem because you can use a direct-near jump instead.

The *direct-near jump* consists of three bytes. The first byte is E9, the opcode. The next two bytes are the difference between the current location and the destination.

The SLA uses an absolute address as the operand for a direct-near jump. With this feature, you always know the destination of the jump without having to compute the relative offset from the current IP. For example, to load the absolute address 1BF00H with a direct-near instruction that jumps to absolute address 1BF05H, enter the following:

```
*SASM 1BF00H='JMP 1BF05H'
01BF00    E90200
```

The relative displacement from the IP is 0002. This instruction skips two bytes. To load absolute address 1BF00H with a direct-near instruction that jumps to absolute address 1BFFCH, enter the following:

```
*SASM 1BF00H='JMP 1BFFCH'
01BF00    E9F9FF
```

The relative displacement from the IP is FFF9H. This is a -7 in 2's complement notation.

The *indirect-near jump* consists of two bytes and possibly a 16-bit displacement. The first byte is the opcode FF, and the second byte contains the MOD field, the R/M field, and three more bits of the opcode (100Y). For example, to load absolute address 1BF00H with an instruction that jumps to the offset contained in BX (the instruction assumes the current CS), enter the following:

```
*ASM 1BF00H='JMP BX'
01BF00H    FFE3
```

The previous example used the keyword ASM instead of SASM. ASM is an alternate form of SASM.

You can get another level of indirection by using brackets around the register name. For example, to load absolute address 1BF00H with an instruction that jumps to the offset (assuming current CS) that is stored in the memory location whose offset is in BX (assuming the current DS), enter the following:

```
*SASM 1BF00H='JMP [BX]'
01BF00H     FF27
```

The *direct-far jump* consists of five bytes. The first byte is the opcode EA, and the last four bytes contain the offset and the selector of the target instruction. The SLA recognizes a direct-far jump by the FAR operator. For example, to load location 3:300H with an instruction that jumps to location 12:34, enter the following:

```
*ASM 3:300H='JMP FAR 12:34'
0003:0300H     EA34001200
```

If you leave out the selector of the target address, the SLA assumes zero. For example, JMP FAR 34H transfers control to the location 00:34. If you specify an address that takes up more than 16 bits, the extra upper bits are ignored. For example, JMP FAR 12345H transfers control to the location 00:2345H.

The *indirect-far jump* consists of two bytes and possibly a 16-bit displacement. The first byte is the opcode FF. The second byte contains the MOD field, the R/M field, and three more bits of the opcode (101Y). For example, to load absolute address 1BF00H with an instruction that jumps to the selector and the offset (assuming the current CS) stored in the memory location whose offset is in BX (assuming the current DS), enter the following:

```
*SASM 1BF00H='JMP DWORD [BX]'
01BF00H     FF2F
```

The SLA mnemonic conventions are similar for the CALL instruction.

RETFAR

To return from a far jump or a far call, the SLA requires the mnemonic RETFAR. ASM-86 knows whether a procedure is near or far, and consequently, it generates the appropriate return. Because the SLA does not have this information, you must specify whether you want a near return or a far return. With the SLA, specify a near return as RET and a far return as RETFAR. For example, to load absolute address 1BF00H with a far return that discards three words from the stack after returning, enter the following:

```
*SASM 1BF00H='RETFAR 6'
01BF00H     CA0300
```

Symbolic Addresses. The SLA will accept symbolic addresses, but, because the SLA does not use the current NAMESCOPE, you must supply a fully-qualified symbolic reference. For example, the SLA accepts the instruction MOV AX,.:mod.proc.var

The period before the colon is a standard PSCOPE operator. It identifies the symbolic reference as resolving to the address of *var* and not the actual value of *var*.

Multiple Forms of an Instruction. ASM-86 is a versatile assembler that often allows more than one version of the same instruction. For example, ASM-86 has a form of the MOV instruction that moves a byte from AL to a memory location. This form is distinct from the one that moves a byte from a register to a memory location. The general MOV from register to memory does contain the ability to specify AL, but ASM-86 uses the shorter form because AX is the accumulator and hence a preferred register.

The SLA assembles the general form and not the shorter form. For example, consider the instruction MOV SUM,AL. ASM-86 assembles this in three bytes as A200 01H, assuming that 100H is the offset of the program variable *sum.* The SLA requires a fully qualified symbolic reference for *sum* and assembles the same instruction in four bytes as 8806 0001H.

Indirect Addressing. ASM-86 allows you to express an indirect address in many different ways. For example, with ASM-86, the following instructions assemble to the same value.

```
MOV AX,[BX+DI+2]
MOV AX,[BX][DI][2]
MOV AX,[BX][DI]+2
```

The SLA only accepts the last form. The following is the general form for an indirect address accepted by the SLA.

symbol[basereg][indexreg]+offset

All the parts are optional. The brackets are part of the syntax and are required when you choose the option. You must choose an option in the order given. For example, if you construct an indirect address with a base register (BX or BP) and an index register (SI or DI), the base register must precede the index register. For example, to load offset 21:3CH with an instruction that moves the contents of the AX register to memory through an indirect address, enter the following:

```
*SASM 21:3C='MOV .:cmaker.purchase[BX][SI]+300H,AX'
0021:003CH     89801003
```

This instruction loads a memory location with the contents of AX. It forms the address of the memory location in the following way:

1. Adds 300H to the offset of the address of the program variable *purchase* in the module *cmaker.* The address of *purchase* is 44:10H.

   ```
   *.:cmaker.purchase
   0044:0010H
   *DS
   0044
   ```

2. At runtime, adds the contents of BX, the contents of SI, and the sum from step 1. This value is the final offset.

3. Assumes the data segment. Gets the selector value from the DS register. Constructs the physical address and loads the contents of AX into the addressed memory location.

The Default Radix. The SLA assumes the PSCOPE current radix. You can override the current radix for an individual number by appending a letter to the number. The SLA interprets a number as binary if you append a Y, as decimal if you append a T, as hexadecimal if you append an H, and as a multiple of 1024 (decimal) if you append a K. PSCOPE always considers line numbers to be decimal regardless of what the current radix is. The SLA always displays the assembled instruction in hexadecimal regardless of what the current radix is.

String Moves. ASM-86 provides the MOVS, MOVSB, and the MOVSW mnemonics. You must have previously loaded the SI and DI registers. The mnemonic MOVS requires two operands: the name of the destination string (the symbolic name for the first location) and the name of the source string. ASM-86 uses these operands to determine whether you are moving bytes or words. The mnemonics MOVSB and MOVSW do not require operands because the B and W identify whether you are moving bytes or words.

The SLA only accepts the MOVSB and MOVSW mnemonics.

8087 Instructions. The SLA handles 8087 instructions differently than ASM-86. There are four areas of difference, as follows:

1. The SLA represents a stack register without the syntactic parentheses.
2. The SLA does not recognize the ESC mnemonic.
3. The SLA is consistent in its treatment of the 8087 no-wait mnemonics.
4. The SLA does not recognize FWAIT. (FWAIT is an alternate way of specifying WAIT, which the SLA does recognize.)

The Stack Registers. The SLA expects the 8087 stack registers to be ST0 through ST7 rather than ST(0) through ST(7). ASM-86 accepts ST as a symbol for the top of the stack. The SLA does not recognize ST; you must code ST0.

The ESC Mnemonic. The SLA supports all of the 8087 mnemonics except the ESC mnemonic. For example, with the SLA, you can code FADD ST0,ST1. This corresponds to FADD ST(0),ST(1) for ASM-86. ASM-86, but not the SLA, accepts ESC 18H,1 as well. ASM-86 would accept both forms of the instruction and load memory with the word, D8C1H.

The No-Wait Mnemonics. ASM-86 inserts a WAIT instruction before the 8087 instruction unless you insert an N as the second character in the 8087 mnemonic. For example, FDISI is preceded by a WAIT; FNDISI is not preceded by a WAIT. There is one exception. The 8087 instruction FNOP is a no-operation that does generate a wait.

The SLA, however, is consistent when it interprets the second character of the 8087 mnemonic. FNOP does not generate a WAIT; FOP does generate a WAIT. ASM-86 does not recognize FOP.

In addition, ASM-86 does not allow some 8087 instructions to have the no-wait form. The SLA always accepts a no-wait mnemonic.

FWAIT. FWAIT is actually not an 8087 instruction. It is an alternate form of the CPU instruction WAIT. The SLA considers one form to be sufficient.

CHAPTER 6
CONTROL CONSTRUCTS

This chapter describes the four control constructs used in PSCOPE's command language; REPEAT, COUNT, IF, and DO.

The IF construct conditionally executes commands. The REPEAT and COUNT constructs repeat a sequence of debugger commands under the control of a variety of exit conditions. The DO construct groups multiple commands and treats them as a single command.

The level to which you can nest REPEAT, COUNT, IF, and DO control constructs depends upon the amount of workspace available to the debugger.

After you enter the first line of a compound command, each subsequent line displays a prompt preceded by a dot (.). The dot indicates that the line is inside a compound construct. The number of dots preceding the prompt indicates the current nesting level.

The REPEAT and COUNT Constructs

The REPEAT and COUNT constructs let you repeat a sequence of debugger commands controlled by any number of exit conditions.

Syntax

```
REPEAT
    [loop-item]*
ENDREPEAT
```

or

```
COUNT expression
    [loop-item]*
ENDCOUNT
```

Where:

loop-item	is any of the following:

command	is any debugger command.
WHILE *expression*	
UNTIL *expression*	

expression	is any expression that can be forced to a Boolean value.

Description

The *loop-item* of the REPEAT command executes until an UNTIL *expression* evaluates to TRUE or until a WHILE *expression* evaluates to FALSE.

The COUNT command is evaluated similarly, but the number of times that the loop body is executed is bound by COUNT *expression*. COUNT *expression* is evaluated only once, when the command is first encountered.

If you prefer, you can use END in place of ENDREPEAT or ENDCOUNT.

Example

The following example uses REPEAT to implement a form of data breakpoint:

```
*repeat
.*lstep
.*until :dc.c == '+'
.endrepeat
```

The following example steps through 10 statements and then stops:

```
*COUNT 10
.*LSTEP
.*END
```

The IF Construct

The IF construct lets you conditionally execute commands.

Syntax

```
IF expression THEN
     [command]*
[ORIF expression THEN
     [command]*]*
[ELSE
     [command]*
ENDIF
```

Where:

expression	is any expression that evaluates to a Boolean value.
command	is any PSCOPE command.

Description

The IF construct contains an IF clause, any number of ORIF clauses, an optional ELSE clause, and a closing ENDIF.

PSCOPE evaluates IFs as follows:

- If the IF *expression* evaluates to TRUE, PSCOPE executes the command list following the IF *expression* up to the first ORIF clause, the ELSE clause, or the ENDIF.

- If the IF *expression* is FALSE, PSCOPE evaluates the subsequent ORIF clauses in order until it finds an ORIF *expression* that is TRUE, in which case it executes the ensuing command list up to the next ORIF clause, ELSE clause, or ENDIF.

- If the IF clause is FALSE and there are no TRUE ORIF clauses (or no ORIF clauses at all), PSCOPE executes the ELSE clause (if present) up to the ENDIF.
- If the IF clause and no ORIF clauses are TRUE and no ELSE clause is present, PSCOPE resumes execution with the first executable statement following the IF construct.

You may use END, if you prefer, in place of ENDIF.

Example

The following example illustrates using the IF construct.

```
*IF 1 = = 2 THEN
.* 'THIS SHOULDN"T BE PRINTED'
.*ORIF 2 = = 2 THEN
.* 'THIS SHOULD BE PRINTED'
.*ELSE
.* THIS ALSO SHOULDN"T BE PRINTED'
.*END
THIS SHOULD BE PRINTED
```

The DO Construct

The DO construct lets you group commands.

Syntax

```
DO
    [command]*
END
```

Where:

command	is any PSCOPE command (with a few restrictions, like LOAD and INCLUDE).

Description

Debugger objects defined in a DO block (using the DEFINE command described in Chapter 7) are local to that block and supercede any previously defined debug symbols with the same names. You can define global debug symbols within a DO block by using the GLOBAL option on the DEFINE command.

You can nest DO blocks; however, each DO must have a corresponding END. A DO block is not complete (and is not executed) until PSCOPE reaches its matching END.

CHAPTER 7 DEBUG OBJECT MANIPULATION COMMANDS

Debug objects, which you define during the debugging session, are symbolic entities similar to the variables and procedures in your program. Debug objects can be variables of any memory type, command abbreviations, debug procedures, code patches, or a collection of breakpoints and tracepoints.

Note that PSCOPE uses the same commands to display, modify, and obtain a directory of program objects and debug objects. Program objects are part of a program; they are accessible to PSCOPE when the program is loaded and are inaccessible when a different program is loaded. Debug objects are not tied to any particular program. PSCOPE has explicit commands to define debug objects, to remove them, and to save their definitions.

This chapter describes the following commands which are used to manipulate debug objects:

Define debug object
Display debug object
Modify debug object
Remove debug object
Put/append debug objects

Debug Objects

Like objects in a program, debug objects have a name and a type. The DEFINE command which creates the debug object specifies the name and type. Debug objects are either global or local.

Global debug objects exist from the time you create them until you remove them with the REMOVE command. (The LOAD command (described in Chapter 4) implicitly removes some global debug objects.) Local objects can exist only within PSCOPE's DO blocks. PSCOPE automatically removes them when control passes out of the DO block within which they were defined.

Like keywords, debug object names have precedence over program symbol names. Thus, to access a program symbol with the same name as a debug object, you must prefix the program symbol with quotation marks ("). ("Referencing Names" in Chapter 3 discusses how to do this.)

Some debug object names also have precedence over other debug object names. Local debug objects have precedence over global debug objects of the same name. Likewise, the most recently defined local debug object takes precedence over other local debug objects with the same name.

You must remove a global debug object's name (with the REMOVE command, described later in this chapter) before you can redefine the global debug object to be a different type. You can define a local debug object with the same name as a global debug object without affecting the definition or value of the global debug object.

You cannot find the address of a debug object using the dot operator (.) explained in Chapter 3. PSCOPE displays an error message if you try.

Debug objects can have any of the memory types or debug types specified in Table 3-5. The properties of the debug objects depend on their type.

Memory Type Debug Objects

In general, you can use a debug object defined as a memory type (BYTE, WORD, INTEGER, REAL, etc.), like a program variable of that type (in expressions, to display and modify). Hence, a memory type debug object is a debug variable.

Debug variables of type CHAR have more capabilities than CHAR variables in a Pascal-86 program. Debug variables can be assigned string values from 0 to 254 characters, including the results the PSCOPE built-in string functions CONCAT and SUBSTR return.

Debug Type Debug Objects

A debug object defined to have one of the five debug types listed in Table 3-5 has different properties from those of program objects.

LITERALLYs are string-replacement macros. When PSCOPE finds a symbol that is a LITERALLY name, it replaces the LITERALLY name with the string value associated with that name (just as in PL/M). LITERALLYs provide a convenient way to abbreviate commands and keywords.

The debug type debug objects are as follows:

Type	Description	Reference
PROC	Debug procedure	Chapter 8
BRKREG	Group of breakpoints	Chapter 11
TRCREG	Group of tracepoints	Chapter 11
PATCH	Debug patch code	Chapter 9

You must redefine debug type objects in order to change their definitions. Use the EDIT command to recall the previous definition of a debug type object in order to redefine it.

The DEFINE Command

The DEFINE command creates a debug object.

Syntax

DEFINE [GLOBAL] *type symbol-name* [= *value*]

Where:

GLOBAL	indicates that the debug object will be global. Debug type debug objects are always global; the GLOBAL option is not allowed for these objects. Memory type debug objects are global unless they are defined inside a PSCOPE DO block. Hence, use the GLOBAL option for memory type objects inside a DO block that you do not want to be automatically removed when control passes out of that block.
type	is any of the memory types or debug types shown in Table 3-5.

symbol-name	is any name other than a PSCOPE keyword (for local objects) or a keyword or existing debug object name (for global debug objects). The name can be up to 254 characters long; the first 40 characters must be a unique combination.
value	is the value to be assigned to the debug object. The *value* can be the result of evaluating an expression. PSCOPE assigns a null value if you do not give a value. The *value* is required for debug type debug objects. The *value* is optional for memory type debug objects.

Description

The DEFINE command creates a debug object with the name, type, and value you specify. If the object being defined is a memory type, the debug object has the same properties as a program variable of the same type. Memory type debug objects can be any of the object types listed in Table 3-5, but cannot be a user-defined type. You cannot use the same name for two different debug objects unless a debug object is defined locally within a DO block. When PSCOPE exits the block, PSCOPE automatically removes the local debug object. You can then assign the name to another object of a different type for use in another block.

The DEFINE command lets you optionally assign initial values to the objects being defined.

Example

The following example illustrates memory type debug variables.

```
*define byte num
*define integer i = 13
*do
.*define word local_word = 2 * i
.*end
*define char char_1 = 'this is a string'
```

The following example illustrates LITERALLYs.

```
define literally lit = 'literally'
define lit       def = 'define'
def    lit       el = 'eval $ line'
```

The DISPLAY Command

The DISPLAY command displays the values of debug objects.

Syntax

[*type*] *symbol-name*

Where:

type	defines the debug symbol. If you specify *type*, PSCOPE displays the definition of the debug symbol. If you omit *type*, PSCOPE expands the symbol (for LITERALLYs) or executes the symbol (for PROCs). You must enter *type* to display any of the following: • LITERALLY • PROC • PATCH • BRKREG/TRCREG
symbol-name	is the name of a previously defined debug object.

Description

For program symbols and memory-type debug variables, you can access the value of a symbol by entering the symbol name. PSCOPE displays the value of the symbol on the following line. Similarly, you can access the value in an expression by entering the name of the symbol.

For debug objects of any of the debug types, you can access the definition of the debug object by entering the type, followed by the symbol name. PSCOPE displays the definition of the named object.

Example

Suppose you defined a memory type debug variable as follows:

```
*define word w1 = 400
```

Entering the symbol name yields the following value:

```
*w1
400
```

However, a debug type debug object functions differently. Consider the following LITERALLY definition:

```
*define literally w = 'write'
```

Entering w by itself automatically expands the LITERALLY, as though you entered write. You can display w definitions by preceding the w with its type, as follows:

```
*literally w
```

PSCOPE responds by printing the following:

```
define literally W = 'write'
```

which is the definition of w.

The MODIFY Command

The MODIFY command modifies the previously defined value of a memory type debug symbol.

Syntax

name = *value*

Where:

name	is the name of a memory type variable.
value	is the new value to be assigned to the debug symbol.

Description

The type of *value* must be the same as *name*'s type, or you must be able to force that type (using the type coercion rules described in Chapter 3). PSCOPE displays an error message if the change is not possible. Note that the MODIFY command for debug memory type variables has the same syntax as the MODIFY command for program symbols.

The MODIFY command works only with memory type objects. You must redefine a debug type object in order to modify it.

Example

The following example shows the operation of the MODIFY command:

```
*define integer i = -150
*i
-150
*i = -2 * i
*i
+300
```

The REMOVE Command

The REMOVE command deletes one or more debug symbols from the debug symbol table by symbol name and object type (or by symbol name or object type).

Syntax

REMOVE *remove-list*

Where:

remove-list is one of the following:

DEBUG

remove-item [, *remove-item*]*

remove-item is one of the following:

memory-type	is one of the memory types given in Table 3-5. PSCOPE deletes all debug symbols of the specified type.
debug-type	is one of the debug-types given in Table 3-5. PSCOPE deletes debug symbols of the specified type.
PATCH *expression*	is an expression yielding an address where you set a patch. PSCOPE removes the patch at this location.
symbol-name	is the name of a debug symbol of any memory type or debug type (except patch). PSCOPE deletes the specified symbol.

Description

The REMOVE command deletes global debug objects. The user specifies the debug type or the memory type object (or a list of objects) to be deleted by type and name (or by type or name). Specifying DEBUG instead of a list of types and names (or a list of types or names) removes all debug objects.

Do not use the REMOVE command to delete local debug objects; PSCOPE automatically deletes them once control passes out of the PSCOPE DO block in which they are defined.

Example

The following examples use the REMOVE command.

```
*remove proc          /* Remove all PROCs */
*remove i             /* Remove the single object i */
*remove debug         /* Remove all debug symbols */
*remove proc, i       /* Remove all PROCs and i */
```

The PUT and APPEND Commands

The PUT and APPEND commands save the definitions of the debug objects in a disk file.

Syntax

PUT *pathname put-list*

APPEND *pathname put-list*

Where:

pathname	is the path name which identifies a file (or any output device) to which you want to send the text containing the debug object definitions.

put-list is one of the following:

DEBUG

put-item [, *put-item*]*

put-item is one of the following:

memory-type	is one of the memory types given in Table 3-5. PSCOPE saves the definitions of all debug symbols of the specified type in the specified file.
debug type	is one of the debug types given in Table 3-5. PSCOPE saves the definitions of all debug symbols of the specified type in the specified file.
PATCH *expression*	is an expression yielding an address where you set a patch. PSCOPE saves the patch at this address in the specified file.
symbol-name	is the name of a debug symbol of any memory type or debug type (except PATCH). PSCOPE saves the definition of the specified debug symbol in the specified file.

Description

The PUT and APPEND commands place definitions of the specified objects in the selected disk file (or the specified output device). (You can retrieve the definitions with the INCLUDE command described in Chapter 10.)

PUT creates a new file to contain the specified definitions, unless a file of that name already exists. In that case, PUT replaces the old file with a new file containing the definitions.

APPEND adds onto the end of an existing file or creates a new file if the specified file does not already exist.

For debug type debug objects, the entire definition text (including the value portion) is placed in the specified file. For memory type debug variables, only the type and name are included in the object definition because memory type debug objects can be easily changed using the modify command, whereas debug type debug objects must be redefined.

Note that except for the file specification, the formats of the PUT and APPEND commands are identical to the REMOVE command's format.

Example

The following example illustrates using the PUT command.

```
*define literally lit = 'literally'
*define lit def = 'define'
*def lit stacktop = 'word ss:sp'
*def integer i = 13
*def word j = 100
*put defs.mac debug
```

The file DEFS.MAC contains the following:

```
define literally lit = 'literally'
define literally def = 'define'
def literally stacktop = 'word ss:sp'
define integer i
define word j
```

Note that the values of the memory type variables i and j are not saved.

The following PUT command:

```
put defs2.mac lit, def, integer
```

places the following text into the file DEFS2.MAC:

```
define literally lit = 'literally'
define literally def = 'define'
define integer i
```

Note that in this example, lit is not expanded to literally when it appears in the PUT command. PSCOPE treats lit as the name of a debug object that happens to be a LITERALLY. The same is true for the REMOVE command and, in fact, for all commands that let you specify a debug object by name.

CHAPTER 8 DEBUG PROCEDURES

PSCOPE lets you define debug procedures to expand the debugger command language and to aid development of the program you are debugging. Debug procedures are one of PSCOPE's most powerful features.

You can use debug procedures (whose type is PROC) to automate the software test process, set up breakpoints based on data values or Boolean conditions, and put together complex commands using PSCOPE's command language. Debug procedures let you use parameters (LITERALLYS do not).

This chapter explains debug procedures basics: how to define them, return values from them, make calls to them, and remove them.

Define Debug Procedure

The DEFINE command defines a debug procedure.

Syntax

DEFINE PROC *name* = *command*

Where:

name	is any name except a reserved keyword. The *name* can be up to 254 characters long, of which the first 40 must be a unique combination.
command	is a PSCOPE command.

Description

A debug procedure contains a single PSCOPE command. This command is usually a DO construct, which allows multiple commands and the declaration of local variables.

A debug procedure does not execute when it is defined, only when it is called. PSCOPE checks syntax when you define the debug procedure. However, if you define a debug procedure within another debug procedure, PSCOPE does not define the inner debug procedure until the enclosing debug procedure is called. Note that while you can define a debug procedure within another debug procedure, all debug procedures are global.

PSCOPE determines the types of all objects in the debug procedure when you define the debug procedure. Changing the type and the definition (or the type or the definition) of an object referenced in a debug procedure before you execute it can cause errors when you run the debug procedure.

Referencing actual and formal parameters when you execute a debug procedure is described later in this chapter.

Debug Procedure Calls

You can call debug procedures using the syntax shown in the following section.

Syntax

name [(*expr*[,*expr*]*)]

Where:

name	is the name of the debug procedure.
expr	is any expression yielding a numeric or string value that is to be passed as an actual parameter in the debug procedure.

Description

PSCOPE executes the command specified in the debug procedure definition. PSCOPE substitutes the values of *expr* for the actual parameter specifier during the execution of the command.

You can call a debug procedure in the following three ways:

- In response to a PSCOPE prompt.
- As an operand of an expression.
- Upon reaching a breakpoint or tracepoint that uses the CALL option (as explained in Chapter 11).

Debug Procedure Return

You can return a value from a debug procedure by placing the RETURN command in the debug procedure.

Syntax

RETURN [*expr*]

Where:

expr	is any expression.

Description

PSCOPE returns the null value if you do not specify *expr*. This can cause a type conversion error if you use the debug procedure as a function.

PSCOPE displays the *expr* value if you use a RETURN command outside a debug procedure. PSCOPE displays an error message if it expects a return value (such as when the debug procedure is used in an expression) but no RETURN is executed.

Return values are required from debug procedures used as operands in expressions or automatically called upon reaching a breakpoint or a tracepoint (but not from debug procedures called in response to a PSCOPE prompt).

Accessing Debug Procedure Parameters

You can reference the values of the parameters passed to the debug procedure when it executes.

Syntax

% *parameter*

Where:

parameter is one of the following:

integer-constant	is an arbitrary unsigned integer constant specifying which parameter you desire. Note that %0 specifies the first parameter in the list of parameters passed to the debug procedure, %1 specifies the second parameter in the list, and so forth.
(*expr*)	is an expression that specifies the desired parameter.
NP	is the total number of parameters in the parameter list passed to the debug procedure. PSCOPE does not limit the number of parameters that you can pass to a debug procedure.

Description

All parameters are passed by value and are local to the specific execution of the debug procedure to which they are passed. Thus, you can call debug procedures recursively.

A debug procedure cannot assign new values to the parameters passed to it.

PSCOPE displays error messages if you try to access non-existent parameters or try to access parameters when no debug procedure is executing.

Example

The following debug procedure executes a recursive factorial function:

```
*define proc factorial = do
.*if %0 < 2 then return 1
..*else return %0 * factorial(%0 - 1)
..*endif
.*end

*factorial(5)
120
```

The following debug procedure returns the sum of all the parameters passed to it:

```
*define proc sum = do
.*define longint n = 0
.*define integer i = 0
.*count %np
..*n = n + %(i)
..*i = i + 1
..*endcount
.*return n
.*end

*sum(1,2,3,4)
+10

*sum(factorial(3), factorial(4))
+30
```

The following debug procedure lets you trace a byte value every time it is modified in a program:

```
* define proc trace_byte = do
.* define byte current_value = byte %0
.* write 'value =', current_value
.* repeat until $ == %1
..* lstep
..* if current_value <> byte %0 then
...* current_value = byte %0
...* eval $ line
...* write 'value =', current_value
...* endif
..* endrepeat
.* eval $ line
.* end

* trace_byte (.c, get_line)
value = 32
:DC#
```

CHAPTER 9
CODE PATCHES

This chapter shows you how to define, display, and remove code patches from your program.

This chapter describes the following commands:

Define patch
Display patch
Remove patch

Defining a Patch

The DEFINE command is used to create a PATCH.

Syntax

```
DEFINE PATCH addr1 [TIL addr2] = patch-value
```

Where:

addr1 and *addr2*	are expressions which evaluate to a program location (e.g., line, procedure, or label reference, preceded by a module name if necessary).
patch-value	is one of the following:
command	is any PSCOPE command, except LOAD, GO, LSTEP, or PSTEP. Use a compound construct (see Chapter 6) to specify more than one command.
NOP	is a special command which implies that no operation is to be performed in the patch. When used with TIL *addr2*, NOP allows statements in your program to be effectively deleted.

Description

A PSCOPE patch is a PSCOPE command that is executed prior to a statement in your program or instead of a sequence of statements in your program. Like all other PSCOPE commands, patches are interpreted (rather than translated).

Patches are active as soon as you define them and remain active until you remove them. Note that the LOAD command implicitly removes them.

You are allowed only one patch per address; PSCOPE will replace the first patch if you specify a second patch.

If you specify only *addr1*, then program execution resumes at *addr1*. If you specify both *addr1* and *addr2*, then program execution resumes at *addr2*. In either case the patch may have changed the execution point (by reassigning $ or CS:IP), in which case execution resumes at the reassigned location.

PSCOPE executes patches after it handles any breaks or traces at the same location. (Chapter 12 discusses the break and trace commands.)

PSCOPE executes *command* (or NOP) upon reaching but before executing *addr1.*

Be careful not to overlap patches. For example, because the following:

```
DEFINE PATCH #10 TIL #15
```

and the following:

```
DEFINE PATCH #13 TIL #18
```

overlap, PSCOPE ignores part of the second patch. The first patch skips lines 10 through 14 and resumes at line 15; PSCOPE will not see the patch at line 13.

You can stop program execution and set the execution point to the location where the patch exists by pressing CNTL-C while executing the patch.

Example

The following patch inserts a command before statement 10:

```
define patch #10 = write 'x = ',x
```

The following patch skips statement 15:

```
define patch 15 til 16 = NOP
```

Displaying a Patch

This section shows you how to display patches.

Syntax

```
PATCH addr1
```

Where:

addr1 is an expression that evaluates to a location in your program that is the beginning of a patch.

Description

PSCOPE displays the patch that begins at *addr1.*

Since *addr* is an expression, you can use any other expression which evaluates to the same program location to reference a patch. However, do not use a symbol whose name is a constant but whose value changes (e.g., $) as a patch name.

Example

The following example displays a patch in the sample program DC (found in Appendix E):

```
*define patch #41 = write 'enter get_line'
*patch #41
define patch #41 = write 'enter get_line'
```

Removing a Patch

You can delete patches with the REMOVE command.

Syntax

```
REMOVE PATCH [addr]
```

Where:

addr	is the location of the patch you want to remove.

Description

The REMOVE command lets you delete the patch at *addr*. If you do not specify *addr*, PSCOPE deletes all patches. PSCOPE displays an error message if you try to remove a patch which you have not defined.

This syntax is required regardless of which form of the DEFINE command you used to define the patch.

Example

The following example removes the patch at line (or statement) 10:

```
*remove patch #10
```

CHAPTER 10 UTILITY COMMANDS AND STRING FUNCTIONS

PSCOPE furnishes a variety of utility commands. This chapter discusses these commands, which include the following:

- EXIT
- DIR
- CALLSTACK
- HELP
- LIST/NOLIST
- INCLUDE
- EVAL
- BASE
- INPUTMODE
- WRITE

This chapter also describes the following PSCOPE built-in functions:

- SUBSTR
- CONCAT
- STRLEN
- CI
- SELECTOR$OF
- OFFSET$OF

The EXIT Command

The EXIT command ends the debugging session.

Syntax

```
EXIT
```

The EXIT command has no arguments.

Description

The EXIT command automatically closes all open files, prints a termination message, and returns you to the host operating system.

Example

The following example illustrates using the EXIT command.

```
*exit                  /* User ends debug session */
PSCOPE terminated      /* PSCOPE prints termination message */
```

The DIRectory Command

The DIR command displays the names of all objects of a specified type that are found in a specified set of symbols. The set of symbols can be either program symbols or debug symbols.

Syntax

DIR [*directory*] [*type*]

Where:

directory is one of the following:

DEBUG	specifies that the symbols PSCOPE displays come from the set of debug symbols (those that were created with the DEFINE command).
PUBLIC	specifies that the symbols PSCOPE displays are those found in the user program and that only those symbols with the PUBLIC attribute are to be listed.
module-name	specifies that the program symbol table (as opposed to the debug symbol table) is to be used for the directory and that symbols in only the specified module are to be listed.

type is any type (memory, debug, or user). PSCOPE lists only objects of the specified type.

Description

The DIR command displays the names and types of the set of objects that PSCOPE recognizes. You can list either program symbols or debug symbols with the DIR command.

PSCOPE lists all symbols from the specified directory if you do not specify *type*.

If you do not specify *directory*, PSCOPE uses the current module of the user program, unless *type* implies that PSCOPE uses the debug directory.

Example

Suppose that you entered the following commands:

```
define literally lit = 'literally'
define lit def = 'define'
load dc.86
```

The following command lists all debug symbols:

```
*dir debug
DEF ..... literally
LIT ...... literally
```

The following command lists all symbols in the module DC. The indentation within some of the procedures indicates local symbol definitions:

```
*dir :dc
DIR of :DC
PQ_OUTPUT .................... TEXT (file)
PQ_INPUT ..................... TEXT (file)
@1000 ........................ label
@9999 ........................ label
T ............................ TOKEN (record)
C ............................ char
BUFFER ....................... TEXT_BUFFER (record)
VARIABLE_TABLE ............... array of integer
ERROR ........................ procedure
E ............................ ERROR_CLASS (enumeration)
GET_LINE ..................... procedure
GET_TOKEN .................... procedure
    DIGIT .......................... procedure
        C ........................ char
    UPPER_CASE ............. procedure
        C ........................ char
    LOWER_CASE ............. procedure
        C ........................ char
    GET_CHAR ............... procedure
FACTOR .................. procedure
FACTORVALUE ............ integer
EXPRESSION_VALUE ....... integer
VARIABLE_INDEX .......... char
TERM .................... procedure
TERM_VALUE ............. integer
FACTOR_1_VALUE .......... integer
FACTOR_2_VALUE .......... integer
OP ...................... char
EXPRESSION ............. procedure
EXPRESSION_VALUE ....... integer
TERM_1_VALUE ............ integer
TERM_2_VALUE ............ integer
OP ...................... char
STATEMENT ............... procedure
EXPRESSION_VALUE ....... integer
```

In the following example, PSCOPE assumes module DC because the user did not specify a module. Note that additional qualification indicates local (not module level) symbols.

```
*dir procedure
DIR of :DC
ERROR
GET_LINE
GET_TOKEN
GET_TOKEN.DIGIT
GET_TOKEN.UPPER_CASE
GET_TOKEN.LOWER_CASE
GET_TOKEN.GET_CHAR
FACTOR
TERM
EXPRESSION
STATEMENT
```

In the following example, the user does not specify a module, but the type specified is a debug type. Hence, PSCOPE uses the debug symbol table for the directory:

```
*dir literally
DEF
LIT
```

The CALLSTACK Command

The CALLSTACK command displays your program's dynamic calling sequence.

Syntax

```
CALLSTACK[n]
```

Where:

n is an optional integer expression that indicates how much of the call stack you want to see.

Description

Using the CALLSTACK command, you can symbolically display the current chain of procedure calls in your program. In response to this command, PSCOPE prints a sequence of fully qualified references to procedures, one per line. The reference listed first is the point to which execution control will return when the current procedure returns (its return address). The second entry is the return address for the procedure that called the current procedure, and so on.

The optional expression *n* indicates how much of the call stack you want PSCOPE to display. PSCOPE displays the entire call stack if you do not specify *n*. A positive *n* value indicates that the first *n* entries are to be displayed (the *n* most recent procedure calls). A negative *n* value indicates that the bottom *n* entries of the call stack are to be displayed (the *n* least recent procedures).

Note that the CALLSTACK command works only when the current execution point is inside a module for which PSCOPE has symbol information.

Example

The following example illustrates using the CALLSTACK command.

```
*load dc.86
*go til get_char
[Break at get_char]
*callstack
:DC.GET_TOKEN +323
:DC.FACTOR + 156
:DC.TERM + 15
:DC.EXPRESSION + 37
:DC.STATEMENT + 15
:DC +1787
```

NOTE

The CALLSTACK command does not operate correctly if the nesting sequence includes a procedure written in assembly language.

The CALLSTACK command does not operate correctly if the last executable statement of the main module calls a procedure. The top-level return address must not be within a procedure.

The HELP Command

The HELP command displays explanatory text about various topics, including PSCOPE commands and extended messages for those errors whose primary error message ends with [*].

Syntax

```
HELP [topic]
```

Where:

topic is one of of the following;

topic-name	is the topic name for which you want help information.
E*n*	is the error number for which you want the extended error message. Note that the form E*n* is used even for warnings.

Description

If you do not specify *topic*, PSCOPE lists all topics for which help is available.

Example

The following example shows how the HELP command is used to get information about the BASE command.

```
*help base
====
BASE
====
... (The help information is printed here.)
```

The following example shows what happens when you request HELP on a topic for which there is no HELP information.

```
*help problem
======
PROBLEM
=======
<sorry, but no help is available>
```

The LIST and NOLIST Commands

The LIST command puts all of PSCOPE terminal output into the specified file, and NOLIST closes the file.

Syntax

```
LIST [file-name]

NOLIST
```

Where:

file-name	is the name of the file into which all PSCOPE terminal output is placed.

Description

The LIST command sends all PSCOPE output to the specified file. If you do not specify *file-name*, PSCOPE displays the name of the currently selected LIST file.

NOLIST closes the currently active LIST file (if any). Changing the LIST file closes the old LIST file.

Note that PSCOPE sends only PSCOPE terminal output to the file. PSCOPE does not send any terminal output printed by a user program to the LIST file.

Example

The following example uses the LIST and the NOLIST command.

```
*list exampl.log
*list
exampl.log
*nolist
```

The INCLUDE Command

The INCLUDE command gets input from a file.

Syntax

```
INCLUDE file-name [NOLIST]
```

Where:

file-name	is the name of the file from which input is to be taken.
NOLIST	suppresses echoing of the selected file's input on the screen.

Description

The INCLUDE command takes input from *file-name* until it reaches the end of the file, at which point input continues from the previous source.

INCLUDE commands may be nested. The level of nesting depends upon available memory.

You can enter INCLUDE commands from the terminal; they must be the last command on a line.

Depending on the severity of the error, an error in the INCLUDE file returns execution to the next command or to the standard command level.

Example

The following example illustrates using the INCLUDE command.

```
*include regs.inc
*include file2 nolist      /* Suppress printing of contents */
```

The EVAL Command

The EVAL command has two forms. The first form evaluates expressions and prints the results. The second form displays program locations symbolically.

Syntax

```
EVAL expr [eval-type]
```

Where:

expr is the expression to be evaluated.

eval-type is one of the following three optional evaluation types:

LINE	indicates line number
PROCEDURE	indicates procedure name
SYMBOL	indicates a fully-qualified reference
SYMBOL ALL	indicates all symbolic variables referenced by a particular address

Description

EVAL prints the results of the expression according to the indicated eval-type. EVAL can evaluate an expression in different bases or as a program symbol or line number.

If you do not specify *eval-type*, the value of *expr* is printed in the following manner, depending upon its type:

Type	**Form of EVAL Display**
BYTE BOOLEAN WORD ADDRESS SHORTINT INTEGER LONGINT SELECTOR DWORD	All 3 bases (binary, decimal, hex) and ASCII
POINTER	seg:off (hex) and 20-bit normalized address
REAL LONGREAL TEMPREAL EXTINT BCD	Hexadecimal bytes
CHAR	

Note that PSCOPE prints non-printing ASCII characters as a dot (.).

If you specify *eval-type*, PSCOPE tries to find a program symbol whose address is equal to the value obtained by evaluating the expression.

If *eval-type* is LINE, then PSCOPE displays the following:

:*module-name #line-number* [+ *offset*]

Where:

module-name	is the name of the module in which the address occurs.
line-number	is the nearest line number in that *module-name* to that address.
offset	is the amount by which the address exceeds the exact address of *line-number.*

If *eval-type* is PROCEDURE, then the next message is displayed:

:*module-name* [.*procedure-name*]* [+ *offset*]

Where:

module-name	is the name of the module in which the address occurs.
procedure-name	is the name of the procedure with the closest match to the address.

offset	is the amount by which the address exceeds the exact address of the *procedure-name.*

If *eval-type* is SYMBOL, then the next message is displayed:

fully-qualified-reference [+ *offset*]

Where:

fully-qualified-reference	is a fully qualified reference, such as ds:token_1.m.
offset	is the amount by which the address exceeds the exact address of *fully-qualified-reference.*

Example

The following example uses the EVAL command.

```
*eval $ procedure
*eval $
*eval ds:14h symbol
```

If *eval-type* is SYMBOL ALL, PSCOPE displays all symbolic variables whose starting address is *expr*, that is, PSCOPE supports the FORTRAN EQUIVALENCE and COMMON statements and the PL/M AT attribute. For example, if *expr* is the address of more than one program variable, PSCOPE displays all variables stored at that location.

The BASE Command

The BASE command establishes the default base for numeric constants during input and output.

Syntax

```
BASE [= expr]
```

Where:

expr is an expression that evaluates to 2, 10, or 16 (decimal).

Description

PSCOPE displays the current radix. When the current base is hexadecimal, PSCOPE displays HEX. When the current base is decimal, PSCOPE displays DECIMAL. When the current base is binary, PSCOPE displays BINARY.

You can change the current radix by setting BASE equal to 2T, 10T, or 16T as shown in the following example.

```
*BASE=10H     /*Setting the current radix to hex.*/
*BASE=16T     /*Another way of setting the current radix to hex.*/
*BASE=10000Y  /*Another way of setting the current radix to hex.*/
*BASE
HEX
```

The initial default base is decimal. During input, you can override the default base by putting an explicit base suffix on the constant (for example, 12t).

Note that PSCOPE command processing goes through the following two steps:

- Scanning for syntax errors and deciding what to do.
- Executing the command.

PSCOPE evaluates numeric constants during the first phase of command processing. PSCOPE assigns values to variables during the second phase. Thus, commands such as the following:

```
*base = 10t
*base = 16; VAR1 = 10
```

will give VAR1 the value 10 (decimal), not 16 (decimal), because PSCOPE scans the entire second command line before either of the two commands in the command line are executed. Thus, the numeric constant 10, in the second command line, is interpreted as 10 (decimal). If you want VAR1 interpreted as 16 (decimal), put the expression VAR1 = 10 on a separate line, as shown below:

```
*base = 10t
*base = 16
*VAR1 = 10
```

Example

The following examples use the BASE command.

```
*base = 16t
*base = 0A
```

INPUTMODE

With the INPUTMODE pseudo-variable, you inform PSCOPE what DQ$SPECIAL mode the program being debugged is using. The DQ$SPECIAL mode determines how the operating system interprets console input.

Syntax

INPUTMODE[=*n*]

Where:

n is an expression resolving to 1T, 2T, or 3T.

Default

2

Description

INPUTMODE informs PSCOPE what DQ$SPECIAL mode the program being debugged uses. PSCOPE uses mode 1. Whenever your program reaches a breakpoint, PSCOPE sets DQ$SPECIAL to 1 before delivering the prompt. When your program resumes executing (you enter a GO command or a STEP command), PSCOPE sets DQ$SPECIAL to the value specified by INPUTMODE.

You need to be aware of this if your program sets the DQ$SPECIAL mode to other than the current setting of INPUTMODE. If you break and then resume program execution, your program may be resuming with a different DQ$SPECIAL mode than what you intended.

DQ$SPECIAL is a UDI system call that determines how another UDI system call, the DQ$READ, handles input from the console keyboard.

Input comes from the console keyboard into a system buffer. The DQ$READ transfers the input from the system buffer to a user buffer specified as a DQ$READ parameter. There are three DQ$SPECIAL modes, called 1, 2, and 3. The default is 2, line-editing mode. The following describes modes 1, 2, and 3.

1 Transparent 1 mode. Sometimes this mode is just referred to as transparent mode.

  The DQ$READ specifies a number of characters. When the system buffer contains that number, the DQ$READ transfers those characters to the user buffer. If the system buffer already contains the specified number of characters, the DQ$READ transfers them immediately, and the user program continues. Otherwise, the DQ$READ waits until the system buffer gains the specified number of characters.

  The DQ$READ transfers the characters exactly as you type them. All characters (except CNTL-D and CNTL-C) go into the user buffer. These characters are not echoed to the screen.

2 Line-editing 2 mode. This mode is the default. The operating system allows editing of the console input line. A line terminates with a carriage return. Once you enter the carriage return, you lose the ability to edit that line. The operating system interprets and removes all editing characters (such as backspace) from the input line and adds a linefeed to the final carriage return if the buffer has enough room.

  If the system buffer already contains an edited line, the DQ$READ transfers it immediately, and the user program continues. Otherwise, the DQ$READ waits until the system buffer obtains a carriage return and completes the line editing.

3 Transparent 3 mode. This mode is sometimes called flush mode and sometimes called polling mode.

  The DQ$READ transfers whatever characters are in the system buffer to the user buffer. If the system buffer contains no characters, no characters are transferred. The DQ$READ does not wait for the system buffer to obtain any characters.

  The DQ$READ transfers the characters exactly as you type them. All characters (except CNTL-D and CNTL-C) go into the user buffer. These characters are not echoed to the screen.

Example

Set INPUTMODE to 1 and display it as follows.

```
*INPUTMODE=1
*INPUTMODE
1
```

Because INPUTMODE is a pseudo-variable, you can use it in a debug procedure. For example, the following debug procedure displays yes on the console if INPUTMODE is 1 and no otherwise.

```
*DEFINE PROC checkinput=DO
.*IF INPUTMODE==1 THEN WRITE 'yes'
..*ELSE WRITE 'no'
..*ENDIF
.*END
*checkinput
no
```

The WRITE Command

The WRITE command lets you display and format information at the terminal.

Syntax

WRITE [USING ('[*radix*,] *format-item* [, [*radix*,] *format-item*]*')] *list*

Where:

USING lets you control output using a format string.

radix is one of the following:

- H Set WRITE command display base to hexadecimal.
- T Set WRITE command display base to decimal.
- Y Set WRITE command display base to binary.

format-item is one of the following:

- *n* Decimal number specifying the width of the output field. PSCOPE determines the format of the field by the type of the expression in the argument corresponding to this format item. If n = 0, then PSCOPE uses the normal display length of the item without padding or truncation for the width of the output field.
- *n*C Move output buffer to column *n* (first column is 1).
- *n*X Skip *n* spaces in the output buffer.

Terminates the format string (optional).

>	Terminates the format string and specifies that no carriage return or line feed is to be issued following the WRITE command.
&	Terminates the format string and specifies that the write output buffer is not to be flushed at the end of this WRITE command but is to be added to by later WRITE commands.
"text"	Puts the text between the quotation marks (") into the output buffer.

list uses the following syntax:

```
{ name        }  [ ,name        ]  *
{ expression  }  [ ,expression  ]
{ string-spec }  [ ,string-spec ]
```

Where:

string-spec	is an expression that evaluates to a string.
expression	is an expression whose value you want to display at the terminal.

Description

The WRITE command displays the items in its argument list at your terminal.

In its simplest form, the WRITE command lets you print a list of expressions. PSCOPE prints the value to be printed according to the current output base.

The USING option lets you control output using a format string consisting of format items separated by commas. If you use a format item for one variable, then you must use a format item for each variable in *list*. When text is the only format item, you must use a terminator (., >, or $).

Example

The following example uses the WRITE command.

```
*write 'hello'
hello

*define byte b = 5
*write using ('"b =",0') b
b = 5
```

The String Functions (SUBSTR, CONCAT, STRLEN, and CI)

PSCOPE provides three string manipulation commands: SUBSTR, CONCAT, and STRLEN. In addition, the CI function lets you enter a single character string from the keyboard.

Syntax

SUBSTR (*string-spec, start, length*)

CONCAT (*string-spec* [, *string-spec*]*)

STRLEN (*string-spec*)

CI

Where:

string-spec	is an expression that evaluates to a CHAR value.
start and *length*	are expressions that evaluate to integer values.

Description

The SUBSTR function returns the specified substring starting at *start* and of length *length.* The first character of a string is in position 1. PSCOPE returns the null string if arguments do not make sense (for example, negative length, start past end of string, etc.). If *start* is valid but *length* goes beyond the end of the string, PSCOPE returns the rest of the string beginning at *start.*

The CONCAT function creates a new string by concatenating specified *string-specs.* You can implicitly concatenate string constants (as described in Chapter 3).

The STRLEN function returns the length of its argument string. The length of the null string is zero. You can use the STRLEN function anywhere a number is valid.

The CI function reads one character from the keyboard and returns a string of length one having that character as its value. When CI is referenced in an expression, execution pauses until you enter a character. PSCOPE does not display the entered character on the terminal screen.

Example

The following examples use the CONCAT, SUBSTR, and the STRLEN commands.

```
*define char ch1 = 'The'
*define char ch2 = ' quick'
*define char ch3 = ' brown'
*define char ch4 = ' fox'
*concat (ch1,ch2,ch3,ch4)
The quick brown fox

*SUBSTR (ch3, 3, 3)
row

*STRLEN (CONCAT(ch1, ch4))
7
```

The following example assumes that the character z is entered when execution pauses during the execution of the command:

```
*if ci = 'z' then write 'sleepy?'
.*endif
sleepy?
```

The SELECTOR$OF and OFFSET$OF Functions

PSCOPE provides two functions for extracting the selector (or segment) and offset portions of a pointer value.

Syntax

SELECTOR$OF (*expr*)

OFFSET$OF (*expr*)

Where:

expr is an expression that evaluates to a pointer value.

Description

SELECTOR$OF returns the selector (or segment) portion of a pointer value.

OFFSET$OF returns the offset portion of a pointer value.

The dollar sign ($) in the names of these functions is optional (as in all PSCOPE names) and is included here to improve readability.

Note that these functions correspond to the PL/M V2.0 functions with the same names.

Example

The following examples use the SELECTOR$OF and the OFFSET$OF functions.

```
*base = 16t
*define pointer p = 123:456
*p
0123H:0456H

*selector$of(p)
123

*offset$of(p)
456
```

CHAPTER 11 ADVANCED EXECUTION AND TRACE COMMANDS

This chapter explains how to control and trace program execution. It describes the break and trace registers, as well as how to load and use them. Automatic calling of debug procedures, conditional break and trace, and the break/trace/patch table are covered as well.

Breaking and Tracing

Using procedures, labels, and statements, PSCOPE's breaking and tracing commands let you control and monitor the execution of the program you are debugging.

Breaking and tracing makes use of debugger objects called break registers (BRKREGs) and trace registers (TRCREGs). Breakpoints and tracepoints are defined and stored in these registers and activated with the GO command.

Break Registers (BRKREG)

Break registers are named registers that can hold any number of breakpoints. You can define any number of break registers within PSCOPE's workspace limits.

Placing breakpoints in a named break register lets you easily switch active breakpoints while maintaining control of program execution.

PSCOPE lets you break upon reaching a particular program location, which can be referenced symbolically as a line number, label, or procedure (see Chapters 3 and 5) or as an actual address. In the latter case, PSCOPE assumes that the user entered a valid break location. This location must be on an instruction boundary.

A break occurs when PSCOPE reaches a specified location and before execution of the statement at that location. If you set a breakpoint at a procedure, PSCOPE stops execution at the prologue of the procedure, before processing the declarations for the procedure and before the first executable statement of the procedure.

Since BRKREG is a debug type, the standard debug object manipulation commands described in Chapter 7 apply to BRKREGs. Also, you must enter the value in the definition; it must be a list of location references (line numbers, labels, procedures, or actual addresses) separated by commas as specified in the following syntax section.

The address you specify in a break register can be the address of a high-level language statement or an assembly language instruction.

Syntax

DEFINE BRKREG *name* = *break-item* [, *break-item*]*

Where:

name	is the name of the break register.

break-item is one of the following:

breaks [CALL *proc-name*]

breaks is one of the following:

break-pt [, *break-pt*]*

break-pt can be any expression that evaluates to a location in your program.

proc-name is the name of a debug procedure that returns a value.

Description

You can create a break register with a specified name that contains all the listed *break-pt's* as its breakpoints. Note that PSCOPE associates the breakpoints only with their defined break register. Breakpoints are not active until you specify their break register in a GO command. (The GO command is discussed later in this chapter.)

If you specify the CALL option, PSCOPE associates *proc-name* with the single or parenthesized list of *break-pt's* preceding it.

Note that you cannot modify break registers with a PSCOPE modify command; you must redefine break registers. However, you can add or delete breakpoints from an existing break register by editing the BRKREG definition with PSCOPE's internal editor (discussed in Chapter 2).

After you activate a break register with the GO command, program execution proceeds until PSCOPE encounters one of the breakpoints contained in that register (i.e., program execution reaches that point). Then PSCOPE stops program execution and displays a breakpoint message. If a debug procedure is associated with the breakpoint, PSCOPE automatically executes the debug procedure. PSCOPE converts the return value from the debug procedure to a Boolean. If the Boolean value is TRUE, PSCOPE breaks and displays a break message, as if it had not called the debug procedure. If the Boolean value is FALSE, PSCOPE continues execution without interruption, as if no breakpoint was there. If there is no return value, PSCOPE detects an error and stops execution. This feature lets you set conditional breakpoints with the decision to break based on any Boolean condition, including program variable values or terminal input (see the CI command in Chapter 10). Note that PSCOPE does not allow parameters on the debug procedure specified in the CALL option.

Break messages have the following form:

[Break at *break-pt*]

Where:

break-pt is the location you specified in the definition of the break register.

Example

Note that all the following examples use the sample program DC (shown in Appendix E).

The following example defines one break register containing four breakpoints, each at a different procedure in DC:

```
*Define Brkreg break_1 = error, statement, term, factor
```

This break register has one breakpoint, which calls a debug procedure:

```
*Define Brkreg input_check = get_line CALL PROC2
```

The following example defines a break register with four breakpoints, two of which call the debug procedure PR1:

```
*Define Brkreg special = (term, value) CALL PR1,
** :dc#68, :dc + 1741
```

Trace Registers (TRCREG)

Trace registers (TRCREG) are defined and operate almost exactly like break registers (BRKREG). The only difference is that the tracepoints contained in trace registers do not stop program execution; they display trace messages instead.

Trace registers are named registers that can hold any number of tracepoints. You can define any number of trace registers within PSCOPE's workspace limits.

Putting tracepoints into a named trace register lets you easily switch active tracepoints while maintaining control of program execution.

PSCOPE lets you trace upon reaching a particular program statement, label, or procedure.

The trace occurs when PSCOPE reaches a specified location and before execution of the statement at that location. Tracing a procedure stops execution at the prologue of the procedure, before the declarations in the procedure are processed and before the procedure's first executable statement.

Since TRCREG is a debug type, the standard debug object manipulation commands described in Chapter 7 apply to TRCREGs. Also, you must enter the value in the definition; it must be a list of location references (line numbers, labels, procedures, or actual addresses) separated by commas as specified in the following syntax section.

Syntax

DEFINE TRCREG *name* = *trace-item* [, *trace-item*]*

Where:

trace-item is one of the following:

traces [CALL *proc-name*]

traces is one of the following:

trace-pt [, *trace-pt*]*

trace-pt can be any expression that evaluates to a location within the user program.

proc-name is the name of a debug procedure that returns a value.

Description

You can create a trace register with a specified name that contains all the listed *trace-pt's* as its tracepoints. Note that PSCOPE associates the tracepoints only with their defined trace register. Tracepoints are not active until you specify the trace register in a GO command. (The GO command is discussed later in this chapter.)

If you specify the CALL option, PSCOPE associates *proc-name* with the single *trace-pt* or parenthesized list of *trace-pts* preceding it.

Note that you cannot modify trace registers with a PSCOPE modify command; you must redefine trace registers. However, you can add or delete tracepoints from an existing trace register by editing the TRCREG definition with PSCOPE's internal editor (discussed in Chapter 2).

After you activate a trace register with the GO command, program execution proceeds until PSCOPE encounters one of the tracepoints contained in that register (i.e., program execution reaches that point). Then PSCOPE displays a trace message, and program execution continues. If a debug procedure is associated with a tracepoint, PSCOPE automatically executes the debug procedure. PSCOPE converts the return value from the debug procedure to a Boolean. If the Boolean value is TRUE, PSCOPE displays the trace message, as if the debug procedure was called. If the Boolean value is FALSE, PSCOPE continues execution without displaying a message, as if there was no tracepoint. If there is no return value, PSCOPE detects an error but continues program execution. This feature lets you set conditional tracepoints with the decision to trace based on any Boolean condition, including program variable values or terminal input (see the CI command in Chapter 10). Note that PSCOPE does not allow parameters on the debug procedure specified in the CALL option.

Trace messages have the following format:

[At *trace-pt*]

Where:

trace-pt is the location specified in the definition of the trace register.

Example

Note that all the following examples use the sample program DC (found in Appendix E).

The following example defines a trace register containing three tracepoints:

```
*define TRCREG trace_1 = #80, #224, @1000
```

The following trace register contains one tracepoint, which calls a debug procedure:

```
*define trcreg error_check = :dc.error CALL write_message
```

The GO Command

The GO command controls user program execution. It also lets you activate any number of breakpoints or tracepoints.

Syntax

GO [*brk-spec*]*

GO FOREVER

Where:

brk-spec is one of the following:

TIL *break-pt* [, *break-pt*]*

USING *reg-item* [, *reg-item*]*

reg-item is one of the following:

break-register	specifies a previously defined break register.
trace-register	specifies a previously defined trace register.
BRKREG	
TRCREG	

Description

The GO command starts executing your program from the current execution point ($). The LOAD command sets the initial value of $.

If you specify FOREVER, PSCOPE starts executing without any breakpoints. Note that you can use CNTL-C to interrupt execution, but execution may stop in a location for which PSCOPE has no symbol information (for example, inside UDI, the universal development interface).

If you do not specify *brk-spec*, PSCOPE resumes execution with the same set of break and tracepoints that the last GO command used (except for any break registers or trace registers that were removed or redefined, in which case they are inactive). If you specify USING, PSCOPE starts program execution using the breakpoints and tracepoints in the break and trace registers specified. If you specify the keywords BRKREG and TRCREG with USING, PSCOPE uses all break registers or trace registers.

If you specify TIL, PSCOPE starts program execution using the points listed. As described in Chapter 4, these may be labels, line numbers, procedures, or actual addresses (in which case PSCOPE assumes that the user entered a valid break address).

You can specify any number of TIL and USING clauses. The number of active breakpoints and tracepoints is limited only by the amount of PSCOPE workspace available.

You can set both a breakpoint and a tracepoint at the same location but only one of each type at the same location. PSCOPE displays a warning message if you try to set a breakpoint (or tracepoint) where an active break (trace) point already exists. The original breakpoint (tracepoint) remains intact.

In addition, you can define a patch at a breakpoint and a tracepoint location (or a breakpoint or a tracepoint location). In this case, PSCOPE handles the tracepoint first (including any debug procedures associated with it). PSCOPE next handles the breakpoint (including any debug procedures associated with it) and finally the patch. However, if PSCOPE stops because of the breakpoint, PSCOPE does not execute the patch until the next GO command.

Breakpoints and tracepoints are active only during execution initiated with the GO command. They are automatically deactivated when control returns to PSCOPE. Note that breakpoints and tracepoints are not active during stepping with the PSTEP and LSTEP commands, while patches are active during stepping.

Note that PSCOPE deactivates all breakpoints and removes all break registers, trace registers, and patches when you invoke the LOAD command.

Example

The following example executes break and trace registers:

```
GO USING break_1, error_check, input
```

The following example reuses the breakpoints and tracepoints activated during the previous GO command:

```
GO
```

The following example activates all trace registers and one breakpoint:

```
GO USING trcreg TIL :dc.error
```

The following example initiates execution with no break or tracepoints:

```
GO FOREVER
```

The following sequence of commands illustrates the combined use of break registers, trace registers, and breakpoints. The example starts execution and prints a trace message every time procedure get_token is called. Execution stops when either error or get_line is called:

```
DEFINE TRCREG T1 = :dc.get_token
*DEFINE BRKREG B3 = error
*GO USING T1, B3 TIL get_line
```

Exception Trapping

PSCOPE automatically traps exception conditions within the user program. The exceptions trapped are from UTS, UDI, and the 8087 emulator and include DQEXIT. Unlike standard user breakpoints, these exceptions are always active; they are created, removed, and replaced only by the LOAD command.

PSCOPE displays two messages when an exception condition occurs. The first message identifies the type of exception. The second message is as follows:

```
[Stop at location]
```

Where:

location is the line number or address of the exception handler, not the location within the user program where the exception occurred.

A trap at DQ$EXIT lets you inspect variables and continue normal debugging when the program has completed its execution. This is a good opportunity to save definitions of debugger objects that you want to use in future debug sessions, such as patches, debug procedures, and break registers. At this point, you cannot continue program execution. A GO command after trapping at DQEXIT causes PSCOPE to exit.

APPENDIX A ERROR MESSAGES

This appendix lists the PSCOPE error messages. PSCOPE error messages are coded by number and listed in numeric order for easy reference.

Classes of Errors

Each of the errors detected by PSCOPE falls into one of the following five classes:

- WARNING. A minor problem which PSCOPE attempts to correct, then executes.
- ERROR. A problem of sufficient severity that PSCOPE aborts the command currently executing and either prompts for a new command or retrieves the next command from the current INCLUDE file (if any).
- SEVERE ERROR. A problem that may cause difficulties beyond the current command. PSCOPE aborts the current command, cancels any pending commands from INCLUDE files, and prompts for a new command from the terminal.
- FATAL ERROR. A problem from which PSCOPE cannot recover and reliably continue operating. PSCOPE closes all files, frees all resources that it or the program being debugged may have allocated, and returns control to the host operating system. (Very few PSCOPE errors are fatal. Do not worry about fatal errors aborting a debug session.)
- INTERNAL ERROR. A violation of one of PSCOPE's internal consistency checks. Please document the situation in which the error occurred and report it to your Intel representative.

HELP

Some errors have extended error messages. You can reach the extended error messages by using the following HELP command:

```
HELP En
```

Where:

n is the number of the error message.

PSCOPE indicates errors that have extended error messages by placing an asterisk enclosed in brackets ([*]) at the end of the primary message for that error.

Error Messages

0	Type definition record with an unrecognizable format.
1	Array's lower bound is unknown - zero is assumed.
2	Symbol is not an array or the symbol has fewer dimensions than specified.

3	Size of the array elements is not known.
4	Referenced array expects a single character array index.
5	Address of module is not known. Tried to reference an assembly language module, a run-time library, OS run-time, or a module with no debug information.
6	Unknown module specified.
7	No line information was loaded for module.
8	No symbol information was loaded for module.
9	Cannot determine module for specified location. [*] Could not find specified location in any known module. Specified location is either outside of the program or in a module for which there is no symbol information.
10	Cannot determine current default module. [*] Could not find current location in any known module. Either the current execution point is outside of the program or it is in a module for which there is no symbol information.
11	Symbol currently not active. [*] Symbol is either not known or is not local to the current procedure. 12. Symbol not known in current context. Change context with the NAMESCOPE command or use a fully qualified symbol reference.
13	No symbol information was loaded for program.
14	Attempt to reference a program symbol of an unsupported type.
15	Symbol is not known to be a record and cannot be qualified.
16	Symbol is not a known record field name.
17	Cannot determine offset of a field from the start of record. [*]. The requested field cannot be referenced because the debugger cannot determine the size of one of the preceding record fields.
18	Nested symbolic references not permitted.
19	Symbol isn't a pointer variable or its dereference type is unknown.
20	Specified line is not an executable statement.
21	Specified line does not exist in module.
22	Cannot evaluate line reference. [*] The segment part of the line reference pointer is not known. Maybe the symbol information was not loaded for the module.
23	Specified type is incompatible with directory. [*] Specified type cannot be used with the specified (or default) directory. For example, DIR PUBLIC LINE is contradictory, as there are no public lines.
24	Cannot perform symbol table request. No user program loaded.

40	Tried to REMOVE debugger object declared locally in DO..END block.
41	Workspace exceeded. [*] Out of workspace. Delete any unnecessary debugger objects (e.g., PROCs, LITERALLYs, PATCHes). This can also be caused by deeply recursive debug procedures.
42	The name is either undefined or not of the correct type.
43	The name is undefined.
44	The name is already defined with a different type.
45	Parameter outside the body of a PROC.
46	The name is not a PROC.
47	Illegal type specified in DIR DEBUG command.
48	The named object is not a literally.
49	Illegal assignment to register.
50	String too long to perform assignment.
51	Error in debug symbol lookup. [*] May be caused by removing a global debug variable referenced in a debug procedure (or patch) and then executing the debug procedure (or patch).
52	No patch defined at the specified location.
64	Attempt to PUT or APPEND a local debug object.
65	I/O error on PUT file.
66	This command is not currently implemented.
67	This command not allowed inside of a compound command.
68	Invalid type.
69	Invalid type conversion.
70	String longer than 254 characters.
71	String too long for numeric conversion. [*] Character strings must be of length 1 to convert to unsigned numbers.
72	Illegal type in output.
73	Unmatched double quotes in format string.
74	Write list too long. The maximum is 20 items.
75	Write data too large. The maximum is 256 bytes.

76	Invalid format string in WRITE command.
77	Output buffer overflow. The limit is 128 characters per line.
78	Invalid floating point value for output.
79	Invalid expression for MTYPE. An illegal value is being assigned to a memory template.
80	Invalid boolean operation.
81	Invalid string operation.
82	Invalid pointer operation.
83	Invalid unknown operation.
84	Attempt to assign value to code instead of variable. [*] Tried to assign an expression to a location associated with user data (e.g., :main.procl = 5, where procl is a procedure in module main). Straight assignments may be made only to variables or with memory modify commands (e.g., byte 100:200 = 5).
85	Attempt to assign illegal value to BASE variable.
86	Cannot use editor if debugger was invoked with SUBMIT control.
87	Not in a procedure or in a procedure with no debug information. [*] In order for the calling procedure to be identified (and the CALL-STACK command to function properly), the current execution point must be in a procedure or in a procedure for which there is debug information.
88	The debugger has overflowed its 86 stack. [*] The debugger has overflowed its stack, probably due to deep recursion of a debug procedure.
89	UDI exception. A PSCOPE operation resulted in a UDI exception. A divide-by-zero on unsigned values will cause this error.
90	Literally nesting too deep.
91	Illegal extended integer.
92	Attempt to assign illegal value to INPUTMODE variable.
93	Error in VIEW command.
110	No data segment information. Program may execute incorrectly. [*] The load module did not provide any information about the data segment. Therefore, execution of the program may have unexpected results.
111	No stack segment information. Program may execute incorrectly. [*] The load module did not provide any information about the stack segment. Therefore, execution of the program may have unexpected results.

112 Program cannot be loaded. [*]
Program start address needs a fix up by the linker.

113 The 8087 Emulator was not found in the load module. [*]
If the E8087 option is specified in the load command, then the 8087 emulator must be linked into the program being debugged. It was not found at load, so it either never existed or it was purged.

114 Missing CH8087 option when loading a program with real math.

115 Bad object record in load file.
Verify that you are loading an LTL object file. If there are still bad records, relink module.

116 Load file contains absolute load addresses. [*]
Load file is not PIC or LTL. Relink with the BIND control.

117 Load file contains unresolved externals.
Program must be relinked before debugging.

118 Support for overlays not implemented.
Loaded program cannot contain overlays.

119 Memory segment request failure during load. [*]
More memory is needed to load the program. Deleting debugger objects will not increase available memory for loading.

120 Load module contained no starting address information. [*]
The load module did not provide any information about the starting address. The load was aborted, and execution of the program is not possible.

136 Divide by zero (operation yields 0 result).

137 Invalid type for arithmetic.

138 Invalid integer operation.

139 Real math is not available. [*]
In order to use real math (including any operations or reference to real numbers), you must use the E8087 option on the LOAD command and have the 8087 emulator linked into the program under debug. This error may be detected if the E8087 option was used on the LOAD command with a program that appears to have the emulator linked into it but does not. (This can happen with Pascal and FORTRAN programs linked with 87NULL.LIB.)

140 Invalid real number.

141 Attempted real comparison with NAN, +infinity or -infinity.

142 Invalid real operation.

143 Invalid extended integer operation.

144 Illegal numeric constant.

160 Attempt to INCLUDE :CI:.

161 I/O error on INCLUDE file.

162	I/O error on LIST file.
163	I/O error while loading object file.
164	Could not open load file.
165	Error while attempting to open virtual symbol table. [*] The virtual symbol table uses :WORK: for the disk-resident portion of the virtual symbol table. Ensure that the device for :WORK: is ready and that PSCOPE has access rights to it.
166	Error while attempting to seek in virtual symbol table.
167	Error while attempting to write to virtual symbol table.
168	Error while attempting to close virtual symbol table.
169	Error while attempting to read virtual symbol table.
177	First address is greater than second address.
178	Attempt to use VIEW command while running PSCOPE under SUBMIT.
196-249	Errors 196 through 511 are PSCOPE internal errors. They result from consistency check failures and should never occur. If an internal error does occur, please notify an Intel representative.
353	Illegal number.
354	Unrecognized 8086/8087 mnemonic.
355	Illegal use of indirect addressing. [*] The correct forms of indirect addressing are: <symbolic ref> [BX] + offset <symbolic ref> [BP] + offset <symbolic ref> [DI] + offset <symbolic ref> [SI] + offset <symbolic ref> [BP] [DI] + offset <symbolic ref> [BP] [SI] + offset <symbolic ref> [BX] [DI] + offset <symbolic ref> [BX] [SI] + offset The symbolic reference (of the form :MODULE.SYMBOL.SYM BOL.etc)and the '+ offset' are optional.
356	Illegal single line assembler operand.
357	Single line assembler syntax error. See HELP SASM.
358	Memory pointer (eg. BYTE, WORD, etc) without memory operand (eg. number or symbolic reference).
359	Too few operands for this instruction.
360	Illegal operands, both operands appear to reference memory.
361	The types of the operand(s) do not match the mnemonic or each other.

362	One byte relative jump is out of range. Range is −128 to +127.
512	The cause of execution break is unknown to PSCOPE. [*] PSCOPE cannot determine how execution was broken; it was not through a known breakpoint or a CNTL-C. You probably placed an interrupt at the given address or entered CNTL-D.
513	This breakpoint is already active. [*] You can activate only one breakpoint of each type (break, trace, or patch) at any one address. The break you originally activated is still intact.
514	Invalid return type from PROC called at breakpoint. [*] The debugger procedure called at the breakpoint or tracepoint returned a value with an invalid type or had no return value. The return value must be a BYTE, WORD, DWORD, BOOLEAN, or INTEGER (including LONG/SHORT). PSCOPE manufactured a return value of TRUE, causing the associated break or trace to be executed.
515	There was a patch in progress and it was not completed. [*] A code PATCH was being executed when execution was interrupted. The current execution point is the standard resume address (the point in the program to which control would normally be transferred after the patch), as if the PATCH had completed (unless the PATCH changed it). The entire PATCH will most likely not have completed execution. If the resume address is the PATCH address, then restarting execution re-executes the patch.
528	Attempted recursive definition of a break or trace register. [*] Tried to define the named break register or trace register while already in the process of defining one. This happens when an expression in the definition of a break or trace register calls a debug procedure which defines the named break or trace register.
529	Cannot determine proper statement address for step. [*] Either PSCOPE cannot determine the current execution point and, therefore, cannot do statement level stepping, or you tried to start statement-level stepping when the current execution point is not the beginning of a statement. In the latter case, use the GO command to get to a statement, then retry the step.
530	No break or trace registers (of the requested type) have been defined.
531	This command cannot occur inside of a PATCH.
532	No program was loaded.
544-546	Errors 544 through 546 are PSCOPE internal errors. They result from consistency check failures and should never occur. If an internal error does occur, please notify an Intel representative.

APPENDIX B CONFIGURING PSCOPE FOR NON-INTEL TERMINALS

Configuration Commands

PSCOPE is designed to run on an Intellec Series III or Series IV development system. The editor expects code from the terminal or sent to the terminal to be code used by Intel terminals.

You can, however, configure PSCOPE to operate with other terminals. You need configuration files when using a non-standard or non-Intel terminal with characteristics different from those of the Series III or Series IV screen. Configuration files let you indicate characteristics of your particular terminal by setting various parameters and specifying control sequences by which various screen functions can be performed. Configuration files are not needed when using a Series III or Series IV with the integrated screen.

You should put configuration commands in a CRT configuration file (e.g., PSCOPE.CRT) so that they are automatically executed when you invoke PSCOPE. The configuration commands let you modify certain keyboard and CRT codes. In some situations, you may not be able to use certain editing functions.

To create a PSCOPE configuration file, compare your terminal's behavior to the actions expected by PSCOPE. Refer to your user manual for the codes that your terminal expects and generates. (See Table B-1 for a list of the PSCOPE configuration commands, their default values, and meaning.)

Note that the CRT configuration commands are compatible with the configuration commands accepted by the Series III or a Series IV text editor, AEDIT, in its AEDIT.MAC file. (See *AEDIT Text Editor User's Guide*, order number 121756.)

PSCOPE expects the following characteristics in a terminal:

- ASCII codes 20H through 7EH display some symbol requiring one column space. Carriage return (0DH) and line feed (0AH) perform their usual functions.
- There are cursor key output codes and CRT cursor output codes for the following cursor functions: down, home, left, right, and up. Output codes for clear screen, clear rest of screen, clear line, clear rest of line, and direct cursor addressing are desirable but not required. You can change default codes, shown in Table B-1, with the configuration commands.
- The terminal accepts a blankout code that blanks out the contents of the screen location from which it is entered. You can change the default, 20H, with the configuration commands.
- The CRT has 22 to 25 lines. You can change the default, 25 lines, with the configuration commands.
- PSCOPE automatically generates a line feed each time you enter a carriage return. Your terminal should not generate a line feed with a carriage return. You can switch this feature on and off on some terminals.

When configuring to execute on a non-Intel terminal, you may have to change some or all of the codes assigned to the following configuration commands:

- The cursor key output codes expected by the editor: AFCH, AFCU, AFCD, AFCR, and AFCL.
- The editor-generated cursor movement codes sent to the CRT: AFMH, AFMU, AFMD, AFMR, AFML.
- The erase screen code, AFES.
- The blankout code, AFBK.
- The screen size code, AV.
- The BREAK character code, AB.
- The codes expected by the editor for the screen mode commands: AFXA, AFXF, AFXX, AFXU, and AFXZ. You may want to change these codes to match function keys or other convenient keys on the terminal keyboard.

Table B-1 lists the configuration commands, their default values, and their meaning.

The following conventions apply to Table B-1:

- *n* must be 22, 23, 24, or 25.
- *h* is a one-byte hexadecimal number.
- *hhhh* is a one- to four-byte hexadecimal number. A null value indicates that the function is not available.
- T is 'T' or 't', indicating true.
- F is 'F' or 'f', indicating false.

You must end all commands in the CRT file with a semicolon (;) or a carriage return.

Table B-1 Configuration Commands

Command	Series III Default	Meaning
AV=n	25	Sets the number of lines of the display.
AB=hhhh	1BH	Sets ESC.
AR=hhhh	7FH	Sets RUBOUT.
AFXA=hhhh	1H	Sets DELETE RIGHT (CNTL-A)
AFXF=hhhh	6H	Sets CHAR DELETE (CNTL-F)
AFXX=hhhh	18H	Sets DELETE LEFT (CNTL-X)

Table B-1 Configuration Commands (continued)

Command	Series III Default	Meaning
AFXZ=hhhh	1AH	Sets CLEAR LINE (CNTL-Z)
AFCD=hhhh	1CH	Sets DOWN.
AFCH=hhhh	1DH	Sets HOME.
AFCL=hhhh	1FH	Sets LEFT.
AFCR=hhhh	14H	Sets RIGHT.
AFCU=hhhh	1EH	Sets UP.
AFIG=h		This character will be ignored if it is entered. This character is needed on terminals, which have multiple character key codes for UP and DOWN, such as the Hazeltine 1510. AFIG should be set to the lead in (tilde) and UP and DOWN should be set to the second letter of the cursor up or down key code. This avoids problems caused by the lack of a type-ahead buffer.
AFMB=hhhh	0DH	Moves the cursor to the start of the line.
AFMD=hhhh	1CH	Moves the cursor down.
AFMH=hhhh	1DH	Moves the cursor home.
AFML=hhhh	1FH	Moves the cursor left.
AFMR=hhhh	14H	Moves the cursor right.
AFMU=hhhh	1EH	Moves the cursor up.
AFES=hhhh	1B45H	Erases the entire screen.
AFER=hhhh	1B4AH	Erases the rest of the screen.
AFEK=hhhh	1B4BH	Erases the entire line.
AFEL=hhhh		Erases the rest of the line.
AFAC=hhhh		Addresses the cursor lead-in. When used, the code will be followed by a column number (0 to 79) and a row number (0 to 24).
AO=h	0H	Offset to add both a row and a column number with an address cursor command.
AX=T or F	T	True if X (column) precedes Y (row) in the address cursor command.
AW=T or F	T	Allows the user to indicate that the terminal wraps when the character is printed in column 80.
AFIL=hhhh		Inserts the line code. Used in line 0 for reverse scrolling.
AFDL=hhhh		Deletes the line code. Used to speed up the display on the Hazeltine 1510 and similar terminals.
AFBK=h	20H	Blankout character. BLANK on most terminals.

Tested Configurations

This appendix contains tested configurations for several non-Intel terminals. The terminals presented here are not the only ones on which you can use PSCOPE; they are just the ones that have been tested. The following sections list the configuration functions and values required to run PSCOPE on the Intel tested terminals. The terminals are as follows:

- ADDS Regent 200 (2400 baud only)
- ADDS Viewpoint 3A Plus
- Beehive Mini-Bee
- DEC VT52
- DEC VT100
- Hazeltine 1420
- Hazeltine 1510 (Tilde lead-in)
- Hazeltine 1510 (ESC lead-in)
- Intel Series III E
- Lear Seigler ADM-3A
- Televideo 910 Plus
- Televideo 925 and 950
- Zentec

The commands to configure PSCOPE for the tested terminals are included on the disk with the PSCOPE program. The name of the file is included in each description.

Configuration files for the following terminals can be created by entering the commands specified in the corresponding tables.

ADDS Viewpoint 3A Plus
Hazeltine 1420
Intel Series IIIE
Televideo 910 Plus
Televideo 925 and 950
Zentec

ADDS Regent Model 200

The ADDS model has a 24-line CRT display with 80 characters per line. Each character is formed in an 8 by 8 dot matrix as a dark character on a light background. The 25th line of the screen displays the operating condition of the terminal. Table B-2 shows the ADDS Regent Model 200 configuration.

Table B-2 ADDS Regent Model 200 Configuration

Function Code	Hexadecimal Value	Graphic or ASCII Name
CD	0A	Line Feed
CH	01	SOH
CL	15	NAK or BS
CR	06	ACK
CU	1A	SUB
MD	0A	Line Feed
MH	1B 59 20 20	
ML	15	NAK or BS
MR	06	ACK
MU	1A	SUB
AC	1B 59	ESC Y
EK	not available	
ER	1B 6B	ESC K
ES	0C	FF
XA	14	DC4
AO	20	SP
AX	F	
XF	1B 45	ESC E
XZ	1B 6C	ESC I
AB	5C	
AV		24

Command File: ADDS.CRT

```
AFCD=0A            AFCL=15       AFCR=06
AFCU=1A            AFCH=01       AFMD=0A
AFML=15            AFMR=06       AFMU=1A
AFMH=1B 59 20 20   AFEK=         AFER=1B 6B
AV=24              AFXA=14       AFES=0C
AFER=1b6b          AFAC=1B 59    AO=20
AX=F               AFXF=1B45     AFXZ=1B 6C
AB=5C
```

NOTE

You must enter DEL CHAR instead of CNTL-F for the delete character. You must enter DEL LINE instead of CNTL-Z for delete line. You must enter CNTL-T instead of CNTL-A for delete right. You must enter the backslash (\) instead of ESCAPE.

ADDS Viewpoint 3A Plus

This terminal has a 24-line CRT display with 80 characters per line. Table B-3 shows the ADDS Viewpoint 3A Plus configuration.

Table B-3 ADDS Viewpoint 3A Plus Configuration

Function Code	Hexadecimal Value	Graphic or ASCII Name
CD	0A	Line Feed
CL	08	BS
CR	0C	FF
CU	0B	VT
CH	1E	RS
MD	0A	Line Feed
ML	08	BS
MR	0C	FF
MU	0B	VT
MH	1E	RS
EK	not available	
ER	1B 59	ESCY
EL	1B 54	ESC
ES	1B 2A	ESC
AC	1B 3D	ESC=
AO	20	SP
AX		F
AV		24

Command File: ADDS.CRT

```
AFCD=0A        AFCL=08        AFCR=0C
AFCU=0B        AFCH=1E        AFMD=0A
AFML=08        AFMR=0C        AFMU=0B
AFMH=1E        AFEK=          AFER=1B 59
AFEL=1B 54     AFES=1B 2A     AFAC=1B 3D
AO=20          AX=F           AV=24
```

Beehive Mini-Bee

You can format the Beehive Mini-Bee terminal to display either 12 or 25 lines of 80 characters per line. Only the 25-character format is usable with PSCOPE. Each character is generated in a 5 by 7 dot matrix. The maximum transmission rate for this terminal is 9600 baud. Note that you must change the ESCAPE character so that the default ESCAPE code can be used; choosing the ↑K is a personal preference. Table B-4 shows the Beehive Mini-Bee configuration.

Table B-4 Beehive Mini-Bee Configuration

Function Code	Hexadecimal Value	Graphic or ASCII Name
CD	1B 42	ESC B
CH	1B 48	ESC H
CL	1B 44	ESC D
CR	1B 43	ESC C
CU	1B 41	ESC A
MD	1B 42	ESC B
MH	1B 48	ESC H
ML	1B 44	ESC D
MR	1B 43	ESC C
MU	1B 41	ESC A
EL	1B 4B	ESC K
ER	1B 4A	ESC J
B	0B	↑K
AV		24

Command File: MICROB.CRT

```
AFCU=1B 41        AFCD=1B 42        AFCR=1B 43
AFCL=1B 44        AFCH=1B 48        AFMU=1B 41
AFMD=1B 42        AFMR=1B 43        AFML=1B 44
AFMH=1B 48        AFEL=1B 4B        AFER=1B 4A
AB=0B             AV=24
```

NOTE

You must enter CNTL-K instead of ESCAPE.

DEC VT52

The DEC VT52 displays 24 lines of 80 characters per line. The characters are generated in a 7 by 9 dot matrix. The maximum transmission rate is 19.2K baud. Note that you must change the ESCAPE character so that the default ESCAPE code can be used; choosing CNTL-K (↑ K) is a personal preference. The DEC VT52 does not have a HOME key. Choosing CNTL-O (↑ O) for the HOME function is a personal preference. Table B-5 shows the DEC VT52 configuration.

Table B-5 DEC VT52 Configuration

Function Code	Hexadecimal Value	Graphic or ASCII Name
CD	1B 42	ESC B
CH	0F	↑O
CL	1B 44	ESC D
CR	1B 43	ESC C
CU	1B 41	ESC A
MD	1B 42	ESC B
MH	1B 48	ESC H
ML	1B 44	ESC D
MR	1B 43	ESC C
MU	1B 41	ESC A
AC	1B 59	ESC Y
W	F	
AO	20	SP
AX	F	
EL	1B 4B	ESC K
ER	1B 4A	ESC J
ES	not available	
EK	not available	
AV		24
B	0B	↑K

Command File: VT52.CRT

```
AFCU=1B 41        AFCD=1B 42        AFCR=1B 43
AFCL=1B 44        AFCH=0F           AFMU=1B 41
AFMD=1B 42        AFMR=1B 43        AFML=1B 44
AFMH=1B 48        AFES=             AFER=1B 4A
AFEL=1B 4B        AFEK=             AB=0B
AV=24             AFAC=1B 59        AO=20
AX=F              AW=F
```

NOTE

You must enter CNTL-K instead of ESCAPE. You must enter CNTL-O instead of HOME.

DEC VT100

You can format the DEC VT100 terminal with 14 lines of 132 characters per line or 24 lines of 80 characters per line. Only the 24-line format is compatible with PSCOPE. The characters are generated in a 7 by 9 dot matrix. The maximum transmission rate is 19.2K baud. You can choose between the DEC VT52 compatible and the ANSI standard (X3.41-1974, X3.64-1977) compatible terminal escape sequences for cursor control and screen erase functions. The ANSI codes are given in the following table. See the DEC VT52 description for the VT52 codes. Note that you must change the ESCAPE character so that the default ESCAPE code can be used; choosing CNTL-K (↑ K) is a personal preference. The DEC VT100 terminal does not have a HOME key. Choosing CNTL-O (↑ O) for the HOME function is a personal preference. Table B-6 shows the DEC VT100 configuration.

Table B-6 DEC VT100 Configuration

Function Code	Hexadecimal Value	Graphic or ASCII Name
CD	1B 42	ESC B
CH	0F	↑O
CL	1B 44	ESC D
CR	1B 43	ESC C
CU	1B 41	ESC A
MD	1B 5B 42	ESC [B
MH	1B 5B 48	ESC [H
ML	1B 5B 44	ESC [D
MR	1B 5B 43	ESC [C
MU	1B 5B 41	ESC [A
EK	1B 5B 30 4B	ESC [0 K
ER	1B 5B 30 4A	ESC [0 J
ES	not available	
EL	1B 5B 4B	
W	F	
AV		24
B	0B	↑K

Command File: VT100.CRT

```
AFCU=1B 41            AFCD=1B 42            AFCR=1B 43
AFCL=1B 44            AFCH=0F               AFMU=1B 5B 41
AFMD=1B 5B 42         AFMR=1B 5B 43         AFML=1B 5B 44
AFMH=1B 5B 48         AFES=                 AFER=1B 5B 30 4A
AFEK=1B 5B 30 4B      AFEL=1B 5B 4B         AB=0B
AV=24                 AW=F
```

NOTE

You must enter CNTL-K instead of ESCAPE. You must enter CNTL-O instead of HOME.

Hazeltine 1420

This terminal displays 24 lines, with 80 characters per line. The maximum transmission rate is 9600 baud. You may choose between the tilde key or the ESC character as the control sequence lead-in. To use the ESC character as the lead-in, substitute escape (1B) for the tilde (7E) in the command file, and add the function code AB=7E. When using the escape lead-in, the tilde must be typed instead of escape. Table B-7 shows the Hazeltine 1420 configuration.

Table B-7 Hazeltine 1420 Configuration

Function Code	Hexadecimal Value	Graphic or ASCII Name
CD	0B	VT
CH	12	DC2
CL	08	BS
CR	10	DLE
CU	0C	FF
MD	7E 0B	~ VT
MH	7E 12	~ DC2
ML	08	BS
MR	10	DLE
MU	7E 0C	~ FF
MB	0D	CR
ES	not available	
ER	7E 18	~ CAN
EK	not available	
EL	7E 0F	~ SI
AC	7E 11	~ DC1
IL	7E 1A	~ SUB
DL	7E 13	~ DC3
AV		24
IG	7E	~

Command File: 1420T.MAC

```
AV=24          AFIG=7E        AFCU=0C
AFCD=0B        AFCR=10        AFCL=8
AFCH=12        AFMU=7E 0C     AFMD=7E 0B
AFMR=10        AFML=8         AFMH=7E 12
AFMB=0D        AFES=          AFER=7E 18
AFEK=          AFEL=7E 0F     AFAC=7E 11
AFIL=7E 1A     AFDL=7E 13
```

Hazeltine 1510

The Hazeltine 1510 terminal displays 24 lines of 80 characters per line. The characters are generated in a 7 by 10 dot matrix. The maximum transmission rate is 19.2K baud. You can choose between the ESC or the tilde character (~) as the control sequence lead-in. However, if you use ESC, you must change the BREAK character, so the tilde is easier to use. Table B-8 shows the Hazeltine 1510 configuration with the tilde lead-in and Table B-9 shows the Hazeltine 1510 configuration with the ESC lead-in.

Table B-8 Hazeltine 1510 Configuration (Tilde Lead-in)

Function Code	Hexadecimal Value (~ lead-in)	Graphic or ASCII Name
CD	0B	~ VT
CH	12	~ DC2
CL	08	~ BS
CR	10	~ DLE
CU	0C	~ FF
MD	7E 0B	~ VT
MH	7E 12	~ DC2
ML	08	~ BS
MR	10	~ DLE
MU	7E 0C	~ FF
MB	0D	
AC	7E 11	~ DC1
EK	not available	
ER	7E 18	~ CAN
ES	not available	
EL	7E 0F	~
XP	0F	SI
IL	7E 1A	~ SUB
DL	7E 13	~ DC3
AV		24

Command File: 1510T.CRT

```
AV=24          AFIG=7E        AFCU=0C
AFCD=0B        AFCR=10        AFCL=8
AFCH=12        AFMU=7E 0C     AFMD=7E 0B
AFMR=10        AFML=8         AFMH=7E 12
AFMB=0D        AFES=          AFER=7E 18
AFEK=          AFEL=7E 0F     AFAC=7E 11
AFIL=7E 1A     AFDL=7E 13
```

Table B-9 Hazeltine 1510 Configuration (ESC Lead-in)

Function Code	Hexadecimal Value (ESC lead-in)	Graphic or ASCII Name
CD	0B	ESC VT
CH	12	ESC DC2
CL	08	ESC BS
CR	10	ESC DLE
CU	0C	ESC FF
MD	1B 0B	ESC VT
MH	1B 12	ESC DC2
ML	08	ESC BS
MR	10	ESC DLE
MU	1B 0C	ESC FF
MB	0D	
EK	not available	
ER	1B 18	ESC CAN
ES	not available	
EL	1B 0F	
IL	1B 1A	ESC SUB
DL	1B 13	ESC DC3
XP	0F	SI
AC	1B 11	ESC DC1
AV		24
B	7E	

Command File: 1510E.CRT

```
AV=24          AB=7E          AFIG=1B
AFCU= 0C       AFCD= 0B       AFCR=10
AFCL=8         AFCH= 12       AFMU=1B0C
AFMD=1B0B      AFMR=10        AFML=8
AFMH=1B12      AFMB=0D        AFES=
AFER=1B18      AFEK=          AFEL=1B0F
AFAC=1B11      AFIL=1B1A      AFDL=1B13
```

Intel Series-IIIE

The Intel Series-IIIE terminal displays 24 lines of 80 characters per line. The maximum transmission rate is 19.2K baud. Table B-10 shows the Intel Series-IIIE configuration.

Table B-10 Intel Series-IIIE Configuration

Function Code	Hexadecimal Value	Graphic or ASCII Name
ER	1B 53	ESC S
AC	1B 59	ESC Y
IL	1B 57 60 3F	ESC W '?
DL	1B 57 3F 60	ESC W ?'
EL	1B 52	ESC R
AX		F
AO	20	SP

Command File: 511IOC.MAC

AX=F	AO=20	AFER=1B 53
AFAC= 1B 59	AFIL= 1B 57 60 3F	AFDL=1B 57 3F 60
AFEL=1B 52		

NOTE

A Series-IIIE with 511 IOC firmware must use this macro to take advantage of the enhanced functionality.

Lear Siegler ADM-3A

The Lear Siegler ADM-3A terminal displays 24 lines of 80 characters per line. The characters are generated in 5 by 7 dot matrix. The maximum transmission rate is 19.2K baud. Table B-11 shows the Lear Siegler ADM-3A configuration.

Table B-11 Lear Siegler ADM-3A Configuration

Function Code	Hexadecimal Value	Graphic or ASCII Name
CD	0A	LF
CH	1E	RS
CL	08	BS
CR	0C	FF
CU	0B	VT
MD	0A	LF
MH	1E	RS
ML	08	BS
MR	0C	FF
MU	0B	VT
EK	not available	
ER	not available	
ES	1A	SUB
AX	F	
AO	20	SP
AC	1B 3D	ESC =
AV		24

Command File: LEAR.CRT

```
AFCU=0B        AFCD=0A        AFCR=0C
AFCL=08        AFCH=1E        AFMU=0B
AFMD=0A        AFMR=0C        AFML=08
AFMH=1E        AV=24          AFES=1A
AFER=          AFEK=          AFAC=1B 3D
AX=F           AO=20
```

Televideo 910 Plus

The Televideo 910 Plus displays 24 lines of 80 characters per line. The maximum transmission rate is 19.2K baud. Table B-12 shows the Televideo 910 Plus configuration.

Table B-12 Televideo 910 Plus Configuration

Function Code	Hexadecimal Value	Graphic or ASCII Name
CD	16	SYN
CH	1E	RS
CL	08	BS
CR	0C	FF
CU	0B	VT
MD	16	SYN
MH	1E	RS
ML	08	BS
MR	0C	FF
MU	0B	VT
ES	1B 2B	ESC +
ER	1B 59	ESC Y
EK	not available	
EL	1B 54	ESC T
AC	1B 3D	ESC =
AX		F
AO	20	SF
IL	1B 45	ESC E
DL	1B 52	ESC R
AV		24

Command File: TV910P.MAC

```
AFCU=0B          AFCD=16          AFCR=0C
AFCL=08          AFCH=1E          AFMU=0B
AFMD=16          AFMR=0C          AFML=08
AFMH=1E          AV=24            AFES=1B 2B
AFER=1B 59       AFEK=            AFEL=1B 54
AFAC=1B 3D       AX=F             AO=20
AFIL=1B 45       AFDL=1B 52
```

Televideo 925 and 950

The Televideo 925 and 950 displays 24 lines of 80 characters per line. The maximum transmission rate is 19.2K baud. Table B-13 shows the Televideo 925 and 950 configuration.

Table B-13 Televideo 925 and 950 Configuration

Function Code	Hexadecimal Value	Graphic or ASCII Name
XA	01	NUL
XF	06	ACK
XU	15	NAK
XX	18	CAN
XZ	1A	SUB
CD	16	SYN
CH	1E	RS
CL	08	BS
CR	0C	FF
CU	0B	VT
MB	0D	CR
MD	16	SYN
MH	1E	RS
ML	08	BS
MR	0C	FF
MU	0B	VT
ES	1B 2B	ESC +
ER	1B 59	ESC Y
EK	not available	
DL	1B 52	ESC R
EL	1B 54	ESC T
IL	1B 45	ESC E
AC	1B 3D	ESC =
AO	20	SP
AX		F
AV		24
AB	1B	ESC
AR	7F	DEL
BK	20	SP

Command File: TV925.MAC

```
AV=24          AB=1B          AR=7F
AFBK=20        AFXA=1         AFXF=6
AFXU=15        AFXX=18        AFXZ=1A
AFCD=16        AFCH=1E        AFCL=08
AFCR=0C        AFCU=0B        AFMB=0D
AFMD=16        AFMH=1E        AFML=08
AFMR=0C        AFMU=0B        AFES=1B 2B
AFER=1B 59     AFEK=          AFDL=1B 52
AFEL=1B 54     AFIL=1B 45     AFAC=1B 3D
AO=20          AX=F
```

Zentec

The Zentec displays 24 lines of 80 characters per line. The maximum transmission rate is 19.2K baud. To rub out a character on this terminal you must use the delete key, SHIFT plus ESC. Table B-14 shows the Zentec configuration.

Table B-14 Zentec Configuration

Function Code	Hexadecimal Value	Graphic or ASCII Name
XA	1	SOH
XF	6	ACK
XU	15	NAK
XX	18	CAN
XZ	1A	SUB
CD	0A	LF
CH	1E	RS
CL	08	BS
CR	0C	FF
CU	0B	VT
MB	0D	CR
MD	0A	LF
MH	1E	RS
ML	08	BS
MR	0C	FF
MU	0B	VT
ES	1B 2B	ESC +
ER	1B 59	ESC Y
EK	not available	
DL	1B 52	ESC R
EL	1B 54	ESC T
IL	1B 45	ESC E
AC	1B 3D	ESC =
AO	20	SP
AX		F
BE	20	SP
AV		24
AB	1B	ESC
AR	7F	DEL

Command File: ZENTEC.MAC

```
AV=24          AB=1B          AR=7F
AFXA=1         AFXF=6         AFXU=15
AFXX=18        AFXZ=1A        AFCD=0A
AFCH=1E        AFCL=08        AFCR=0C
AFCU=0B        AFMB=0D        AFMD=0A
AFMH=1E        AFML=08        AFMR=0C
AFMU=0B        AFES=1B 2B     AFER=1B 59
AFEK=          AFDL=1B 52     AFEL=1B 54
AFIL=1B 45     AFAC=1B 3D     AO=20
AX=F           AFBK=20
```

APPENDIX C ADDITIONAL INFORMATION FOR SERIES III USERS

This appendix contains specific information on the Intellec Series III development system. It covers the following subjects:

- Series III program development process (and related manuals).
- Hardware and software required.
- User programs supported.
- System resources used by the debugger.
- System-specific examples of debugger invocation line, sign-on message, and commands.

Operation of the Series III

The following manuals describe the general operation of the Series III:

- *Intellec® Series III Microcomputer Development System Product Overview*, order number 121575
- *Intellec® Series III Microcomputer Development System Console Operating Instructions*, order number 121609
- *Intellec® Series III Microcomputer Development System Programmer's Reference Manual*, order number 121618

Program Development Process

Figure 1-1 shows how the debugger fits into your program development process. Figure C-1 shows the same process. The following manuals will provide information on the different stages of your program development.

- *ISIS-II CREDIT™ CRT-Based Text Editor User's Guide*, order number 9800902
- *AEDIT TEXT EDITOR User's Guide*, order number 121756
- *Pascal-86 User's Guide*, order number 121539
- *PL/M-86 User's Guide for 8086-Based Development System*, order number 121636
- *FORTRAN-86 User's Guide*, order number 121570
- *iAPX 86,88 Family Utilities User's Guide*, order number 121616

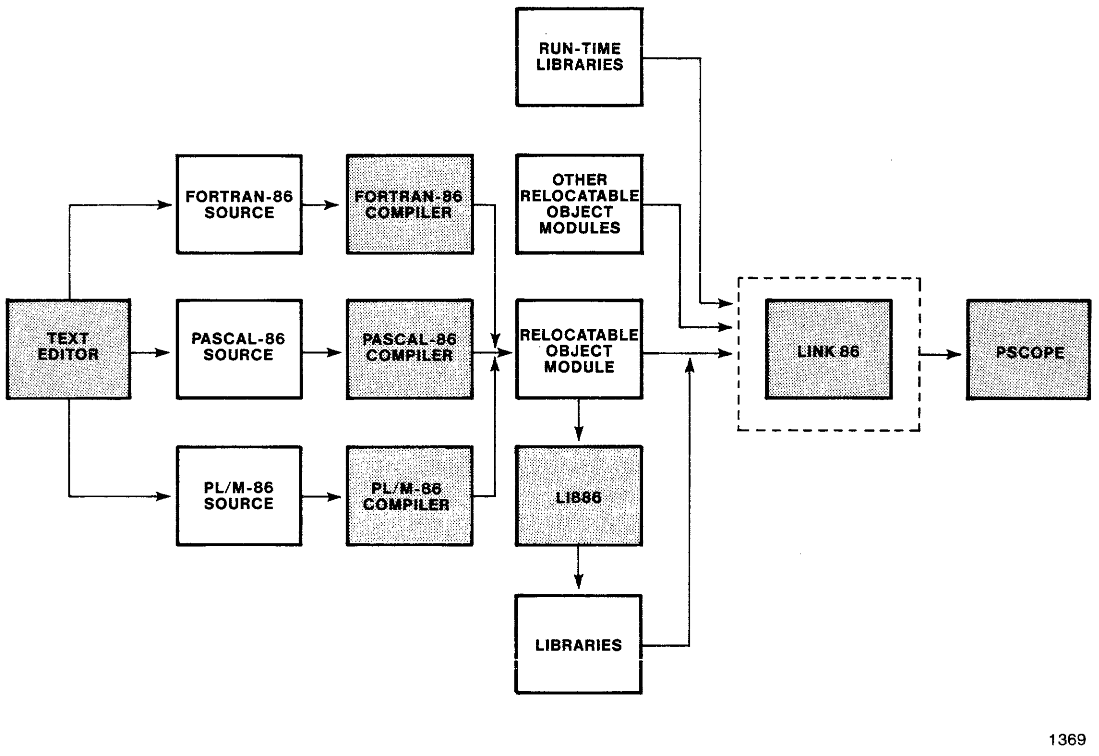

Figure C-1 Series III Program Development Process

Hardware and Software Required

You need the following hardware and software to run the debugger:

- Intellec Series III development system (run release 2.0 or later).
- ISIS operating system (release 4.1 or later).
- At least one single- or double-density flexible disk drive, a hard disk unit plus a single- or double-density flexible drive, or a remote disk on an NDS I or NDS II.
- Pascal compiler (release 2.0 or later), PL/M-86 compiler (release 2.0 or later), or FORTRAN-86 compiler (release 2.0 or later).
- 8086-based utilities.
- PSCOPE high-level program debugger.

User Programs Supported

The amount of memory available to your program (the program under debug) depends upon the amount of memory in your system. You can expand the Series III up to one megabyte of memory addressable by the 8086. PSCOPE requires approximately 110K bytes. You must add more memory to accommodate additional workspace and your program.

Your program must be a load-time locatable (LTL) or a position independent code (PIC) object module produced by LINK86 with the BIND control. You must produce the object modules used as input to LINK86 with either a Pascal-86 compiler (release 2.0 or later), a PL/M-86 compiler (release 2.0 or later), or a FORTRAN-86 compiler (release 2.0 or later) with the DEBUG control. Because the reliability of some debug functions can be affected by cross-statement compiler optimizations, you must use OPTIMIZE (0).

System Resources Used

The debugger requires certain system resources, such as memory space and open files, that can affect your program.

Memory

The debugger occupies 110K bytes of memory, including space for symbol and line number information.

To reduce memory usage, the debugger provides for virtual storage of compiler-generated debug information. Symbol table information (from the compiler) is sent out to disk if necessary. Your program must reside in memory, however.

File Requirements

Under the Series III operating system, up to six open files are available for an application, plus terminal input. (Terminal output does not count as an open file.) Terminal input does not count against the total of six open files allowed because the operating system shares terminal input between PSCOPE and your program.

Of those six files, PSCOPE may require one or more files from each of the following groups:

- Console input
- Virtual symbol table
- LOAD, HELP, PUT, INCLUDE, CRT, MACRO, PSCOPE overlay
- List

The number of open files increases if you have nested open files, such as a PUT command inside of an INCLUDE file.

Other Resources Required

The debugger requires the following additional host system resources:

- The software interrupt 3 (the one-byte, debugger-oriented INT instruction)
- The trap flag (used for single-stepping)
- The CNTL-C trap (system call DQ$TRAP$CC)

Your program should not use these host system resources.

In addition, PSCOPE uses interrupts 0, 4, 5, 16, 17, and 20 through 31 for error handling and floating point operations. However, your program can use these interrupts, since PSCOPE maintains separate copies of these interrupt vectors for itself and your program.

Invocation Line

To invoke the debugger in the 8086 execution environment of the Series III, preface the invocation line with the RUN command. The ISIS-II operating system prompt is a hyphen (-).

The general format of the invocation is:

-RUN [:F*n*:] PSCOPE [*controls*]

or

>[:Fn:] PSCOPE [*controls*]

Where:

:F*n*:	is the disk in drive *n*. The *n* can be 0 through 9.
controls	is any number of invocation controls from the list specified in Chapter 3.

PSCOPE signs on with the following message:

```
SERIES-III PSCOPE-86, Vx.y
```

Example

The following example shows the beginning of a PSCOPE debugging session:

```
-RUN PSCOPE MACRO(:F1:PROCS.MAC)

SERIES-III PSCOPE-86, Vx.y
*LOAD :F1:DC.86
*GO
```

APPENDIX D ADDITIONAL INFORMATION FOR SERIES IV USERS

This appendix contains specific information on the Intellec Series IV development system. It covers the following subjects:

- Series IV program development process (and related manuals).
- Hardware and software required.
- User programs supported.
- System resources used by the debugger.
- System-specific examples of the debugger invocation line, sign-on message, and commands.

Operation of the Series IV

The following manuals describe the general operation of the Series IV:

- *Intellec® Series IV Microcomputer Development System Overview*, order number 121752
- *Intellec® Series IV Operating and Programming Guide*, order number 121753
- *Intellec® Series IV ISIS-IV User's Guide*, order number 121880

Program Development Process

Figure 1-1 shows how the debugger fits into your program development process. Figure D-1 shows the same process. The following manuals provide information that will aid in your program development.

- *ISIS-IV CREDIT™ CRT-Based Text Editor User's Guide*, order number 9800902
- *AEDIT TEXT EDITOR User's Guide*, order number 121756
- *Pascal 86 User's Guide*, order number 121539
- *PL/M-86 User's Guide for 8086-Based Development System*, order number 121636
- *FORTRAN-86 User's Guide*, order number 121570
- *iAPX 86,88 Family Utilities User's Guide*, order number 121616

Hardware and Software Required

You need the following hardware and software to run the debugger:

- Intellec Series IV development system.
- iNDX operating system.
- At least one single- or double-density flexible disk drive, a hard disk unit plus a single- or double-density 5 1/4 inch floppy, or a remote disk on an NDS I or NDS II.
- Pascal compiler (release 2.0 or later), PL/M-86 compiler (release 2.0 or later), or FORTRAN-86 compiler (release 2.0 or later).
- 8086-based utilities.
- PSCOPE high-level program debugger.

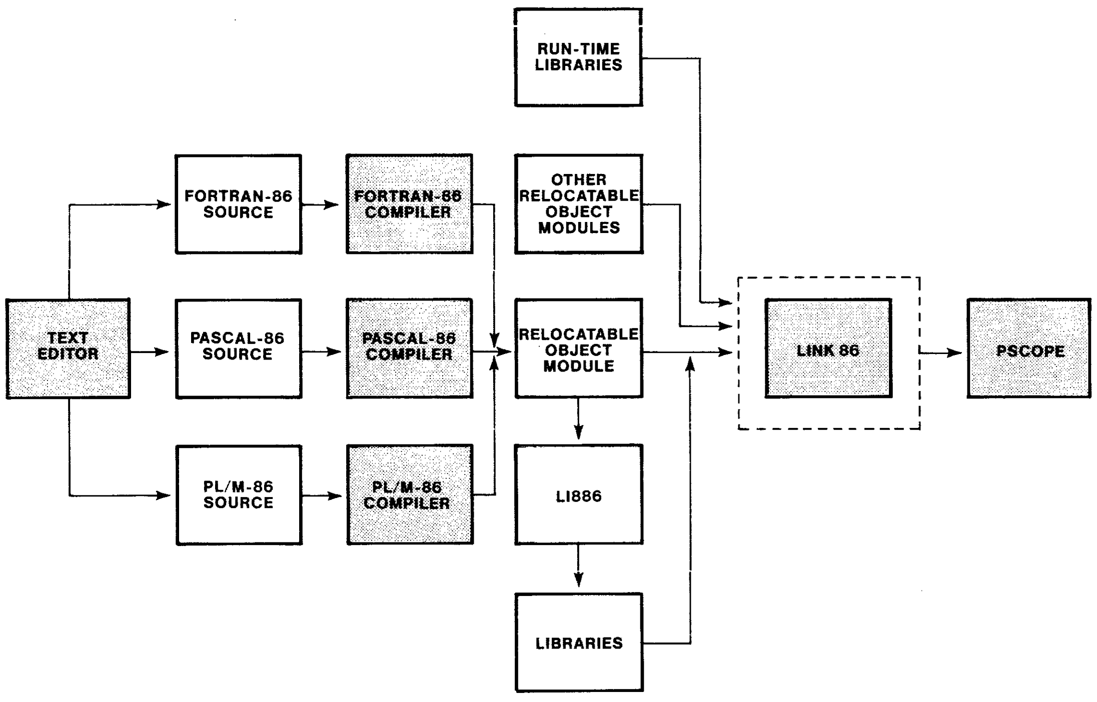

Figure D-1 Series IV Program Development Process

User Programs Supported

The amount of memory available to your program (the program under debug) depends upon the amount of memory in your system. You can expand the Series IV up to one megabyte of memory addressable by the 8086. PSCOPE requires approximately 110K bytes. You must add more memory to accommodate additional workspace and the user program.

Your program must be a load-time locatable (LTL) or a position independent code (PIC) object module produced by LINK86 with the BIND control. You must produce the object modules used as input to LINK86 with either a Pascal-86 compiler (release 2.0 or later), a PL/M-86 compiler (release 2.0 or later), or a FORTRAN-86 compiler (release 2.0 or later) with the DEBUG control. Because the reliability of some debug functions can be affected by cross-statement compiler optimizations, you must use OPTIMIZE (0).

System Resources Used

The debugger requires certain system resources, such as memory space and open files, that can affect your program.

Memory

The debugger occupies 110K bytes of memory, including space for symbol and line number information.

To reduce memory usage, the debugger provides for virtual storage of compiler-generated debug information. Symbol table information (from the compiler) will be sent out to disk if necessary. Your program must reside in memory, however.

File Requirements

Under the Series IV operating system, up to six open files are available for an application, plus terminal input. (Terminal output does not count as an open file.) Terminal input does not count against the total of six open files allowed because the operating system shares terminal input between PSCOPE and your program.

Of those six files, PSCOPE may require one or more files from each of the following groups:

- Terminal input
- Virtual symbol table
- LOAD, HELP, PUT, INCLUDE, CRT, MACRO, or PSCOPE overlay
- List

The number of open files increases if you have nested open files, such as a PUT command inside of an INCLUDE file.

Other Resources Required

The debugger requires the following additional host system resources:

- The software interrupt 3 (the one-byte, debugger-oriented INT instruction).
- The trap flag and interrupt 1 (used for single-stepping).
- The CNTL-C trap (system call DQ$TRAP$CC).

Your program or any background program should not use these host system resources.

In addition, PSCOPE uses interrupts 0, 4, 5, 16, 17, and 20 through 31 for error handling and floating point operations. Your program can use these interrupts, since PSCOPE maintains separate copies of these interrupt vectors for itself and your program. However, a background program must not use any of these interrupts.

Invocation Line

The general format of the invocation is as follows:

/W/ PSCOPE [*controls*]

or

PSCOPE [*controls*]

Where:

controls is any number of invocation controls from the list specified in Chapter 3.

PSCOPE signs on with the following message:

```
SERIES-IV PSCOPE-86, Vx.y
```

Example

The following example shows the beginning of a PSCOPE debugging session:

```
-PSCOPE MACRO(PROCS.MAC)

SERIES-IV PSCOPE-86, Vx.y
*LOAD DC.86
*GO
```

APPENDIX E SAMPLE PROGRAM LISTING

This appendix contains the sample program DC referred to throughout this manual.

```
SERIES-III Pascal-86, V2.0


Source File: :F7:DC7.PAS
Object File: :F7:DC7.OBJ
Controls Specified:  XREF, DEBUG.

STMT LINE NESTING        SOURCE TEXT: :F7:DC7.PAS
   1    1  0  0          (*  This program implements an interactive Desk Calculator.  It
                           accepts lines of text as input.  Each line should contain one
                           expression.  Each line is parsed, evaluated, and the result
                           is printed.  The expressions are allowed to contain embedded
                           assignment statements to single-letter variables.  An error
                           will abort the evaluation of the current expression. *)

                         program dc (input, output);

   2   10  0  0            label 1000, 9999;

   3   12  0  0            const max_line_length = 40;

   4   14  0  0            type  error_class = (illegal_token,   line_too_long,       end_of_line,
                                                missing_r_paren, error_in_expression, error_in_factor,
                                                error_in_statement,  error_in_term);

   5   18  0  0                  token_class = (add_op,  mul_op,   assign,    l_paren,
                                                r_paren, variable, int_const, line_end);

   6   21  0  0                  token = record
   6   22  0  1                            case class   : token_class of
   7   23  0  1                              add_op     : (add_op_value : char);
   8   24  0  1                              mul_op     : (mul_op_value : char);
   9   25  0  1                              assign     : ();
  10   26  0  1                              l_paren    : ();
  11   27  0  1                              r_paren    : ();
  12   28  0  1                              variable   : (variable_value  : char);
  13   29  0  1                              int_const  : (int_const_value : integer);
  14   30  0  1                              line_end   : ();
  15   31  0  1                            end (* record *);

  16   33  0  0                  text_buffer = record
  16   34  0  1                                  status     : (empty, full);
  17   35  0  1                                  length     : 0 .. max_line_length;
  18   36  0  1                                  index      : 0 .. max_line_length;
  19   37  0  1                                  last_index : 0 .. max_line_length;
  20   38  0  1                                  str        : packed array [1..max_line_length] of char
                                                end (* record *);

  21   41  0  0            var   t      : token;
  22   42  0  0                  c      : char;
  23   43  0  0                  buffer : text_buffer;

  24   45  0  0                  variable_table : array['a'..'z'] of integer;

  25   47  0  0            (* ---------------------------------------------------------------- *)
                           procedure error(e : error_class); (* print error & restart *)
  26   49  1  0            begin
  26   50  1  1              write (' ':(buffer.last_index+3), '^ DC Error: ');
  27   51  1  1              case e of
```

```
SERIES-III Pascal-86, V2.0
ERROR

STMT LINE NESTING       SOURCE TEXT: :F7:DC7.PAS
  28   52  1  2               illegal_token       : write ('Illegal token');
  29   53  1  2               line_too_long       : write ('Input line too long');
  30   54  1  2               end_of_line         : write ('End of line');
  31   55  1  2               missing_r_paren     : write ('Missing right paren');
  32   56  1  2               error_in_factor     : write ('Illegal factor');
  33   57  1  2               error_in_term       : write ('Error detected in term');
  34   58  1  2               error_in_expression : write ('Error detected in expression');
  35   59  1  2               error_in_statement  : write ('Illegal statement');
  36   60  1  2             end (* case *);
  38   61  1  1             writeln;
  39   62  1  1             goto 9999;
  40   63  1  1           end (* error *);

  41   65  0  0           (* ------------------------------------------------------------------ *)
                          procedure get_line; (* input line & set c to 1st char of line *)
  42   67  1  0           begin (* get_line *)
  42   68  1  1             buffer.length     := 0;
  43   69  1  1             buffer.status     := empty;
  44   70  1  1             buffer.last_index := 1;
  45   71  1  1             repeat
  45   72  1  2               write('    ');
  46   73  1  2               while eof do reset(input);
  48   74  1  2               while not eoln do
  49   75  1  2                 if buffer.length < max_line_length then begin
  50   76  1  3                   buffer.length := buffer.length + 1;
  51   77  1  3                   read(buffer.str[buffer.length])
                                 end
  52   79  1  2                 else error(line_too_long);
  54   80  1  2               readln;
  55   81  1  2             until buffer.length > 0;
  57   82  1  1             buffer.status := full;
  58   83  1  1             buffer.index  := 1;
  59   84  1  1             c             := buffer.str[buffer.index];
  60   85  1  1           end (* get_line *);

  61   87  0  0           (* ------------------------------------------------------------------ *)
                          procedure get_token; (* scan line & set t to its value *)

  62   90  1  0             function digit(c: char): boolean; (* true if c is a digit *)
  63   91  2  0             begin
  63   92  2  1               digit := ('0' <= c) and (c <= '9')
                            end;

  64   95  1  0             function upper_case(c: char): boolean; (* true if c is upper case *)
  65   96  2  0             begin
  65   97  2  1               upper_case := ('A' <= c) and (c <= 'Z')
                            end;

  66  100  1  0             function lower_case(c: char): boolean; (* true if c is lower case *)
  67  101  2  0             begin
  67  102  2  1               lower_case := ('a' <= c) and (c <= 'z')
                            end;

  68. 105  1  0             (* ---------------------------------- *)
                            procedure get_char; (* set c to next char in line *)
```

```
SERIES-III Pascal-86, V2.0
GET_CHAR

STMT LINE NESTING          SOURCE TEXT: :F7:DC7.PAS
  69  107  2  0                begin (* get_char *)
  69  108  2  1                  if buffer.status = empty then get_line
  70  109  2  1                  else begin
  71  110  2  2                    if buffer.index < buffer.length then begin
  72  111  2  3                      buffer.index := buffer.index + 1;
  73  112  2  3                      c            := buffer.str[buffer.index];
  74  113  2  3                    end
  75  114  2  2                    else begin
  76  115  2  3                      c             := cr;
  77  116  2  3                      buffer.status := empty
                                   end
  78  118  2  2                  end
  79  119  2  1                end (* get_char *);

  80  121  1  0              (* ----------------------------------- *)
                             begin (* get_token: scan line & set t to its value *)

  80  124  1  1                while c = ' ' do get_char;         (* skip leading spaces *)
  82  125  1  1                buffer.last_index := buffer.index; (* for error reporting *)

  83  127  1  1                if lower_case(c) then begin        (* lower case variable *)
  84  128  1  2                  t.class          := variable;
  85  129  1  2                  t.variable_value := c;
  86  130  1  2                  get_char;
  87  131  1  2                end

  88  133  1  1                else if upper_case(c) then begin   (* upper case variable *)
  90  134  1  2                  t.class          := variable;
  91  135  1  2                  t.variable_value := chr(ord(c) + (ord('a') - ord('A')));
  92  136  1  2                  get_char;
  93  137  1  2                end

  94  139  1  1                else if digit(c) then begin        (* integer constant *)
  96  140  1  2                  t.class          := int_const;
  97  141  1  2                  t.int_const_value := 0;
  98  142  1  2                  while digit(c) do begin
  99  143  1  3                    t.int_const_value := 10*t.int_const_value + ord(c) - ord('0');
 100  144  1  3                    get_char;
 101  145  1  3                  end;
 103  146  1  2                end

 104  148  1  1                else if c = cr then begin          (* end of line *)
 106  149  1  2                  t.class := line_end;
 107  150  1  2                  c       := lf;
 108  151  1  2                end

 109  153  1  1                else begin                         (* symbol: + - * / := ( ) # *)
 110  154  1  2                  case c of
 111  155  1  3                    '+', '-' : begin t.class := add_op; t.add_op_value := c; end;
 115  156  1  3                    '*', '/' : begin t.class := mul_op; t.mul_op_value := c; end;
 119  157  1  3                    ':'      : begin
 119  158  1  4                                 get_char;
 120  159  1  4                                 if c = '=' then t.class := assign
 121  160  1  4                                 else error(illegal_token);
 123  161  1  4                               end;
```

```
SERIES-III Pascal-86, V2.0
GET_TOKEN

STMT LINE NESTING       SOURCE TEXT: :F7:DC7.PAS
 125  162  1  3                '('      : t.class := l_paren;
 126  163  1  3                ')'      : t.class := r_paren;
 127  164  1  3                '#'      : goto 1000;
 128  165  1  3                otherwise  error(illegal_token);
 130  166  1  3              end (* case *);
 132  167  1  2              get_char;
 133  168  1  2            end (* begin *);

 135  170  1  1          end (* get_token *);

 136  172  0  0          (* ----------------------------------------------------------------- *)
                         procedure factor(var factor_value : integer);
 137  174  1  0          (* parse: <variable> [":=" <expression>] ! "("<expression>")" ! <number> *)
                           var  expression_value : integer;
 138  176  1  0                variable_index   : char;
 139  177  1  0          begin (* factor *)
 139  178  1  1            case t.class of
 140  179  1  2              variable : begin
 140  180  1  3                  variable_index := t.variable_value;
 141  181  1  3                  get_token;
 142  182  1  3                  if t.class <> assign then
 143  183  1  3                    factor_value := variable_table[variable_index]
                                 else begin
 144  185  1  4                    get_token;
 145  186  1  4                    expression(expression_value);
 146  187  1  4                    variable_table[variable_index] := expression_value;
 147  188  1  4                    factor_value := expression_value;
 148  189  1  4                  end;
 150  190  1  3                end;
 152  191  1  2              l_paren : begin
 152  192  1  3                  get_token;
 153  193  1  3                  expression(expression_value);
 154  194  1  3                  factor_value := expression_value;
 155  195  1  3                  if t.class = r_paren then
 156  196  1  3                    get_token
                                 else error(missing_r_paren);
 158  198  1  3                end;
 160  199  1  2              int_const : begin factor_value := t.int_const_value; get_token; end;
 164  200  1  2              otherwise error(error_in_factor);
 166  201  1  2            end (* case *);
 168  202  1  1          end (* factor *);

 169  204  0  0          (* ----------------------------------------------------------------- *)
                         procedure term(var term_value : integer);
 170  206  1  0          (* parse: <factor> [<mul_op> <factor>]... *)
                           var  factor_1_value : integer;
 171  208  1  0                factor_2_value : integer;
 172  209  1  0                op             : char;
 173  210  1  0          begin (* term *)
 173  211  1  1            factor (factor_1_value);
 174  212  1  1            while t.class = mul_op do begin
 175  213  1  2              op := t.mul_op_value;
 176  214  1  2              get_token;
 177  215  1  2              factor (factor_2_value);
 178  216  1  2              case op of
```

```
SERIES-III Pascal-86, V2.0
TERM


STMT LINE NESTING        SOURCE TEXT: :F7:DC7.PAS
 179  217  1  3                '*' : factor_1_value := factor_1_value * factor_2_value;
 180  218  1  3                '/' : factor_1_value := factor_1_value div factor_2_value;
 181  219  1  3                otherwise error (error_in_term);
 183  220  1  3              end (* case *);
 185  221  1  2            end;
 187  222  1  1            term_value := factor_1_value;
 188  223  1  1          end (* term *);

 189  225  0  0          (* ---------------------------------------------------------------- *)
                         procedure expression (var expression_value : integer);
 190  227  1  0          (* parse: [<add_op>] <term> [<add_op> <term>]... *)
                           var  term_1_value : integer;
 191  229  1  0                term_2_value : integer;
 192  230  1  0                op           : char;
 193  231  1  0          begin (* expression *)
 193  232  1  1            if t.class = add_op then begin
 194  233  1  2              op      := t.add_op_value;
 195  234  1  2              get_token;
 196  235  1  2            end
 197  236  1  1            else op := '+';
 199  237  1  1            term (term_1_value);
 200  238  1  1            case op of
 201  239  1  2              '+' : (* null *);
 202  240  1  2              '-' : term_1_value := -term_1_value;
 203  241  1  2              otherwise error(error_in_expression);
 205  242  1  2            end (* case *);
 207  243  1  1            while t.class = add_op do begin
 208  244  1  2              op := t.add_op_value;
 209  245  1  2              get_token;
 210  246  1  2              term (term_2_value);
 211  247  1  2              case op of
 212  248  1  3                '+' : term_1_value := term_1_value + term_2_value;
 213  249  1  3                '-' : term_1_value := term_1_value - term_2_value;
 214  250  1  3                otherwise error(error_in_expression);
 216  251  1  3              end (* case *);
 218  252  1  2            end;
 220  253  1  1            expression_value := term_1_value;
 221  254  1  1          end (* expression *);

 222  256  0  0          (* ---------------------------------------------------------------- *)
                         procedure statement;
 223  258  1  0          (* parse: <expression> <line_end> *)
                           var  expression_value : integer;
 224  260  1  0          begin (* statement *)
 224  261  1  1            expression (expression_value);
 225  262  1  1            if t.class = line_end then writeln(expression_value:1)
 226  263  1  1            else error(error_in_statement);
 228  264  1  1          end (* statement *);


 229  267  0  0        begin (* main program *)

 229  269  0  1        (* initialize variable table *)
                         for c := 'a' to 'z' do variable_table[c] := 0;
```

```
SERIES-III Pascal-86, V2.0


STMT LINE NESTING       SOURCE TEXT: :F7:DC7.PAS
 231  272  0  1         (* sign on *)
                          writeln ('Desk Calculator (DC)');

 232  275  0  1         (* error restart *)
                          9999:
                          repeat (* forever *)
 232  278  0  2             get_line;
 233  279  0  2             get_token;
 234  280  0  2             statement;
 235  281  0  2           until false;

 237  283  0  1         (* sign off *)
                          1000:
                          writeln ('Exit');

 238  287  0  1         end.
```

```
SERIES-III Pascal-86, V2.0
                                        Cross-Reference Listing


Name                   Offset  Length     Attributes and References
--------------------   ------  ------     -------------------------

1000 . . . . . . . .                      label in DC at 237; read: 127.
9999 . . . . . . . .                      label in DC at 232; read: 39.
ADD_OP . . . . . . .                1     TOKEN_CLASS constant in DC at 5; read: 7 111 193 207.
ASSIGN . . . . . . .                1     TOKEN_CLASS constant in DC at 5; read: 9 121 142.
BOOLEAN. . . . . . .                1     primitive type; read: 62 64 66.
BUFFER . . . . . . .      14H      44     TEXT_BUFFER variable in DC at 23; write: 42 43 44 50 51 57 58 72 77 82;
                                           read: 26 49 50 51 56 59 59 69 71 71 72 73 73 82.
C. . . . . . . . . .      74H       1     CHAR variable in DC at 22; write: 59 73 76 107 229; read: 80 83 85 89 91 95
                                          98 99 105 110 112 116 120 230.
C. . . . . . . . . .       4H       1     CHAR parameter in DIGIT at 62; read: 63 63.
C. . . . . . . . . .       4H       1     CHAR parameter in LOWER_CASE at 66; read: 67 67.
C. . . . . . . . . .       4H       1     CHAR parameter in UPPER_CASE at 64; read: 65 65.
CHAR . . . . . . . .                1     primitive type; read: 7 8 12 20 22 62 64 66 138 172 192.
CR . . . . . . . . .                1     predefined CHAR constant; read: 76 105.
DIGIT. . . . . . . .                      BOOLEAN function in GET_TOKEN at 62; write: 63; read: 95 98.
E. . . . . . . . . .       4H       1     ERROR_CLASS parameter in ERROR at 25; read: 27.
EMPTY. . . . . . . .                1     (EMPTY,...,FULL) constant in DC at 16; read: 43 69 77.
END_OF_LINE. . . . .                1     ERROR_CLASS constant in DC at 4; read: 30.
ERROR. . . . . . . .                      procedure in DC at 25; read: 53 122 129 157 165 182 204 215 227.
ERROR_CLASS. . . . .                1     (ILLEGAL_TOKEN,...,ERROR_IN_TERM) type in DC at 4; read: 25.
ERROR_IN_EXPRESSION.                1     ERROR_CLASS constant in DC at 4; read: 34 204 215.
ERROR_IN_FACTOR. . .                1     ERROR_CLASS constant in DC at 4; read: 32 165.
ERROR_IN_STATEMENT .                1     ERROR_CLASS constant in DC at 4; read: 35 227.
ERROR_IN_TERM. . . .                1     ERROR_CLASS constant in DC at 4; read: 33 182.
EXPRESSION . . . . .                      procedure in DC at 189; read: 145 153 224.
EXPRESSION_VALUE . .       4H       2     INTEGER var parameter in EXPRESSION at 189; write: 220.
EXPRESSION_VALUE . .    FFFCH       2     INTEGER variable in FACTOR at 137; write: 145 153; read: 146 147 154.
EXPRESSION_VALUE . .    FFFCH       2     INTEGER variable in STATEMENT at 223; write: 224; read: 226.
FACTOR . . . . . . .                      procedure in DC at 136; read: 173 177.
FACTOR_1_VALUE . . .    FFFCH       2     INTEGER variable in TERM at 170; write: 173 179 180; read: 179 180 187.
FACTOR_2_VALUE . . .    FFFAH       2     INTEGER variable in TERM at 171; write: 177; read: 179 180.
FACTOR_VALUE . . . .       4H       2     INTEGER var parameter in FACTOR at 136; write: 143 147 154 160.
FALSE. . . . . . . .                1     predefined BOOLEAN constant; read: 236.
FULL . . . . . . . .                1     (EMPTY,...,FULL) constant in DC at 16; read: 57.
GET_CHAR . . . . . .                      procedure in GET_TOKEN at 68; read: 81 86 92 100 119 132.
GET_LINE . . . . . .                      procedure in DC at 41; read: 70 232.
GET_TOKEN. . . . . .                      procedure in DC at 61; read: 141 144 152 156 161 176 195 209 233.
ILLEGAL_TOKEN. . . .                1     ERROR_CLASS constant in DC at 4; read: 28 122 129.
INPUT. . . . . . . .       8H       8     predefined TEXT variable; write: 47; read: 46 48 51 54.
INTEGER. . . . . . .                2     primitive type; read: 13 24 136 137 169 170 171 189 190 191 223.
INT_CONST. . . . . .                1     TOKEN_CLASS constant in DC at 5; read: 13 96 160.
LF . . . . . . . . .                1     predefined CHAR constant; read: 107.
LINE_END . . . . . .                1     TOKEN_CLASS constant in DC at 5; read: 14 106 225.
LINE_TOO_LONG. . . .                1     ERROR_CLASS constant in DC at 4; read: 29 53.
LOWER_CASE . . . . .                      BOOLEAN function in GET_TOKEN at 66; write: 67; read: 83.
L_PAREN. . . . . . .                1     TOKEN_CLASS constant in DC at 5; read: 10 125 152.
MAX_LINE_LENGTH. . .                2     INTEGER constant in DC at 3; read: 17 18 19 20 49.
MISSING_R_PAREN. . .                1     ERROR_CLASS constant in DC at 4; read: 31 157.
MUL_OP . . . . . . .                1     TOKEN_CLASS constant in DC at 5; read: 8 115 174.
OP . . . . . . . . .    FFF9H       1     CHAR variable in EXPRESSION at 192; write: 194 198 208; read: 200 211.
OP . . . . . . . . .    FFF9H       1     CHAR variable in TERM at 172; write: 175; read: 178.
OUTPUT . . . . . . .       0H       8     predefined TEXT variable; read: 26 28 29 30 31 32 33 34 35 38 45 226 231
```

```
SERIES-III Pascal-86, V2.0
                                          Cross-Reference Listing


                                               237.
R_PAREN. . . . . . .                     1     TOKEN_CLASS constant in DC at 5; read: 11 126 155.
STATEMENT. . . . . .                           procedure in DC at 222; read: 234.
T. . . . . . . . . .         75H         3     TOKEN variable in DC at 21; write: 84 85 90 91 96 97 99 106 111 112 115 116
                                               121 125 126; read: 99 139 140 142 155 160 174 175 193 194 207 208 225.
TERM . . . . . . . .                           procedure in DC at 169; read: 199 210.
TERM_1_VALUE . . . .       FFFCH         2     INTEGER variable in EXPRESSION at 190; write: 199 202 212 213; read: 202
                                               212 213 220.
TERM_2_VALUE . . . .       FFFAH         2     INTEGER variable in EXPRESSION at 191; write: 210; read: 212 213.
TERM_VALUE . . . . .          4H         2     INTEGER var parameter in TERM at 169; write: 187.
TEXT_BUFFER. . . . .                    44     record type in DC at 16; read: 23.
TOKEN. . . . . . . .                     3     record type in DC at 6; read: 21.
TOKEN_CLASS. . . . .                     1     (ADD_OP,...,LINE_END) type in DC at 5; read: 6.
UPPER_CASE . . . . .                           BOOLEAN function in GET_TOKEN at 64; write: 65; read: 89.
VARIABLE . . . . . .                     1     TOKEN_CLASS constant in DC at 5; read: 12 84 90 140.
VARIABLE_INDEX . . .       FFFBH         1     CHAR variable in FACTOR at 138; write: 140; read: 143 146.
VARIABLE_TABLE . . .         40H        52     array[ 'a' .. 'z' ] of INTEGER variable in DC at 24; write: 146 230; read:
                                               143.


Summary Information:

PROCEDURE                  OFFSET    CODE SIZE      DATA SIZE      STACK SIZE
ERROR                      00C8H   014CH    332D                  000EH     14D
GET_LINE                   0214H   00BEH    190D                  0012H     18D
GET_TOKEN                  0399H   0147H    327D                  0008H      8D
DIGIT                      02D2H   002CH     44D                  0008H      8D
UPPER_CASE                 02FEH   002CH     44D                  0008H      8D
LOWER_CASE                 032AH   002CH     44D                  0008H      8D
GET_CHAR                   0356H   0043H     67D                  0008H      8D
FACTOR                     04E0H   00B4H    180D                  000EH     14D
TERM                       0594H   007BH    123D                  0010H     16D
EXPRESSION                 060FH   00B7H    183D                  0010H     16D
STATEMENT                  06C6H   0036H     54D                  000CH     12D
DC                         06FCH   00E7H    231D   0078H    120D  000EH     14D
-CONST IN CODE-                    00C8H    200D

Total                              07E3H   2019D   0078H    120D  00C4H    196D

 287 Lines Read.
   0 Errors Detected.
 44% Utilization of Memory.
```

APPENDIX F
PSCOPE GRAMMAR

This appendix contains the grammar that describes the syntax of PSCOPE's command language. "Notational Conventions" in the Preface to this manual explains the notational conventions used.

Note that the command line is the unit in which PSCOPE commands are processed. Hence, the symbol *command-line* is the start symbol of the grammar.

PSCOPE Grammar

```
command-line ::= [command] [; command]*

    command ::= asm-command
              |base-command
              |callstack-command
              |count-command
              |define-command
              |directory-command
              |display-command
              |do-command
              |edit-command
              |eval-command
              |exit-command
              |go-command
              |help-command
              |if-command
              |inputmode-command
              |include-command
              |list-command
              |load-command
              |modify-command
              |port-command
              |put-command
              |remove-command
              |repeat-command
              |return-command
              |step-command
              |view-command
              |write-command

asm-command ::= ASM  ADDRESS = 'assembler-mnemonic
                       '[,'assembler-mnemonic]*
           |SASM     ADDRESS = 'assembler-mnemonic
                       '[,'assembler-mnemonic]*
           |ASM      ADDRESS [length-spec]
           |SASM     ADDRESS [length-spec]

base-command ::= BASE [=expr]

callstack-command ::= CALLSTACK [expr]
```

```
count-command ::= COUNT expr
                     [loop-command]*
                  end-count

  loop-command ::= WHILE expr
                  |UNTIL expr
                  |[command]

  end-count ::= ENDCOUNT
               |END

define-command ::= DEFINE BRKREG name = break-group [, break-group]*
                  |DEFINE TRCREG name = break-group [, break-group]*
                  |DEFINE PATCH expr [TIL expr] = patch-value
                  |DEFINE PROC name = command
                  |DEFINE LITERALLY name = string [string]*
                  |DEFINE [GLOBAL] mtype name [= expr]

  break-group ::= (break-point [, break-point]*) [CALL proc-name]
                 |break-point [CALL proc-name]

    break-point ::= expr

  patch-value ::= command
                 |NOP

directory-command ::= DIR [directory] [directory-type]

  directory ::= DEBUG
               |PUBLIC
               |: module-name

  directory-type ::= mtype
                    |dtype
                    |PATCH
                    |ARRAY
                    |ENUMERATION
                    |FILE
                    |LABEL
                    |LINE
                    |MODULE
                    |PROCEDURE
                    |RECORD
                    |SET

display-command ::= PROC proc-name
                   |LITERALLY literally-name
                   |BRKREG brkreg-name
                   |TRCREG trcreg-name
                   |PATCH expr
                   |mtype address [length-spec]
                   |expr
                   |REGS

  length-spec ::= LENGTH expr
                 |TO expr

do-command ::= DO
                  [command]*
               END
```

```
edit-command ::= EDIT [edit-item]

  edit-item ::= name
              |PATCH expr
              |GO

eval-command ::= EVAL expr [eval-type]

  eval-type ::= LINE
              |PROCEDURE
              |SYMBOL [ALL]

exit-command ::= EXIT

go-command ::= GO [break-spec]*
             |GO FOREVER

  break-spec ::= USING brkreg-item [, brkreg-item]*
               |TIL expr [, expr]*

    brkreg-item ::= BRKREG
                  |brkreg-name
                  |TRCREG
                  |trcreg-name

help-command ::= HELP [name]

if-command ::= IF expr THEN
                  [command]*
               [ORIF expr THEN
                  [command]*]*
               [ELSE
                  [command]*]
               end-if

  end-if ::= ENDIF
           |END


include-command ::= INCLUDE pathname [NOLIST]

inputmode-command ::= INPUTMODE [=N]

list-command ::= LIST [pathname]
               |NLIST

load-command ::= LOAD pathname [load-option]*
                 [CONTROLS controls-text]

load-option ::= NOLINES
              |NOSYMBOLS
              |CH8087              /*for iRMX*/
              |E8087               /*for Series III/Series IV*/

port-command ::= PORT port-# expr [=expr]
               |WPORT port-# expr [=expr]

modify-command ::= variable = expr
                 |mtype address [length-spec] = modify-list
    modify-list ::= expr [, expr]*
                  |mtype address [length-spec]

      length-spec ::= LENGTH expr
                    |TO expr

  put-command ::= PUT pathname put-list
                |APPEND pathname put-list

    put-list ::= put-item [, put-item]*
              |DEBUG

      put-item ::= mtype
                 |dtype
                 |name
                 |PATCH [expr]

  remove-command ::= REMOVE remove-item [, remove-item]*
                   |REMOVE DEBUG

    remove-item ::= mtype
                  |dtype
                  |name
                  |PATCH [expr]

  repeat-command ::= REPEAT
                       [loop-command]*
                     end-repeat

    loop-command ::= WHILE expr
                   |UNTIL expr
                   |[command]

    end-repeat ::= ENDREPEAT
                 |END

  return-command ::= RETURN [expr]

  step-command ::= LSTEP
                 |PSTEP
                 |ISTEP

  view-command ::= VIEW view-item

    view-item ::= file-name

  write-command ::= WRITE
                      [USING (string-expr)]
                      [expr [, expr]*]

    expr ::= logic-term [or-op logic-term]*

      or-op ::= OR
             |XOR

      logic-term ::= logic-factor [AND logic-factor]*

        logic-factor ::= [NOT] logic-primary

          logic-primary ::= arith-expr [relational-op arith-expr]
```

```
relational-op ::= <
                 |>
                 |==
                 |<=
                 |>=
                 |<>

arith-exp ::= [mtype] address

   address ::= term [add-op term]*

      add-op ::= +
                |-
                |/
                |MOD

      term ::= factor [mult-op factor]*

         factor ::= [add-op] primary

            primary ::= primitive [: primitive]

               primitive ::= (expr)
                            |variable
                            |value

               variable ::= symbolic-reference
                           |mtype-variable-name
                           |$
                           |BASE
                           |NAMESCOPE
                           |reg-name

                  symbolic-reference ::= [: module-name .]
                                          symbol [qualifier]*
                                        |[: module-name]
                                          #line-number

                   |symbol ::= ["] name

                   qualifier ::= left-bracket expr [, expr]*
                                  right-bracket
                                |. symbol
                                |↑

                      left-bracket ::= [

                      right-bracket ::= ]
```

```
                     reg-name ::= AX    /* for iRMX only */
                     |BX                |ST0
                     |CX                |ST1
                     |DX                |ST2
                     |BP                |ST3
                     |SP                |ST4
                     |DI                |ST5
                     |SI                |ST6
                     |CS                |ST7
                     |DS                |FIA
                     |ES                |FIO
                     |SS                |FCW
                     |IP                |FDA
                     |FLAGS
                     |AL
                     |AH
                     |BL
                     |BH
                     |CL
                     |CH
                     |DL
                     |DH
                     |FL
                     |FH

                   value ::= integer-constant
                         |real-constant
                         |Boolean-constant
                         |string-constant [string-constant]*
                         |proc-name [(expr [, expr]*)]
                         |% actual-parameter
                         |SUBSTR (string-expr, expr, expr)
                         |CONCAT (string-expr[, string-expr]*)
                         |STRLEN (string-expr)
                         |CI
                         |ACTIVE (symbolic-reference)
                         |. symbolic-reference
                         |SELECTOROF (expr)
                         |OFFSETOF (expr)
                     |actual-parameter ::= integer-constant
                                        |(expr)
                                        |NP


  dtype ::= LITERALLY
          |BRKREG
          |TRCREG
          |PROC
```

```
mtype ::= BOOLEAN
         |CHAR
         |BYTE
         |WORD
         |DWORD
         |ADDRESS
         |SELECTOR
         |POINTER
         |SHORTINT
         |INTEGER
         |LONGINT
         |EXTINT
         |BCD
         |REAL
         |LONGREAL
         |TEMPREAL
```

APPENDIX G RESERVED KEYWORDS

This appendix contains the keywords PSCOPE recognizes and uses. You cannot use keywords as user-defined object names. To reference a program symbol whose name is the same as a PSCOPE keyword, you must prefix the symbol with a quotation mark ("), as discussed in Chapter 3. PSCOPE also recognizes special operators and delimiters which, like the reserved keywords, you cannot use in any other way. PSCOPE reports all attempts to incorrectly use a PSCOPE keyword or delimiter as syntax errors.

PSCOPE Keywords

A	ACTIVE	ADDRESS	AFL
AH	AL	ALL	AND
APPEND	ARRAY	ASM	AX
BASE	BCD	BH	BL
BOOLEAN	BP	BRKREG	BX
BYTE	CALL	CALLSTACK	CFL
CH8087	CH	CHAR	CI
CL	CONCAT	CONTROLS	COUNT
CS	CX	DEBUG	DEFINE
DFL	DH	DI	DIR
DL	DO	DS	DWORD
DX	E8087	EDIT	ELSE
END	ENDCOUNT	ENDIF	ENDREPEAT
ENUMERATION	ES	EVAL	EXIT
EXTINT	FALSE	FCW	FH
FDA	FIA	FILE	FIO
FL	FLAG	FOREVER	FSW
GLOBAL	GO	HELP	IF
IFL	INCLUDE	INPUTMODE	INTEGER
IP	ISTEP	LABEL	LENGTH
LINE	LIST	LITERALLY	LOAD
LONGINT	LONGREAL	LSTEP	MOD
MODE	MODULE	NAMESCOPE	NOCODE
NOLINES	NOLIST	NOSYMBOLS	NOP
NOT	NP	OFFSET$OF	OFFSETOF
OFL	OR	ORIF	PATCH
PFL	POINTER	PORT	PROC
PROCEDURE	PSTEP	PUBLIC	PUT
REAL	RECORD	REGS	REMOVE
REPEAT	RETURN	SASM	SELECTOR
SELECTOR$OF	SELECTOROF	SET	SFL
SHORTINT	SI	SP	SS
ST0	ST1	ST2	ST3
ST4	ST5	ST6	ST7
STRLEN	SUBSTR	SYMBOL	TEMPREAL
TFL	THEN	TIL	TO
TRACEACT	TRACEREGS	TRCREG	TRUE
UNTIL	USING	VIEW	VSTB
VSTBUFFER	WHILE	WORD	WPORT
WRITE	XOR	ZFL	

PSCOPE Operators and Delimiters

```
;    ,    :    $    "    ↑    =    .    [    ]    (    )    %

<=    <    >    >=    <>    ==    +    -    *    /

/*    */
```

APPENDIX H PSCOPE COMMAND INDEX

This appendix lists each PSCOPE command and refers you to the section in this manual where you can find more information.

Command **Page**

% 8-3

ACTIVE 5-11
APPEND 7-6
ASM 5-13

BASE 10-9
B(lock) 2-11
BRKREG 11-1

call debug procedures 8-2
CALLSTACK 10-4
change 8086/8088 flags 5-5
change 8086/8088 registers 5-7
change 8087 register 5-8
change NAMESCOPE 5-10
change program symbol 5-5
CI 10-13
CONCAT 10-13
COUNT 6-1
CRT 2-1

debug procedure calls 8-2
debug procedure parameters 8-3
debug procedure return 8-2
D(elete) 2-11
DEFINE 7-2
DEFINE debug procedure 8-1
DIR 10-2
DISPLAY 7-3
display debug objects 7-3
display memory 5-11
display program symbol 5-3
DO 6-3

EDIT 2-8
EVAL 10-7
EXIT 10-1

fully qualified reference 5-1

G(et) 2-11
GO 4-2, 11-4

HELP 10-5, A-1

IF 6-2

Command **Page**

INCLUDE 10-6
INPUTMODE 10-10
I(nsert) 2-11
ISTEP 4-4

line number reference 3-3
LIST 10-6
LOAD 4-1
LSTEP 4-3

MACRO 2-2
memory reference 3-9
MODIFY 7-5
modify memory 5-12
modify debug symbol 7-5

NAMESCOPE 5-1, 5-10
NOCRT 2-1
NOLIST 10-6
NOMACRO 2-2
NOSUBMIT 2-3

OFFSET$OF 10-15

PATCH 9-1
PSTEP 4-3
PUT 7-6

Q(uit) 2-12

REGS 5-8
read and write I/O ports 5-9
REPEAT 6-1
REMOVE 7-5
RUN 2-1

SASM 5-13
SELECTOR$OF 10-15
SLA 5-15
STRLEN 10-13
SUBMIT 2-3
SUBSTR 10-13
symbolic reference 3-9

TRCREG 11-3

V(iew) 2-12, 2-13

Command	**Page**
VSTBUFFER	2-4
WRITE	10-12
X(change)	2-12

APPENDIX I PSCOPE UNDER THE iRMX™ OPERATING SYSTEM

PSCOPE runs on any 86/3xx microcomputer system running the iRMX-86 operating system, release 5 or greater. PSCOPE requires at least 110K bytes of user memory. Your application program requires additional memory.

The following major iRMX-86 subsystems must be present.

Nucleus
Basic I/O system
Extended I/O system
Human interface
Application loader
Universal Development Interface (UDI)

When running under the iRMX-86 operating system, PSCOPE differs in the following areas.

Linking
Invocation
Multitasking support
8087 support
CNTL-C limitation

Linking

Be sure to use the link library files supplied with the iRMX-86 operating system.

When invoking link86, include the following two options during the final link.

```
SEGSIZE(STACK(+2048))
MEMPOOL(+2500,0FF000H)
```

Invocation

Because systems that run iRMX-86 are 8086-based, you do not use the RUN program to invoke PSCOPE. A typical invocation line is as follows:

pathname PSCOPE.86 *options*

Where:

pathname specifies the path of the file back to the root directory.

For example, /user/world/prog/pscope.86 means that PSCOPE is a file in the directory *prog* which itself is in the directory *world* which itself is in the directory user which is a directory under the root.

Multitasking Support

Although the iRMX-86 operating system handles multiple tasks, PSCOPE can only debug one task at a time. PSCOPE has no cognizance of tasks initiated by the task being debugged.

Because PSCOPE directly manipulates the iRMX-86 interrupt vector table, no other task may manipulate the interrupt vector table while PSCOPE is running. When you invoke PSCOPE, PSCOPE saves the interrupt vector table. When you exit PSCOPE, PSCOPE restores the interrupt vector table. The consequence of this is that in a multi-station system, only one station at a time may run PSCOPE.

If PSCOPE hangs, you cannot be sure of what the interrupt vector table may hold. Reboot the system to return the interrupt vector table to a known state.

8087 Support

If the program you want to debug performs real arithmetic, the microcomputer system must contain the iSBC 337 MULTIMODULE (8087 hardware). You cannot use the 8087 software emulator.

When you load your program with the LOAD command, include the CH8087 option, as in the following example:

```
*LOAD pager.86 CH8087
```

Other Resources Required

The debugger requires the following additional host resources:

- The software interrupt 3 (the one-byte, debugger-oriented INT instruction).
- The trap flag and interrupt 1 (used for single-stepping).
- The CNTL-C trap (system call DQ$TRAP$CC).

Your program should not use these host system resources.

In addition, PSCOPE uses interrupts 0, 4, 5, 16, 17, and 20 through 31 for error handling and floating point operations. However, your program can use these interrupts since PSCOPE maintains separate copies of these interrupt vectors for itself and your program.

INDEX

&, 2-6, 3-1
$, 3-2, 11-5
%, 8-3

Accessing debug procedure parameters, 8-3, 8-4
ACTIVE function, 5-11
ADDS Regent Model 200 (2400 baud only)
  configuration, B-5
ADDS Viewpoint 3A Plus configuration, B-6
APPEND command, 7-6 thru 7-8

BASE command, 10-9, 10-10
BLOCK command, 2-11, 2-12
Break registers, 11-1 thru 11-3
Beehive Mini-Bee configuration, B-7
Breakpoints, 11-1 thru 11-7
BRKREG command, 11-1 thru 11-3

Calling debug procedures, 3-11, 8-2
Calling sequence, see CALLSTACK command
CALLSTACK command, 10-4, 10-5
Change 8086/8088 flags, 5-5, 5-6
Change 8086/8088 registers, 5-7, 5-8
Change 8087 registers, 5-8, 5-9
Change name scope, 5-10, 5-11
Change program symbol, 5-5
Character string constants, 3-4, 3-5
CI command, 10-13 thru 10-15
Code patch display, 9-2, 9-3
Code patches, 9-1 thru 9-3
Command entry, 2-6
Command index, H-1, H-2
Commands, see PSCOPE commands
Comments, 3-6
Compile restrictions, 1-2
CONCAT command, 10-13 thru 10-15
Configuration commands, B-1 thru B-3
Configuration
  ADDS Regent Model 200 (2400 baud only), B-5
  ADDS Viewpoint 3A Plus, B-6
  Beehive Mini-Bee, B-7
  DEC VT52, B-8
  DEC VT100, B-9
  Hazeltine 1420, B-10
  Hazeltine 1510 (Tilde Lead-in), B-11
  Hazeltine 1510 (ESC Lead-in), B-12
  Intel Series IIIE, B-13
  Lear Siegler ADM-3A, B-14
  Televideo 910 Plus, B-15
  Televideo 925 and 950, B-16
  Zentec, B-17
Configuring PSCOPE for non-Intel
  terminals, B-1 thru B-17
Constants, 3-9

Continuation flags
  &, 2-6, 3-1
  !, 2-9
Control constructs
  COUNT, 6-1, 6-2
  DO, 6-3
  IF, 6-2, 6-3
  REPEAT, 6-1, 6-2
COUNT control construct, 6-1, 6-2
CRT invocation control, 2-1, 2-2
Current name scope, 5-1 thru 5-3
Cursor control keys (for edit), 2-6, 2-7, 2-9 thru 2-11

DEC VT52 configuration, B-8
DEC VT100 configuration, B-9
Debug objects, 7-1, 7-2
  debug type, 7-2
  memory type, 7-2
Debug parameter reference, 3-11
Debug procedure calls, 3-11, 8-2
Debug procedure definitions, 8-1
Debug procedure parameter access, 8-3, 8-4
Debug procedure return, 8-2, 8-3
Debug session example, 1-6 thru 1-18
Debug symbol object types, 3-3
Debug symbols, 3-2
Debug type debug objects, 7-2
Debug variable references, 3-10
Debugger invocation, 2-1
Debugging session termination, see EXIT command
DEC VT52 configuration, B-8
DEC VT100 configuration, B-9
DEFINE BRKREG command, 11-1 thru 11-3
DEFINE code patches, 9-1, 9-2
DEFINE command, 7-2, 7-3, 8-1
DEFINE debug procedures, 8-1
DEFINE PATCH command, 9-1, 9-2
DEFINE PROC command, 8-1
DEFINE TRCREG command, 11-3, 11-4
DELETE (D) command, 2-12
Delimiters, 3-1, G-2
DIR command, 10-2 thru 10-4
Directory command, see DIR command
Display code patch, 9-2, 9-3
DISPLAY command, 7-3, 7-4
Display information at the terminal,
  see WRITE command
Display memory, 5-11, 5-12
Display program symbol, 5-3, 5-4
DO control construct, 6-3

EDIT command, 2-8
Editor display, 2-9
Editor, see internal screen-oriented editor

End a debugging session, see EXIT command
Enter PSCOPE commands, 2-1, 2-2 thru 2-6
Error messages, A-1 thru A-7
Error messages, invocation, A-1
Errors in syntax, 2-7
ESC key to invoke the internal editor, 2-8
EVAL command, 10-7 thru 10-9
Evaluating expressions, see EVAL command
Example PSCOPE debug session, 1-6 thru 1-18
Exception trapping, 11-6, 11-7
Execute user program, see GO command
EXIT command, 10-1, 10-2
Explicit typing of memory references,
  see memory references with explicit typing
Expression evaluation, see EVAL command
Expressions, 3-8

Fatal errors, A-1
Floating point numbers, 3-4
Fully qualified line references, 5-1 thru 5-3
Fully qualified name, 5-1 thru 5-3
Fully qualified references, 5-1 thru 5-3
Further reading, vi, vii

GET (G) command, 2-12
Global debug objects, 7-1
GO command, 4-2, 4-3, 11-4 thru 11-6
Grammar, PSCOPE, F-1 thru F-7

Hazeltine 1420 configuration, B-10
Hazeltine 1510 (Tilde Lead-in) configuration, B-11
Hazeltine 1510 (ESC Lead-in) configuration, B-12
HELP command, 10-5, 10-6, A-1
Host system execution environment, 1-2

IF control construct, 6-2, 6-3
INCLUDE command, 10-6, 10-7
Index of PSCOPE commands, H-1
INPUTMODE command, 10-10 thru 10-12
iRMX information, I-1
INSERT (I) command, 2-12, 2-13
Integers, 3-3, 3-4
Intellec Series III information, C-1
Intellec Series IV information, D-1
Intel Series III E configuration, B-13
Internal errors, A-1
Internal screen-oriented editor, 2-7 thru 2-11
  Cursor control, 2-9 thru 2-11
  Commands, 2-9
  Display, 2-9
  Entering, 2-8
  Exiting, 2-8
Invocation controls, 2-1 thru 2-5
  CRT | NOCRT, 2-1, 2-2
  MACRO | NOMACRO, 2-2, 2-3
  NOSUBMIT | SUBMIT, 2-3
  VSTBUFFER, 2-4
Invocation error messages, 2-4, 2-5
Invoking the debugger, 2-1

ISTEP command, 4-4

Keys
  Cursor control during edit, 2-9 thru 2-11
  Line editing, 2-6, 2-7
Keywords, 3-2, 3-3, G-1

Lear Siegler ADM-3A configuration, B-14
Line editing keys, 2-6, 2-7
Line number references, 3-3, 3-10
Line numbers, 3-3
LIST command, 10-6
Listing file, see LIST command
LOAD command, 4-1, 4-2
Local debug objects, 7-1
LSTEP command, 4-3, 4-4

Machine register references, 3-9
MACRO invocation control, 2-2, 2-3
Memory display, 5-11, 5-12
Memory modification, 5-12, 5-13
Memory references with explicit typing, 3-9, 3-10
Memory symbol object types, 7-5, 7-6
Memory type debug objects, 7-2
Memory type differences, 3-7
Menu commands, see PSCOPE menu commands
MODIFY command, 7-5
Modify memory, 5-12 thru 5-14

Name scope, 5-1 thru 5-3
Name scope change, 5-10, 5-11
Names, 3-2, 3-3
NOCRT invocation control, 2-1, 2-2
NOLIST command, 10-6
NOMACRO invocation control, 2-2, 2-3
NOSUBMIT invocation control, 2-3
Notational conventions, vii, viii
Number base, see BASE command
Numeric constants, 3-3, 3-4

Object file, 7-6 thru 7-8
OFFSET$OF command, 10-15
Operands, 3-8 thru 3-11
Operators, 3-5, 3-11, 3-12, G-2
OPTIMIZE(O) compiler option, 1-2 thru 1-4

Parameter accessing, 8-3, 8-4
Parameter references, 3-11
Partially qualified line references, 3-3, 5-3
Pascal program example,
  see sample PSCOPE debug session
PATCH command, 9-1 thru 9-3
Patch definition, 9-1, 9-2
Patch removal, 9-3
PL/M program example,
  see example PSCOPE debug session
Precedence of operators, see operators
PROC command, 8-1
Procedure calls, 3-11, 8-2

Product definition, 1-1
Program development process, 1-1, 1-2
Program example, E-1 thru E-8
Program execution, see GO command
Program symbol change, 5-5
Program symbol display, 5-3, 5-4
Program symbol references, 3-9, 5-1 thru 5-3
Program symbols, 3-2
PSCOPE command entry, 2-6
PSCOPE command index, H-1
PSCOPE commands
  Accessing debug procedure parameters, 8-3, 8-4
  ACTIVE, 5-11
  APPEND, 7-6 thru 7-8
  ASM, 5-13 thru 5-19
  BASE, 10-9, 10-10
  BLOCK, 2-11, 2-12
  BRKREG, 11-1 thru 11-3
  CALLSTACK, 10-4, 10-5
  Change name scope, 5-10, 5-11
  Change program symbol, 5-5
  Change 8086/8088 flags, 5-5, 5-6
  Change 8086/8088 registers, 5-7, 5-8
  Change 8087 registers, 5-8, 5-9
  CI, 10-13 thru 10-15
  CONCAT, 10-13 thru 10-15
  COUNT construct, 6-1, 6-2
  Debug procedure calls, 8-2
  Debug procedure return, 8-2, 8-3
  DEFINE, 7-2, 7-3
  DEFINE BRKREG, 11-1 thru 11-3
  DEFINE debug procedure, 8-1
  DEFINE PATCH, 9-1, 9-2
  DEFINE PROC, 8-1
  DEFINE TRCREG, 11-3, 11-4
  DELETE, 2-12
  DIR, 10-2 thru 10-4
  DISPLAY, 5-11, 5-12, 7-3, 7-4
  Display PATCH, 9-2, 9-3
  Display program symbol, 5-3, 5-4
  DO construct, 6-3
  EDIT, 2-8
  EVAL, 10-7 thru 10-9
  Exception trapping, 11-6, 11-7
  EXIT, 10-1, 10-2
  Fully qualified references, 5-1 thru 5-3
  GET, 2-12
  GO, 4-2, 4-3, 11-4 thru 11-6
  HELP, 10-5, 10-6, A-1
  IF construct, 6-2, 6-3
  INCLUDE, 10-6, 10-7
  INPUTMODE, 10-10 thru 10-12
  INSERT, 2-12, 2-13
  ISTEP, 4-4
  LIST, 10-6
  LOAD, 4-1, 4-2
  LSTEP, 4-3, 4-4
  MODIFY, 5-12, 5-13, 7-5
  NOLIST, 10-6
  OFFSET$OF, 10-15
  PATCH, 9-1 thru 9-3
  PROC, 8-1
  PSTEP, 4-3, 4-4
  PUT, 7-6 thru 7-8
  QUIT, 2-13
  Read and Write I/O ports, 5-9, 5-10
  REGS, 5-8
  REMOVE, 7-5, 7-6
  REMOVE PATCH, 9-3
  REPEAT construct, 6-1, 6-2
  SASM, 5-13 thru 5-19
  SELECTOR$OF, 10-15
  STRLEN, 10-13 thru 10-15
  SUBSTR, 10-13 thru 10-15
  TRCREG, 11-3, 11-4
  VIEW, 2-13 thru 2-15
  WRITE, 10-12, 10-13
  XCHANGE, 2-13
PSCOPE constructs
  COUNT, 6-1, 6-2
  DO, 6-3
  IF, 6-2, 6-3
  REPEAT, 6-1, 6-2
PSCOPE controls
  CRT, 2-1, 2-2
  Invocation, 2-1
  MACRO, 2-2, 2-3
  NOCRT, 2-1, 2-2
  NOMACRO, 2-2, 2-3
  NOSUBMIT, 2-3
  SUBMIT, 2-3
  VSTBUFFER, 2-4
PSCOPE debug session example, 1-6 thru 1-18
PSCOPE delimiters, G-2
PSCOPE directory, 10-2 thru 10-4
PSCOPE grammar, F-1 thru F-7
PSCOPE menu commands
  BLOCK (B), 2-11, 2-12
  DELETE (D), 2-12
  GET (G), 2-12
  INSERT (I), 2-12, 2-13
  QUIT (Q), 2-13
  VIEW (V), 2-13 thru 2-15
  XCHANGE (X), 2-13
PSCOPE operators, G-2
PSCOPE reserved keywords, G-1
PSCOPE.CRT file, 2-8
PSCOPE.MAC file, 2-2
PSTEP command, 4-3, 4-4
PUT command, 7-6 thru 7-8

QUIT (Q) command, 2-13

Related publications, vi, vii
Read and write I/O ports, 5-9, 5-10
Referencing names, 3-2, 3-3
Referencing program symbols,
  see program symbol reference

Registers
  Break, 11-1 thru 11-3
  Trace, 11-3, 11-4
Remove code patch, 9-3
REMOVE command, 7-5, 7-6
REMOVE PATCH command, 9-3
REPEAT control construct, 6-1, 6-2
Reserved keywords, G-1
Returning from a debug procedure, 8-2, 8-3
RUN command, C-4

Sample Pascal program,
  see sample PSCOPE debug session
Sample PL/M-86 program,
  see sample PSCOPE debug session
Sample PSCOPE debug session, 1-7 thru 1-18
Screen-oriented editor,
  see internal screen-oriented editor
SELECTOR$OF command, 10-15
Series III information, C-1
Series IV information, D-1
Severe errors, A-1
SLA, see single-line assembler
Single-line assembler, 5-13 thru 5-17
Single-line disassembler, 5-13 thru 5-17
Single-stepping through a program, 4-3, 4-4
Stepping commands,
  see the ISTEP, LSTEP, and PSTEP commands
String constants, 3-9
String functions
  CI, 10-13 thru 10-15
  CONCAT, 10-13 thru 10-15
  STRLEN, 10-13 thru 10-15
  SUBSTR, 10-13 thru 10-15
STRLEN command, 10-13 thru 10-15
SUBMIT file, 2-3
SUBMIT invocation control, 2-3
SUBSTR command, 10-13 thru 10-15
Symbol object types, 3-6
Symbol references, see program symbol references
Symbol table, 3-6
Syntax errors, 2-7
Syntax notation conventions, vii, viii

Televideo 910 Plus configuration, B-15
Televideo 925 and 950 configuration, B-16
Terminate debugging session, 2-6
Tested configurations, B-4 thru B-17
Tokens, 3-1 thru 3-6
Trace registers, 11-3, 11-4
Tracepoints, 11-3 thru 11-6
Trapping exceptions, 11-6, 11-7
TRCREG command, 11-3, 11-4
Type conversions, 3-12, 3-13
  for expressions, 3-13
  for assignments, 3-13

Unprintable characters, 2-9
User symbol object types, 3-6
Utility commands, 10-1 thru 10-13
  BASE, 10-9, 10-10
  CALLSTACK, 10-4, 10-5
  DIR, 10-2 thru 10-4
  EVAL, 10-7 thru 10-9
  EXIT, 10-1, 10-2
  HELP, 10-5, 10-6, A-1
  INCLUDE, 10-6, 10-7
  INPUTMODE, 10-10 thru 10-12
  LIST, 10-6
  NOLIST, 10-6
  WRITE, 10-12, 10-13

VIEW command, 2-13 thru 2-15
Virtual symbol table, see VSTBUFFER invocation control
VSTBUFFER invocation control, 2-4

Warnings, A-1
WRITE command, 10-12, 10-13

XCHANGE (X) command, 2-13

Zentec configuration, B-17

ıe Single Line Assembler/Disassembler

```
ASM address [LENGTH expr
             TO address ]

{ASM     address = 'assembler-mnemonic' [,assembler-mnemonic]
SASM}
```

ƷCOPE Memory Types

ADDRESS
BCD
BOOLEAN
BYTE
CHAR
DWORD
EXTINT
INTEGER
LONGINT
LONGREAL
POINTER
REAL
SELECTOR
SHORTINT
TEMPREAL
WORD

ƷCOPE Operations

AND	Logical AND
MOD	Modulo
NOT	Logical NOT
OR	Logical OR
XOR	Logical exclusive OR
*	Multiplication
−	Negation or subtraction
+	Identity or addition
/	Division
= =	Boolean equality
< >	Inequality
>	Greater than
<	Less than
> =	Greater than or equal
< =	Less than or equal
.	The address of
( )	Bracketing
[ ]	Array indexing
=	Assignment
:	Pointer constructor

PSCOPE-86
Pull-out Reference Card

How to Use This Card

italics	Italics indicate a generic term. Replace it with a user-defined symbol, a PSCOPE keyword, or other system-defined term.
[ ]	Brackets indicate an option. You may or may not enter one of the choices set off by brackets. Do not type the brackets.
{ }	Braces indicate a required choice. You must choose one of the choices set off by the braces. Do not type the braces.
*	An asterisk after a set of brackets means that you can make multiple choices from within the brackets or repeat your choice. Do not type the asterisk.

Any other symbols (parentheses or commas) are to be considered as part of the command.

Invoking and Exiting PSCOPE

```
{[pathname]PSCOPE.86 [CRT [MACRO [SUBMIT [VSTB(decimal-number)]

RUN[pathname]PSCOPE} NOCRT] NOMACRO] NOSUBMIT]

    EXIT
```

File Handling Commands

```
APPEND pathname [memory-type       [,memory-type
                 debug-type         ,debug-type
                 PATCH address      ,PATCH address
                 debug-object-name] ,debug-object-name]*

INCLUDE pathname [NOLIST]

LIST [pathname]

LOAD pathname [E8087           [CONTROLS command-tail]
               CH8087
               NOLINES
               NOSYMBOLS]

NOLIST

PUT pathname [memory-type        [,memory-type
              debug-type          ,debug-type
              PATCH address       ,PATCH address
              debug-object-name] ,debug-object-name]*
```

Defining Debug Registers

```
DEFINE BRKREG name= break-pt [CALL proc-name] [,break-pt [CALL proc-name]]*

DEFINE TRCREG name= trace-pt [CALL proc-name] [,trace-pt [CALL proc-name]]*
```

The called debug procedure must return a boolean value, as follows.

```
RETURN TRUE
RETURN FALSE
```

Defining Debug Procedures, Debug Variables and Patches

```
DEFINE PROC name = [command]
```

Accessing Debug Procedure Parameters

```
%expr-for-passed-parameter
%NP

DEFINE memory-tyep name = expr

DEFINE PATCH address [TIL address] = [command
                                      NOP     ]
```

Removing Debug Objects

```
REMOVE [memory-type       [,memory-type
        debug-type         ,debug-type
        PATCH address      ,PATCH address
        debug-object-name] ,debug-object-name]*
```

Entering Emulation

```
GO [TIL break-pt [,break-pt]*
     USING {break-register-name  [,break-register-name
            trace-register-name   ,trace-register-name
            BRKREG                ,BRKREG
            TRCREG                ,TRCREG               ]*}
     FOREVER                                            }

ISTEP

PSTEP

LSTEP
```

Displaying Debug Objects, Program Variables, and Registers

```
BRKREG break-register-name

TRCREG trace-register-name

name

memory-type {address    [LENGTH expr
             .name }     TO address]

PATCH address

REGS
```

8086/8088 Registers:

AX	CX	CS	DI
AH	CH	DS	SI
AL	CL	ES	
BX	DX	SS	
BH	DH	SP	
BL	DL	IP	

8087 Registers:

ST0	ST4	FSW	FIO
ST1	ST5	FCW	
ST3	ST6	FIA	
ST4	ST7	FDA	

Block Commands

```
COUNT expr
[WHILE expr
 UNTIL expr
 [command]*]
END[COUNT]

REPEAT
[WHILE expr
 UNTIL expr
 [command]*
END[REPEAT]

DO
[command]*
END

IF expr THEN [command]
[ORIF expr THEN [command]]*
[ELSE [command]]
END[IF]
```

String Commands

```
CONCAT(string-spec[,string-spec])

CI

STRLEN(string-spec)

SUBSTR(string-spec,start,length)
```

Utility Commands

```
$[=expr]

ACTIVE(symolic-reference)

BASE[=expr]

CALLSTACK [expr]

DIR [DEBUG          [memory-type
  PUBLIC             debug-type
  module-name]       user-type     ]

EDIT [debug-object-name
      GO
      PATCH address
```

CNTL-A	Deletes line to right of cursor.
CNTL-C	Cancels current editing command.
CNTL-F	Deletes character at the cursor position.
CNTL-X	Deletes line to left of cursor.
CNTL-Z	Deletes current line.
HOME	Moves cursor to left/right of line after left/right arrow.
RUBOUT	Deletes character to left of cursor.

```
EVAL expr [LINE
           PROCEDURE
           SYMBOL [ALL]]

HELP [topic-name
      en               ]
INPUTMODE[=expr]
```

Utility Commands

```
NAMESCOPE[=expr]

OFFSET$OF(expr)

PORT(port-number)[=expr]

WPORT(port-number)[=expr]

SELECTOR$OF(expr)

WRITE [USING (format-item)] [expr[,expr]*]

format-item =
```

n	Decimal number specifying the width of the output field.
*n*C	Move output buffer pointer to column *n* (first column).
*n*X	Skip *n* spaces.
H	Set display base in hexadecimal.
T	Set display base to decimal.
Y	Set display base to binary.
.	Terminate the format string.
>	Terminate the format string; do not issue a <cr> or <lf> until the next WRITE; only has meaning when within a block command or procedure that has more than one WRITE.
&	Terminate the format string; do not execute the WRITE until the next WRITE.
"	Delimit text to be written.

```
VIEW pathname
```

PSCOPE-86 User's Guide
165496-001

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

____
____
____
____
____
____

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

____
____
____
____
____

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

____
____
____
____
____
____

4. Did you have any difficulty understanding descriptions or wording? Where?

____
____
____
____

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). ____________

NAME ______________________________ DATE ____________

TITLE ____

COMPANY NAME/DEPARTMENT ______________________________

ADDRESS ____

CITY ____________________ STATE ____________ ZIP CODE ____________
(COUNTRY)

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE NECESSARY IF MAILED IN U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 N.E. Elam Young Parkway.
Hillsboro, Oregon 97123

DSHO Technical Publications

intel®

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.

INSTRUMENTATION

DS-132/5K/0784/OSPS