PSCOPE-86 HIGH-LEVEL PROGRAM DEBUGGER USER'S GUIDE

Order Number: 121790-002

PSCOPE-86 HIGH-LEVEL PROGRAM DEBUGGER USER'S GUIDE

Order Number: 121790-002

NOTE

Before running PSCOPE on a Series IV development system, rename the file *PSCOPE.86* to *PSCOPE*. This allows the command **RUN PSCOPE** to invoke PSCOPE on both the Series III and Series IV. Because the Series IV ignores **RUN**, entering **PSCOPE** is sufficient for invocation.

AEDIT	iLBX	iOSP	MULTIBUS
BITBUS	im	iPDS	MULTICHANNEL
BXP	iMMX	iRMX	MULTIMODULE
COMMputer	Insite	iSBC	Plug-A-Bubble
CREDIT	Int$_e$l	iSBX	PROMPT
i	Int$_e$lBOS	iSDM	Promware
iATC	Intelevision	iSXM	Ripplemode
I^{2}ICE	int$_e$ligent Identifier	Library Manager	RMX/80
ICE	int$_e$ligent Programming	MCS	RUPI
iCS	Intellec	Megachassis	System 2000
iDBP	Intellink	MICROMAINFRAME	UPI
iDIS			

REV.	REVISION HISTORY	DATE
-001	Original issue.	5/82
-002	Added Series IV information. Made miscellaneous corrections.	3/83

PREFACE

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

- 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 7 appendices, consisting of:

Chapter 1	"Introduction" 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	"Using the Debugger" 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	"Command Language and Expressions" 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	"Loading and Executing Programs" details how to load programs and control execution using the GO command.
Chapter 5	"Examining and Modifying Program Symbols" 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	"Control Constructs" describes the four control constructs used by PSCOPE command language: REPEAT, COUNT, IF, and DO.
Chapter 7	"Debug Object Manipulation Commands" describes the command syntax necessary for defining, displaying, modifying, saving, and removing debug objects.
Chapter 8	"Debug Procedures" describes what debug procedures are and how to define, use, display, save, and remove them.
Chapter 9	"Code Patches" explains how to use code patches in a user's program.
Chapter 10	"Utility Commands and String Functions" describes and explains the syntax for each of the utility commands and string functions available with PSCOPE.
Chapter 11	"Advanced Execution and Trace Commands" describes how to use breakpoints and tracepoints to control and monitor program execution.
Appendix A	"Error Messages" is a numerically ordered list of the PSCOPE error messages.
Appendix B	"Configuring PSCOPE for Non-Intel Terminals" provides the codes necessary to configure PSCOPE to run on non-Intel terminals.
Appendix C	"Additional Information for Series III Users" provides further information about using PSCOPE on the Series III Microprocessor Development System.
Appendix D	"Additional Information for Series IV Users" provides further information about using PSCOPE on the Series IV Microprocessor Development System.
Appendix E	"Sample Programs" contains the program upon which the examples in this manual are based.
Appendix F	"PSCOPE Grammar" lists the lexical conventions PSCOPE recognizes.
Appendix G	"Reserved Keywords" lists the PSCOPE reserved keywords.

Related Publications

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

Pascal-86 User's Guide, 121539

PL/M-86 User's Guide for 8086-Based Development System, 121636

FORTRAN-86 User's Guide, 121570

The following publications contain further information on the Series III:

Intellec® Series III Microprocessor Development System Product Overview, 121575

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

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

The following publications contains further information on the Series IV:

Intellec® Series IV Microcomputer Development System Overview, 121752

Intellec® Series IV Operating and Programming Guide, 121753

Intellec® Series IV ISIS-IV User's Guide, 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.
lowercase words	Grammar symbols for which you must substitute a value. These symbols are underlined as well as printed in lowercase.
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 ({ }) exactly as shown. For example, you must enter all the punctuation shown in the following command: `SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')`
filename	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 that locates the file by identifying the device and/or 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

CONTENTS (continued)

FIGURES

TABLES

CHAPTER 1 INTRODUCTION

This chapter introduces the high-level program debugger PSCOPE-86 and explains how to use PSCOPE-86 on a software development system.

Product Definition

PSCOPE-86 is an interactive, symbolic debugger for object modules produced by the PASCAL-86 compiler (version 1.0 or later), the PL/M-86 compiler (version 2.0 or later), and the FORTRAN-86 compiler (version 1.1 or later).

PSCOPE-86 (PSCOPE) lets you examine and modify a program's execution to find software logic errors. PSCOPE features:

- Breakpoints, single stepping, and execution trace of procedures, statements, and labels.
- High-level code patches, written in the PSCOPE command language.
- An internal screen-oriented editor for creating code patches and debug procedures, as well as for editing commands.
- Unlimited (virtual) storage of program symbols.
- Access to program symbols using arbitrary expressions and Pascal-like variable references.

Both PSCOPE and the program being debugged reside in the development system memory (which is expandable to one megabyte).

The Program Development Process

PSCOPE is part of the development software for your microcomputer system. Figure 1-1 shows how PSCOPE fits into the program development process.

Program development includes the following operations:

1. Write the source code using your host system editor.
2. Produce relocatable object code using the 8086/8088 compilers and assembler.
3. Correct any compile-time errors using the editor, then recompile.
4. Link the program to run-time libraries and create a loadable version of the program using LINK86 (with the BIND option).
5. Execute, test, and debug the loadable object module using PSCOPE.

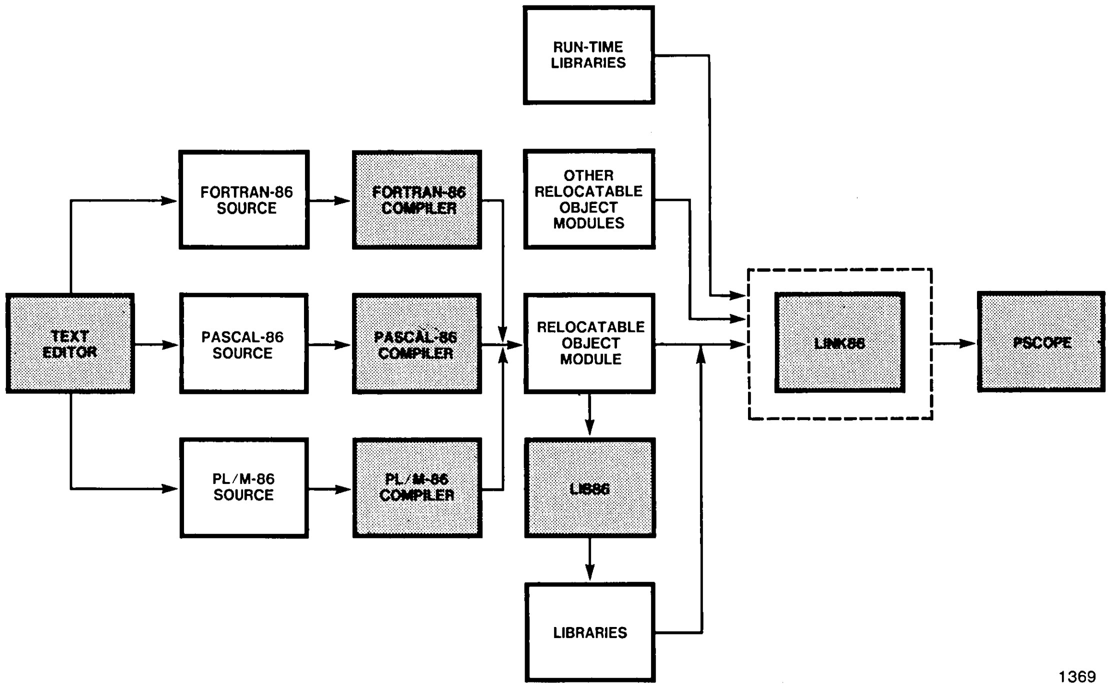


Figure 1-1 Generalized Program Development Process

Host System Execution Environment

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

PSCOPE Restrictions

Compiler controls used when the program object code is produced can affect the reliability of some debug functions. To avoid possible confusion resulting from compiler optimizations, use the OPTIMIZE(O) control when compiling code to debug.

PSCOPE does not support debug load modules that use overlays.

Major Functions

This section 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 provides two methods of stepping through a program.

- Symbolic Debugging (Chapter 5)

  PSCOPE lets you symbolically examine and modify data for all PASCAL-86/88 and PL/M-86/88 data types. A virtual symbol tables stores symbol and line number information.

- 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)

  PSCOPE lets you define and manipulate various types of debug objects, including LITERALLYs (a form of command macro) and debug variables.

- Debug Procedures (Chapter 8)

  You can define and edit debugger procedures and symbols.

- Code Patching (Chapter 9)

  PSCOPE lets you add and delete code at the statement level without permanently changing your program. Patches are valid for only the debugging session in which they are generated.

- Utility Commands (Chapter 10)

  The utility commands:

  - Provide an online HELP command.
  - Let you save part or all of a debug session in a log file.
  - Let you execute a command file.

- Breakpoints and Tracepoints (Chapter 11)

  Breakpoints and tracepoints let you control and monitor debugging.

Introductory Sample Session

Figure 1-2 shows a Pascal-86 program containing several bugs. Figure 1-3 shows how to debug this Pascal-86 program using PSCOPE. Appendix E contains a more sophisticated Pascal-86 program (DC.86). These two programs are used throughout this manual as examples of various PSCOPE features.

```
SERIES-III Pascal-86, V2.0


Source File: :F3:SAMPLE.PAS
Object File: :F3:SAMPLE.OBJ
Controls Specified:  DEBUG.

STMT LINE NESTING        SOURCE TEXT: :F3:SAMPLE.PAS
   1    1  0  0          (*********************************************************
                          The following program reads in a list of positive
                          integer values, prints the number of values read, the
                          mean, and the high and low values.  The end of the list
                          is indicated by a negative value.

                          *** Note that this version of the program contains
                              several bugs, and is designed as a simple demo
                              for PSCOPE.

                          *********************************************************)

                         program sample (input, output);
   2   14  0  0          var
                           value : integer;
   3   16  0  0            sum   : integer;
   4   17  0  0            high  : integer;
   5   18  0  0            low   : integer;
   6   19  0  0            count : integer;
   7   20  0  0          begin
   7   21  0  1            (***** bug: sum & count not initialized *)
                           writeln ('Input a list of non-negative integers, one per line.');
   8   23  0  1            writeln ('(A negative value indicates the end of the list.)');
   9   24  0  1            readln (value);
  10   25  0  1            (****** bug: high & low not initialized *)
                           while value >= 0 do begin
  11   27  0  2              count := count + 1;
  12   28  0  2              sum   := sum + value;
  13   29  0  2              if value > high then high := value;
  15   30  0  2              if value < low then low := value;
  17   31  0  2              readln (value);
  18   32  0  2              end;
  20   33  0  1            writeln ('Number of values = ', count);
  21   34  0  1            writeln ('Mean             = ', sum div count);
  22   35  0  1            writeln ('Low              = ', low);
  23   36  0  1            writeln ('High             = ', high);
  24   37  0  1          end.


Summary Information:

PROCEDURE                 OFFSET    CODE SIZE      DATA SIZE     STACK SIZE
SAMPLE                     00C2H   019AH    410D  001AH     26D  000EH     14D
-CONST IN CODE-                    00C2H    194D

Total                              025CH    604D  001AH     26D  0042H     66D
```

1370

Figure 1-2 Sample Pascal Program

```
SERIES-III PSCOPE-86, V1.0
*load :f7:sample.86
dir
DIR of :SAMPLE
PQ_OUTPUT . . . . . . . . . . TEXT (file)
PQ_INPUT . . . . . . . . . . . TEXT (file)
VALUE  . . . . . . . . . . . . integer
SUM  . . . . . . . . . . . . . integer
HIGH . . . . . . . . . . . . . integer
LOW  . . . . . . . . . . . . . integer
COUNT  . . . . . . . . . . . . integer
*
* /* We know there is a bug in this program,                     */
* /*  it prints wrong values for all the variables               */
* /* First, we will determine whether the values are being input */
* /*  correctly by setting a breakpoint after the 1st cycle      */
*
*go til #17
Input a list of non-negative integers, one per line.
(A negative value indicates the end of the list.)
5
[Break at #17]
* /* input was 5 */
* /* look at some variables */
*
*value
+5
*sum
+309
* /* let's make it easier - define a proc to print out all the variables */
*
*define proc print = write 'value =', value, ' sum =', sum, &
** ' high =', high, ' low =', low, ' count =', "count
*
* /* since count is a PSCOPE keyword, it is necessary to prefix it with "    */
*
*print
value = +5  sum = +309  high = +7632  low = -4091  count = +22
*
* /* value is right - everything else is wrong.  Looks like nothing is       */
* /*  getting initialized. Try setting to their correct values and continue  */
*
*sum = 5; high = 5; low = 5; "count = 1
*
*print
value = +5  sum = +5  high = +5  low = +5  count = +1
*go
9
[Break at #17]
* /* input was 9 */
*print
value = +9  sum = +14  high = +9  low = +5  count = +2
* /* ta da! */
*go forever
3
10
6
0
-1
Number of values = 6
Mean             = 5
Low              = 0
High             = 10

EXCEPTION: Program call to DQ$Exit
[Break at location 1061H:0030H]
* /* inputs were 3,10,6,0 and -1  -- a ok */
*exit
PSCOPE terminated                                                       1371
```

Figure 1-3 Sample PSCOPE Debug Session

CHAPTER 2
USING THE DEBUGGER

This chapter describes how to use the debugger, including:

- 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 I NOCRT
MACRO I NOMACRO
NOSUBMIT I SUBMIT
WORKSPACE

Each control is described individually in the following sections.

CRT I NOCRT

Syntax

CRT [(filename)]
NOCRT

Abbreviation

CR I NOCR

Default

PSCOPE.CRT

Definition

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 "Configuring PSCOPE for Non-Intel Terminals".

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 developement 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.

Example

```
RUN PSCOPE

run pscope crt(1510t.crt)

RUN PSCOPE NOCRT
```

MACRO | NOMACRO

Syntax

MACRO [(<u>filename</u>)]
NOMACRO

Abbreviation

MR | NOMR

Default

PSCOPE.MAC

Definition

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.

Example

```
run pscope macro(procs.mac)

RUN PSCOPE NOMR
```

NOSUBMIT | SUBMIT

Syntax

NOSUBMIT
SUBMIT

Abbreviation

NOSM | SM

Default

NOSUBMIT

Definition

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.

Example

```
run pscope sm

RUN PSCOPE NOSM
```

WORKSPACE

Syntax

WORKSPACE (decimal-number)

Abbreviation

WS

Definition

WORKSPACE assigns the amount of system memory to be used by PSCOPE for storing debug objects. The WORKSPACE default is 15000 bytes. The minimum value for decimal-number is 10000 bytes; the maximum value is 65535 bytes. The amount of memory specified with WORKSPACE is taken from user memory at initialization, thereby directly affecting the amount of memory available for the program being debugged.

NOTE

Decreasing the the value of WORKSPACE below 15000 bytes leaves more memory for user program but decreases room for debugger objects. (Debugger objects are discussed in Chapter 7).

Example

```
run pscope workspace(12000)
```

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, Vx.y
```

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

In Vx.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-C)
- 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, "Utility Commands", 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 EXIT command to exit the debugger:

```
EXIT
```

The debugger responds with:

```
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 command 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 left of the cursor.
CNTL-F or DEL CHAR (RUB OUT)	Deletes the character at the cursor.

CNTL-X	Deletes all characters left of the cursor.
CNTL-A	Deletes the character at the cursor and all characters right of the cursor.
CNTL-Z or CLEAR LINE	Deletes the entire current line.
Left Arrow key	Moves the cursor one character left.
Right Arrow key	Moves the cursor one character 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 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 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

The debugger contains a built-in editor for creating and modifying debugger objects. It helps you create long command lines and edit the last command line.

The internal editor is screen-oriented and has the options shown in figure 2-1. If your terminal is not a standard Intel terminal, the internal editor uses information in the PSCOPE.CRT file to define I/O sequences for cursor control. Additionally, the PSCOPE.CRT file lets you 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.

Entering the Internal Editor

You can invoke the internal editor in 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.

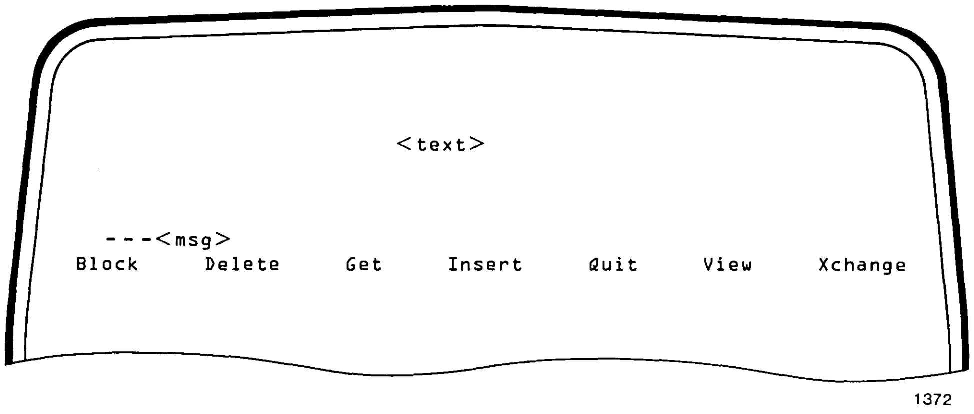
If you do not specify anything, the editor invokes with an empty edit space.

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.



```
                    <text>



   ---<msg>
Block     Delete     Get     Insert     Quit     View     Xchange
```

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 carriage return (CR) and linefeed (LF) characters printable characters. The CR/LF combination acts as a single character called Return. A tab is interpreted as a single space.

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 follows:

Up Arrow Key Pressing the up arrow key moves the cursor up one line in 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 in 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 was in the last line of the text display area, the text is scrolled up one line. If the cursor was at the end of the text, pressing the RETURN key has no affect.

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 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, above) 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:

- 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 (described earlier). 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 com mand 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 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.

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 pair of characters that separates or marks 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-comment 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, "Reserved Keywords," 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, and/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	A - 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:

```
:DC.@9999
```

The debugger accepts names of unlimited length; however, it uses only the first 40 characters. The debugger interprets uppercase and lowercase characters to be 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 symbols 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 format for a line number reference is:

[:<u>module-name</u>] #<u>line-number</u>

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 decimal value:

0.24×10^{-2}

may be expressed as

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:

'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:

'WHAT''S UP?'

is stored as

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	Multiplication
−	Minus sign	Negation or substraction
+	Plus sign	Identity or addtion
/	Slash	Division
==	Double equal signs	Equality
<>	Angle brackets	Inequality
>	Angle bracket	Greater than
<	Angle bracket	Less than
>=	Bracket, equals	Greater than or equal
<=	Bracket, equals	Less than or equal
.	Dot	Address of (prefix operator)
( )	Parenthesis	Bracketing
[ ]	Square brackets	Array indexing
=	Equal sign	Assignment
↑	Up arrow	Indirection via pointer
:	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 */. For example:

```
/* 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

Expressions

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

Expressions can be:

- A single number, constant, or symbolic reference. Examples are:

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:

```
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 earlier 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 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:

.symbolic-reference

Machine Register References

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

The registers are:

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 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. For 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:

```
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:

```
#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:

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 operator, pointer selector operators (pointer selection uses built-in functions), arithmetic operators, memory-type operators, relational operators, and logical operators.

Table 3-6 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 noted otherwise. All operations are binary unless specified as unary operations.

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.

Table 3-6 Precedence of Operators (Highest to Lowest)

Group	Operator	Operation	Precedence
Arithemetic	+	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 for Expressions

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

- The type of value expected by an operator or function.
- Not loosing 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 object files into the debugger. You can ignore debug information when loading or specify that the 8087 emulator is linked into the object file.

Syntax

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

where:

file is the object file you want to debug

load-option is one of the following:

E8087 links the 8087 emulator to the program object file. You must select this option if you want to use REALs. (The *8087 Support Library Reference Manual* contains information on the 8087 emulator and numeric support.)

NOLINES suppresses your program's line numbers.

NOSYMBOLS suppresses your program's symbol information.

command-tail is any arbitrary text expected by the loaded program for invocation controls.

Description

LOAD loads the specified file into the development 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.

CONTROLS 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

```
*LOAD dc.86
```

The GO Command

The GO command transfers execution control to the loaded program.

Syntax (simplified)

GO [TIL <u>expression</u> [, <u>expression</u>]*]

or

GO FOREVER

where:

<u>expression</u> 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 either 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

```
*pstep
```

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 object commands
  Display Program Symbols
  Change Program Symbols
  Change Name Scope
  Active function

- Memory manipulation commands
  Display Memory
  Modify Memory

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 the 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 below.

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:

```
: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:

```
: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:

```
: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 fully qualified reference example from above (: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:

```
get_token.digit.c
```

The fully qualified referenced 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 their scope. This is useful for operations like setting breakpoints, patches, etc. However, referencing symbols outside 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 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.

Table 5-1 Default Display Formats (continued)

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 above. 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 above. 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

```
*buffer.str[buffer.index] = 'x'

*buffer.index = buffer.index + 1

*term.factor_1_value = 0

*:dc.variable_table['a'] = -23
```

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:

```
NAMESCOPE = $
```

returns the current name scope to the current execution point.

Resuming program execution (using GO, $STEP, 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 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

```
*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 below:

```
*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.

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 bounded 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

```
*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 debugger symbols with the same names. You can define global debugger 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 commands 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 that 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 implicitly removes some global debug objects. Chapter 4 describes the LOAD command.) 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) 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:

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 local 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. value can be the result of evaluating an expression. PSCOPE assigns a null value if you do not give a value. value is required for debug type debug objects. 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 value shown below:

```
*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

value's type 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/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 delete 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 memory type object (or a list of objects) to be deleted by name and/or type. Specifying DEBUG instead of a list of types and/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

```
*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 that 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 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 save 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

```
*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 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. 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/or 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 below.

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 three ways:

- In response to a PSCOPE prompt.
- As an operand of an expression.
- On 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 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 %
...* 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 addrl [TIL addr2] = patch-value

where:

addrl 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 replace the first patch if you specify a second patch.

If you specify only addrl, then program execution resumes at addrl. If you specify both addrl 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 break and trace commands.)

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

Be careful not to overlap patches. For example, because

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

and

```
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
```

This 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 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

SCOPE furnishes a variety of utility commands. This chapter discusses these commands, which include:

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

```
*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
FACTOR_VALUE ................. 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

```
*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
```

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.

En is the error number for which you want the extended error message. Note that the form En 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 LIST files 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

```
*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

```
*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 three optional evaluation types:

LINE	indicates line number
PROCEDURE	indicates procedure name
SYMBOL	indicates a fully-qualified reference

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 message displayed is:

: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 that most nearly matches 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 message displayed is:

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

```
*eval $ procedure
*eval $
*eval ds:14h symbol
```

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 default base in the current base if you enter BASE without expr.

You can modify BASE by setting it to an expr whose value is 2, 10, or 16 (decimal values). Note that BASE acts like a variable; it is assigned a value or displayed just like a BYTE debug variable.

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 two steps:

- Scanning for syntax errors and figuring what to do.
- Executing the command.

PSCOPE evaluate numeric constants during the first phase of command processing. PSCOPE assigns values to variables during the second phase. Thus, commands such as

```
*base = 10t
*base = 16; VAR1 = 10
```

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

```
*base = 16t
*base = 0A
```

The WRITE Command

The WRITE command lets you display and format information at the terminal.

Syntax

WRITE [USING (string-spec)] [expr [, expr]*]

where:

USING lets you control output using a format string.

string-spec is an expression that evaluates to a string.

expr 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. PSCOPE recognizes the following format items:

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.
nC	Move output buffer pointer to column n (first column is 1.)
nX	Skip n spaces in the output buffer.
H	Set WRITE command display base to hexadecimal.
T	Set WRITE command display base to decimal.
Y	Set WRITE command display base to binary.
.	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.

PSCOPE reuses the format string if the argument contains more items than specified in the format string.

Example

```
*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

```
*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

```
*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 make 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 below.

Syntax

DEFINE BRKREG name = break-item [, break-item]*

where:

name is the name of the break register.

break-item is:

breaks [CALL proc-name]

breaks is:

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 were 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 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 registers 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 prolog 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 below.

SYNTAX

DEFINE TRCREG name = trace-item [, trace-item]*

where:

trace-item is:

traces [CALL proc-name]

traces is:

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 were called. If the Boolean value is FALSE, PSCOPE continues execution without displaying a message, as if there were 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 Ap pendix 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/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 register, 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 messages 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 using the HELP command:

HELP E_n_

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 unrecognizable format.

1 Array's lower bound is unknown - zero is assumed.

2 Symbol is not an array or has fewer dimensions than specified.

3 Size of 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 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 current execution point is outside of program or 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 field from 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 is not 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.
Segment part of line reference pointer not known. Maybe symbol information was not loaded for 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.

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 CALLSTACK 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.

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 fix up by 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 was purged.

114 Missing E8087 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 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).

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 +infinity or -infinity or NaN.

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.

169 Error while attempting to read virtual symbol table.

196 - 511
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.

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. Your 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.

516 Execution halted.
PSCOPE halted execution due to the unexpected setting of the single-step trap flag. This can occur if you enter CNTL-C while executing an interrupt routine entered while PSTEPing or LSTEPing. A subsequent GO command should allow normal execution to continue.

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 Series IV text editor, AEDIT, in its AEDIT.MAC file. (See AEDIT Text Editor User's Guide, 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 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. CONTROL A.
AFXF=hhhh	6H	Sets CHAR DELETE. CONTROL F.
AFXX=hhhh	18H	Sets DELETE LEFT. CONTROL X.
AFXZ=hhhh	1AH	Sets CLEAR LINE. CONTROL Z.
AFCD=hhhh	1CH	Sets DOWN.
AFCH=hhhh	1DH	Sets HOME.

Table B-1 Configuration Commands (continued)

Command	Series III Default	Meaning
AFCL=hhhh	1FH	Sets LEFT.
AFCR=hhhh	14H	Sets RIGHT.
AFCU=hhhh	1EH	Sets UP.
AFIG=h		This character will be ignored if input. This character is needed on terminals, such as the Hazeltine 1510, which have multiple character key codes for UP and DOWN. 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 typeahead buffer.
AFMB=hhhh	0DH	Moves cursor to start of line.
AFMD=hhhh	1CH	Moves cursor down.
AFMH=hhhh	1DH	Moves cursor home.
AFML=hhhh	1FH	Moves cursor left.
AFMR=hhhh	14H	Moves cursor right.
AFMU=hhhh	1EH	Moves cursor up.
AFES=hhhh	1B45H	Erases entire screen.
AFER=hhhh	1B4AH	Erases rest of screen.
AFEK=hhhh	1B4BH	Erases entire line.
AFEL=hhhh		Erases rest of line.
AFAC=hhhh		Addresses cursor lead in. When used, code will be followed by column number (0 to 79) and row number (0 to 24).
AO=h	0H	Offset to add both row and column number with address cursor command.
AX=T or F	T	True if X (column) precedes Y (row) in address cursor command.
AW=T or F	T	Allows user to indicate that terminal wraps when character is printed in column 80.
AFIL=hhhh		Inserts line code. Used in line 0 for reverse scrolling.
AFDL=hhhh		Deletes line code. Used to speed up 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:

- ADDS Regent 200 (2400 baud only)
- Beehive Mini-Bee
- DEC VT52
- DEC VT100
- Hazeltine 1510
- Lear Seigler ADM-3A

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.

Adds Regent Model 200

This ADDS model has a 24-line CRT display with 80 character 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=1B 45;  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.

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-3 shows the Beehive Mini-Bee configuration.

Table B-3 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-4 shows the DEC VT52 configuration.

Table B-4 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 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-5 shows the DEC VT100 configuration.

Table B-5 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 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-6 shows the Hazeltine 1510 configuration.

Table B-6 Hazeltine 1510 Configuration

Function Code	Hexadecimal Value (~ Lead-In)	Graphic or ASCII Name
1510T 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	~ DCI
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=7E0C;
AFMD=7E0B;  AFMR=10;  AFML=8; AFMH=7E12
AFMB=0D;  AFES=;  AFER=7E18;  AFEK=;
AFEL=7E0F  AFAC=7E11;  AFIL=7E1A;  AFDL=7E13;
```

Table B-6 Hazeltine 1510 Configuration (continued)

Function Code	Hexadecimal Value (ESC Lead-In)	Graphic or ASCII Name
1510E		
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;

NOTE

You must enter a tilde instead of ESCAPE.

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-7 shows the Lear Siegler ADM-3A configuration.

Table B-7 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
```

APPENDIX C ADDITIONAL INFORMATION FOR SERIES III USERS

This appendix contains specific information on the Intellec Series III Microcomputer 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,* 121575
- *Intellec® Series III Microcomputer Development System Console Operating Instructions,* 121609
- *Intellec® Series III Microcomputer Development System Programmer's Reference Manual,* 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 number under each operation refers to the manual(s) that describe that operation.

1. *ISIS-II CREDIT™ CRT-Based Text Editor User's Guide,* 9800902
2. *AEDIT™ TEXT EDITOR User's Guide,* 121956
3. *Pascal-86 User's Guide,* 121539
4. *PL/M-86 User's Guide for 8086-Based Development System,* 121636
5. *FORTRAN-86 User's Guide,* 121570
6. *iAPX 86,88 Family Utilities User's Guide,* 121616

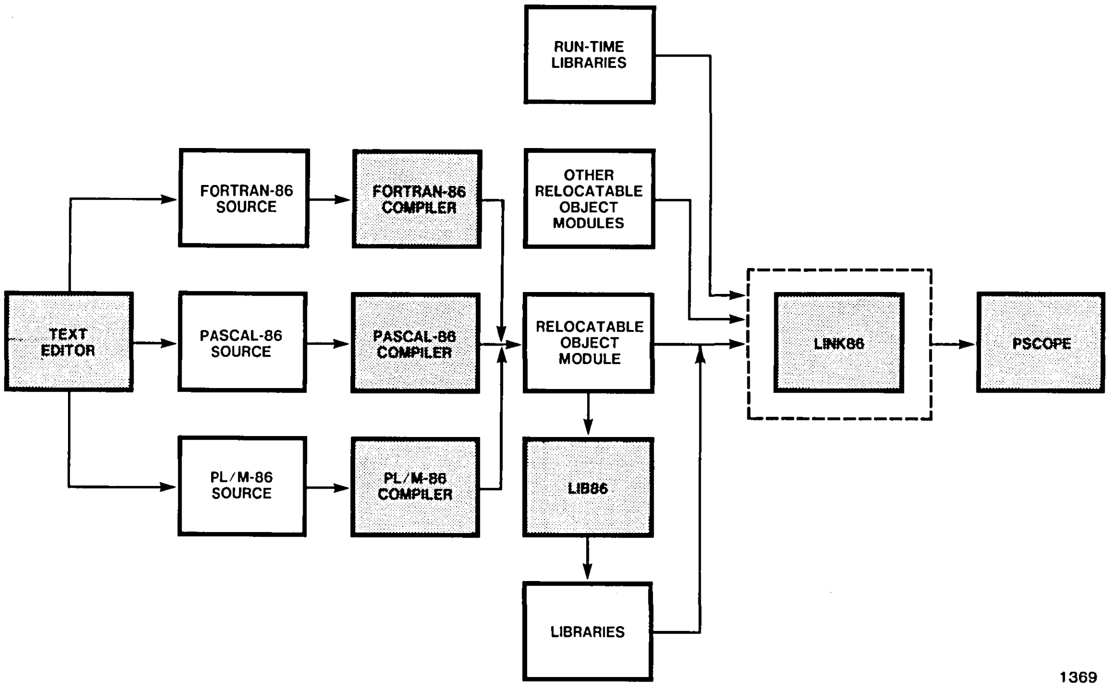


Figure C-1 Series III Program Development Process

Hardware and Software Required

You need the following hardware and software to run the debugger:

- Intellec Series III development system (run release 1.2 or later)
- ISIS-II 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 1.1 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 96K bytes. You must add more memory to accommodate additional workspace and your program.

Your program must be a load-time locatable (LTL) or 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 1.1 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 96K 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<u>n</u>:] PSCOPE [<u>controls</u>]

or

> [:F<u>n</u>:] PSCOPE [<u>controls</u>]

where:

:Fn: is the disk in drive <u>n</u>. <u>n</u> can be 0 through 9.

<u>controls</u> 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 Microcomputer 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 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,* 121752
- *Intellec® Series IV Operating and Programming Guide,* 121753
- *Intellec® Series IV ISIS-IV User's Guide,* 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 number under each operation refers to the manual(s) that describe that operation.

1. *ISIS-IV CREDIT™ CRT-Based Text Editor User's Guide,* 9800902
2. *AEDIT™ TEXT EDITOR User's Guide,* 121956
3. *Pascal-86 User's Guide,* 121539
4. *PL/M-86 User's Guide for 8086-Based Development System,* 121636
5. *FORTRAN-86 User's Guide,* 121570.
6. *iAPX 86,88 Family Utilities User's Guide,* 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 1.1 or later).
- 8086-based utilities
- PSCOPE high-level program debugger

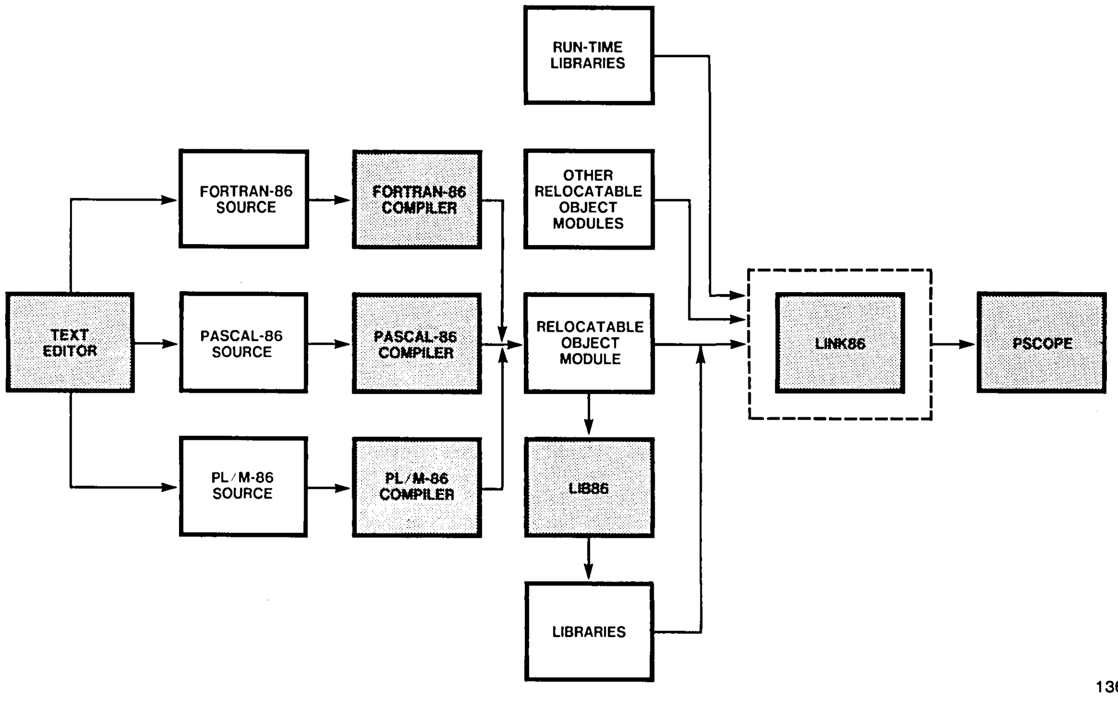


Figure D-1 Series IV Program Development Process

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 96K bytes. You must add more memory to accommodate additional workspace and the user program.

Your program must be a load-time locatable (LTL) or 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 1.1 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 96K 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:

```
/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.

```
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
```

```
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 *)
```

```
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;
```

```
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
```

```
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;




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.
```

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 :: = callstack-command
                   | count-command
                   | define-command
                   | directory-command
                   | display-command
                   | do-command
                   | edit-command
                   | eval-command
                   | exit-command
                   | go-command
                   | help-command
                   | if-command
                   | include-command
                   | list-command
                   | load-command
                   | modify-command
                   | put-command
                   | remove-command
                   | repeat-command
                   | return-command
                   | step-command
                   | write-command

  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

  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
```

```
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]

list-command :: = LIST [pathname]
                | NLIST

load-command ::= LOAD pathname [load-option]*
                   [CONTROLS controls-text]


load-option :: = NOLINES
               | NOSYMBOLS
               | 1E8087

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


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
                 | BX
                 | CX
                 | DX
                 | BP
                 | SP
                 | DI
                 | SI
                 | CS
                 | DS
                 | ES
                 | SS
                 | IP
                 | FLAG
                 | 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	AH
AL	AND	APPEND	ARRAY
AX	BASE	BCD	BH
BL	BOOLEAN	BP	BRKREG
BX	BYTE	CALL	CALLSTACK
CH	CHAR	CI	CL
CONCAT	CONTROLS	COUNT	CS
CX	DEBUG	DEFINE	DH
DI	DIR	DL	DO
DS	DWORD	DX	E8087
EDIT	ELSE	END	ENDCOUNT
ENDIF	ENDREPEAT	ENUMERATION	ES
EVAL	EXIT	EXTINT	FALSE
FH	FILE	FL	FLAG
FOREVER	GLOBAL	GO	HELP
IF	INCLUDE	INTEGER	IP
LABEL	LENGTH	LINE	LIST
LITERALLY	LOAD	LONGINT	LONGREAL
LSTEP	MOD	MODE	MODULE
NAMESCOPE	NOCODE	NOLINES	NOLIST
NOSYMBOLS	NOP	NOT	NP
OFFSET$OF	OFFSETOF	OR	ORIF
PATCH	POINTER	PROC	PROCEDURE
PSTEP	PUBLIC	PUT	REAL
RECORD	REMOVE	REPEAT	RETURN
SELECTOR	SELECTOR$OF	SELECTOROF	SET
SHORTINT	SI	SP	SS
STRLEN	SUBSTR	SYMBOL	TEMPREAL
THEN	TIL	TO	TRACEACT
TRACEREGS	TRCREG	TRUE	UNTIL
USING	WHILE	WORD	WRITE
XOR			

PSCOPE Operators and Delimiters

; , : $ " ↑ = . [ ] ( ) % <= < >

>= <> == + − * / /* */

APPENDIX H PSCOPE COMMAND INDEX

This appendix lists each PSCOPE command and refers you to the section in text where you can find more information.

INDEX

intel®

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 CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.

DEVELOPMENT SYSTEMS

DS-139/5.2K/0884/AP