

DB86 SOFTWARE DEBUGGER USER'S GUIDE

Order Number: 481850-001

REV.	REVISION HISTORY	DATE
-001	Original Issue.	12/88

In the United States, additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are printed on the last page of this document.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

Intel Corporation retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products. (Registered trademarks are followed by a superscripted ®.)

Above	iLBX	Intellink	MICROMAINFRAME	Ripplemode
BITBUS	im [®]	iOSP	MULTIBUS [®]	RMX/80
COMMputer	iMDDX	iPAT	MULTICHANNEL	RUPI
CREDIT	iMMX	iPDS	MULTIMODULE	Seamless
Data Pipeline	Inboard	iPSC	ONCE	SLD
ETOX	Insite	iRMK	OpenNET	SugarCube
FASTPATH	Intel [®]	iRMX [®]	OTP	UPI
Genius	intel [®]	iSBC [®]	PC BUBBLE	VLSiCE
Δ	Intel376	iSBX	Plug-A-Bubble	376
i [®]	Intel386	iSDM	PROMPT	386
i ² ICE	intelBOS	iSXM	Promware	386SX
ICE	Intel Certified	KEPROM	QueX	387
iCEL	Intelevison	Library Manager	QUEST	387SX
iCS	Intelligent Identifier	MAPNET	Quick-Erase	4-SITE
iDBP	Intelligent Programming	MCS [®]	Quick-Pulse Programming	
iDIS	Intellec [®]	Megachassis		

IBM, PC AT, Personal System/2, and PS/2 are registered trademarks and PC XT is a trademark of International Business Machines Corporation.

Copyright © 1988, Intel Corporation, All Rights Reserved

Preface.....	vii
--------------	-----

Chapter 1 Introduction

1.1	Overview of Debugger Capabilities.....	1-1
1.1.1	Debugging Features.....	1-2
1.1.2	Human Interface Features.....	1-3
1.2	The Application Development Process.....	1-4
1.3	Compatibility Issues.....	1-7
1.4	Trademarks.....	1-7
1.5	A Quick Tour of DB86.....	1-8
1.5.1	Invoking the Debugger.....	1-8
1.5.2	The Debugger Screen.....	1-9
1.5.3	Entering Commands and Getting Help.....	1-12
1.5.4	Scrolling through Source Code.....	1-16
1.5.5	Setting a Breakpoint.....	1-18
1.5.6	Executing the Program.....	1-20
1.5.7	Showing User Output.....	1-22
1.5.8	Exiting from the Debugger.....	1-24

Chapter 2 Debugging with DB86

2.1	Preparing a Program for Debugging.....	2-2
2.2	Invoking the Debugger.....	2-6
2.3	Loading a Program.....	2-8
2.4	Setting Breakpoints.....	2-10
2.4.1	Tracepoints.....	2-10
2.4.2	Breakpoints.....	2-12
2.4.3	Breaking on Access to Data.....	2-13
2.4.4	Other Breakpoint Commands.....	2-14
2.5	Executing a Program.....	2-15
2.6	Observing the Output of a Program.....	2-19
2.7	Examining and Modifying the Program.....	2-19
2.8	Examining and Modifying Data.....	2-22
2.8.1	Simple Ways to Display Data.....	2-22
2.8.2	Advanced Techniques for Displaying Data.....	2-23
2.8.3	Modifying Data.....	2-26
2.9	Configuring the Debug Environment.....	2-27
2.10	Finding a Bug.....	2-28

Chapter 3 DB86 Menu System

3.1	Introduction	3-1
3.1.1	Starting the Menu System	3-2
3.1.2	Selecting Commands from the Menus	3-2
3.1.3	Exiting from the Menu System.....	3-3
3.2	The Debug Menu (Alt-D).....	3-4
3.3	The Window Menu (Alt-W).....	3-7
3.4	The Go Menu (Alt-G).....	3-11
3.5	The Set Menu (Alt-S).....	3-15
3.6	The View Menu (Alt-V).....	3-17
3.7	The Browse Menu (Alt-B).....	3-19
3.8	The Help Menu (Alt-H).....	3-23

Chapter 4 DB86 Keyboard Controls

4.1	Introduction	4-1
4.2	Navigational Key Controls.....	4-1
4.3	Line-editing Key Controls	4-3
4.4	Function Key Controls.....	4-5
4.5	Other Keyboard Controls.....	4-8

Chapter 5 DB86 Command Language Encyclopedia

5.1	How to Use this Chapter	5-1
5.2	Functional Overview of Commands.....	5-3
	Alphabetical Entries	5-10

Appendix A DB86 Installation

A.1	Installing the DB86 Debugger Software.....	A-1
A.2	Modifying the System Configuration.....	A-3
A.3	Setting Environment Variables.....	A-3
A.4	Keyboard Templates	A-4

Appendix B DB86 Invocation

B.1	Syntax.....	B-1
B.2	Controls.....	B-1
B.3	Examples.....	B-4

Appendix C Shortcuts and Tips

C.1	Setting up a Second Monitor	C-1
C.2	Using RAM Disk for the Virtual Symbol Table Buffer	C-2
C.3	Shortcut for Setting SPATH	C-3
C.4	Hints on Naming Modules	C-3
C.5	Using the Debug Control in ASM86 Programs	C-4
C.6	Debugging Programs with Overlays	C-4
C.7	Using the SWAP Control	C-4
C.8	Tips on Variable Names and Reserved Words	C-5
C.9	Tips on Clearing Breakpoints	C-5

Appendix D Language Support

D.1	iC-86	D-1
D.2	PL/M-86	D-1
D.3	ASM86	D-2
D.4	Fortran-86	D-2
D.5	Pascal-86	D-2

Appendix E Error Messages

Appendix F Reserved Words

Appendix G ASCII Codes

Glossary

Index

Service Information Inside Back Cover

Figures

1-1	Initial DB86 Screen	1-1
1-2	The Application Development Process	1-5
1-3	The Debugger Screen	1-10
2-1	DB86 Screen with EXAMPLE Program Loaded.....	2-1
2-2	DOS Batch File for EXAMPLE Program.....	2-3
2-3	Link Response File for EXAMPLE Program	2-5
3-1	Debugger Screen with Help Menu	3-1
5-1	DB86 Command Language Help Screen	5-1

Tables

5-1	DB86 Control Constructs	5-4
5-2	DB86 Debug Environment Commands	5-4
5-3	DB86 Processor Status Commands.....	5-5
5-4	DB86 Source Display Commands	5-6
5-5	DB86 Memory Access Commands.....	5-6
5-6	DB86 Execution and Watch Commands.....	5-7
5-7	DB86 Topical Entries	5-8
5-8	DB86 Breakpoint Commands	5-17
5-9	DB86 Command-line Editing	5-26
5-10	Display Output Control Keys	5-27
5-11	DB86 Control Constructs	5-28
5-12	DB86 User-program Types.....	5-33
5-13	DB86 mtypes and their Language-specific Names	5-41
5-14	Operators.....	5-54
5-15	Types and their Valid Operators.....	5-55
5-16	Disparate Types and their Valid Binary Operators.....	5-56
5-17	Type Conversions in Expressions.....	5-58
5-18	Type Conversions in Assignments.....	5-60
5-19	Special Character Delimiters.....	5-78
5-20	8086/8088 Registers	5-96
5-21	8087 Registers.....	5-100
G-1	ASCII Code List	G-1
G-2	ASCII Control Code Definition.....	G-3

This manual describes the DB86 debugger and how to operate it on a DOS-host computer. DB86 is an interactive software debugger for finding logical errors in programs. It works with programs in Intel OMF86 format bound with LINK86.

This manual is written for programmers who use Intel compilers or assemblers, such as iC-86, PL/M-86, ASM86, Fortran-86, or Pascal-86, to write their applications software. To use this product, you should also be familiar with PC-DOS Version 3.0 or later.

Chapters 1 and 2 of this manual introduce the debugger and provide examples that illustrate tactics for using the debugger. Chapters 3 through 5 provide reference material on the menu system, keyboard controls, and debugger commands. The appendixes provide additional reference material, e.g., installation instructions and invocation instructions.

Manual Organization

The manual is organized as follows.

- Chapter 1 introduces the debugger and describes how it fits into the application development process. It also includes a short sample session that gets you started with the debugger.
- Chapter 2 provides numerous examples of using the debugger, showing how related commands work together to do specific debugging tasks.
- Chapter 3 is a reference guide to the debugger menu system.
- Chapter 4 is a reference guide to the debugger keyboard controls.
- Chapter 5 is a reference guide to the debugger command language. The entries in the command encyclopedia are alphabetized for quick reference and include both command descriptions and debugger topics.

- Appendix A contains the installation instructions.
- Appendix B describes the invocation controls and the DOS command line for invoking the debugger.
- Appendix C provides shortcuts and tips for setting up and using the debugger more efficiently.
- Appendix D describes the level of symbolic support for each supported language.
- Appendix E lists error messages with brief descriptions of the causes and suggested actions for recovery.
- Appendix F lists reserved words.
- Appendix G is an ASCII code table.

Related Publications

DB86 is part of Intel's 8086 and 80186 family of software development tools. The family consists of compilers, assemblers, software utilities, and debuggers for developing programs that can be run on 8086- and 80186-based systems, operating in real mode.

Refer to the following publications for related information on compilers and assemblers, utilities, and architecture:

AEDIT Manual Set for DOS Systems, order number 122716.

ASM86 Assembly Language Reference Manual, order number 122386

ASM86 Macro Assembler Operating Instructions, order number 122391.

iAPX 86, 88 Family Utilities User's Guide, order number 122396

Operating System Interface Libraries Manual, order number 122401.

8087 Support Support Library Reference Manual, order number 122406.

An Introduction to ASM86, order number 121689.

iC-86 Compiler User's Guide for DOS Systems (in two parts), order number 480561 for the general compiler reference section and 480564 for the DOS-specific operation section; and *iC-86 Binder/Slipcase*, order number 480588.

CLIB-86 Supplement, order number 481690.

C: A Reference Manual [Harbison-Steele/Tarton Labs, Prentice Hall], order number 555107.

Fortran-N86 Compiler User's Guide for DOS Systems (in two parts), order number 480986 for the general compiler reference section and 480987 for the DOS-specific operation section; and *Fortran-86/186/286 Binder/Slipcase*, order number 481160.

Fortran 77 [Katzen, Van Nostrand Reinhold], order number 481199.

PL/M Programmer's Guide, order number 452161; *DOS Supplement to the PL/M Programmer's Guide*, order number 452162; and *PL/M Programmer's Guide Binder/Slipcase*, order number 452160.

Pascal-86 User's Guide, order number 122426.

iAPX 86/88, 186/188 User's Manual Programmer's Reference, order number 210911.

iAPX 86/88, 186/188 User's Manual Hardware Reference, order number 210912.

8086/8088 16-Bit Microprocessor Primer, [Morgan and Waite], order number 555094.

The 8086 Microprocessor Architecture, Software and Interfacing Techniques, [Singh and Triebel], order number 555953.

The 16-Bit Microprocessor, [Goody], order number 555104.

The 8087 Primer, [Palmer and Morse], order number 555694.

C

C

C

Contents

Chapter 1 Introduction

1.1	Overview of Debugger Capabilities	1-1
1.1.1	Debugging Features	1-2
1.1.2	Human Interface Features.....	1-3
1.2	The Application Development Process	1-4
1.3	Compatibility Issues	1-7
1.4	Trademarks	1-7
1.5	A Quick Tour of DB86.....	1-8
1.5.1	Invoking the Debugger.....	1-8
1.5.2	The Debugger Screen	1-9
1.5.3	Entering Commands and Getting Help	1-12
1.5.4	Scrolling through Source Code	1-16
1.5.5	Setting a Breakpoint	1-18
1.5.6	Executing the Program	1-20
1.5.7	Showing User Output	1-22
1.5.8	Exiting from the Debugger.....	1-24

(C)

(C)

(C)

This chapter introduces the DB86 source-level debugger. It contains an overview of debugger capabilities, an overview of the application development process, and a sample debugging session. Compatibility issues are also discussed.

1.1 Overview of Debugger Capabilities

DB86 is a source-level, symbolic debugger with a windowed human-interface that includes pulldown menus as well as a command-line interpreter for controlling debugging.

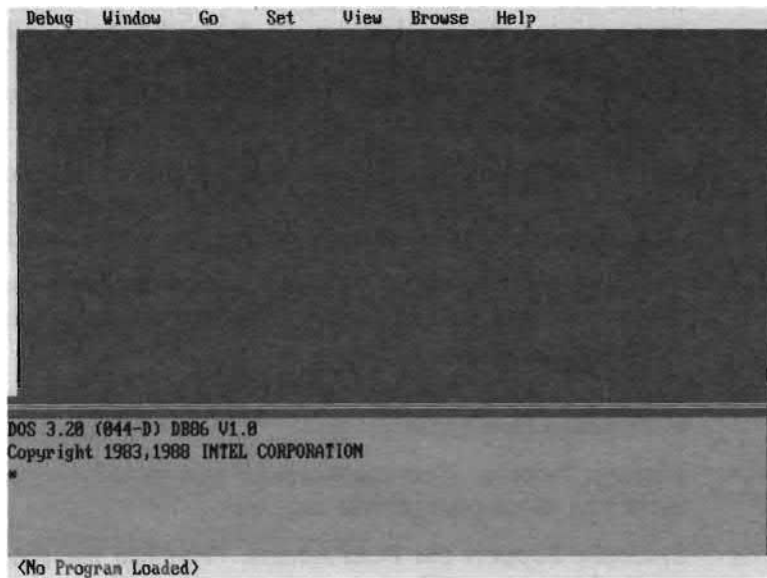


Figure 1-1 Initial DB86 Screen

With DB86, you can find logical errors in your application software by scrolling and browsing through your source code; by controlling the execution of the program; by examining data, memory, and processor registers during execution; and by observing the output of your program.

1.1.1 Debugging Features

DB86 provides the following debugging features:

- **Source-level Display.** Source code for high-level languages is displayed in the View Window. The debugger can also display the disassembled code for each high-level language statement. You can scroll through the source and browse from module to module in your program or browse to any executable point in the source.
- **Breakpoints.** You can define locations in your program at which execution is to be halted under control of the debugger. These points can be defined as temporary, fixed, or conditional breakpoints. Breakpoints can be defined symbolically using module names, procedure names, and line numbers. You can also set tracepoints at program locations to display an announcement when execution reaches that location.
- **Single-step.** You can execute your program a step at a time, stepping through an assembly-language instruction, a high-level language statement or through a function or procedure.
- **Assembly/Disassembly.** You can display memory using assembler mnemonics and you can patch programs by assembling instructions into memory.
- **Memory Access.** You can display and modify memory formatted as common data types.
- **Register Access.** You can examine and modify registers, including math coprocessor registers.
- **Symbolic Support for Data Access.** You can use program symbols to display and modify memory.
- **Watch Expressions.** You can define watch expressions and observe them in the Watch Window as the program executes.

- **Screen Flipping.** You can flip to the display screen of the application program to observe the output of the program during execution.
- **Coprocessor Support.** You can execute numeric processor instructions on an 8087 emulator or on an 8087 math processor.
- **Transparent Overlay Support.** Symbolic overlay support is built into the debugger. You can set breakpoints in overlays and not be concerned, during subsequent debugging, about when the overlay is resident.

1.1.2 Human Interface Features

The debugger provides a windowed human interface with several ways to issue commands.

- The menu system lets you select items from a menu to issue commands while you are still learning how to use the debugger.
- Many debugging functions in DB86 can be performed with single key controls. In other words, pressing a single key starts the function immediately. This responsive level of control combines with the source-level display of the application program to provide a point-and-shoot model for debugging.

First, you point at the source code shown in the View Window by scrolling to a location in the program. Then, you fire off debugging commands using the function keys.

- At the command line, you can control debugging with a powerful set of debugging commands. These include control constructs, automated execution of commands from a file, expression evaluation, and a DOS shell command. When entering commands, you can edit the command line or re-issue commands from the history buffer.

On-line help is available throughout the debugger and includes extended help on error messages.

The debugger can also keep track of the commands issued during a session by logging them to a list file.

1.2 The Application Development Process

DB86 is part of a family of products aimed at easing the development of applications for the 8086 family of microprocessors. Figure 1-2 shows how DB86 fits in the development process.

In the program-development cycle for 8086 and 80186 applications, object files are created from separately translated modules following these steps.

- Using an editor that generates pure ASCII files, such as Intel's AEDIT, create the source text. See step 1 in Figure 1-2.
- Invoke a compiler or assembler to translate the source text into object code. See step 2 in Figure 1-2.
- Compile and test parts of your application before other parts are written by separating your source text into several files (i.e., modules) and specifying only one or some of the source files in each compilation.
- Include source text from several files or from a set of alternate files by specifying the appropriate compiler controls and preprocessor directives in a single compilation.
- Call functions written in different Intel languages, including iC-86, PL/M-86, ASM86, Fortran-86, or Pascal-86.

To organize some of your object modules into libraries, use LIB86. Later, when you link a program to a library, the linker extracts only those library modules that the program needs. Libraries are good places to store commonly used procedures. See step 3 in Figure 1-2.

Use LINK86 to link your modules together. LINK86 can produce a relocatable module or a module for incremental linking. LINK86 is part of the ASM86 package. See step 4 in Figure 1-2.

After producing linked modules, use DB86 to test and debug your program. Correct errors in the source text, translate the source text again, and link it again until you are satisfied that the program is reliable. See step 5 in Figure 1-2.

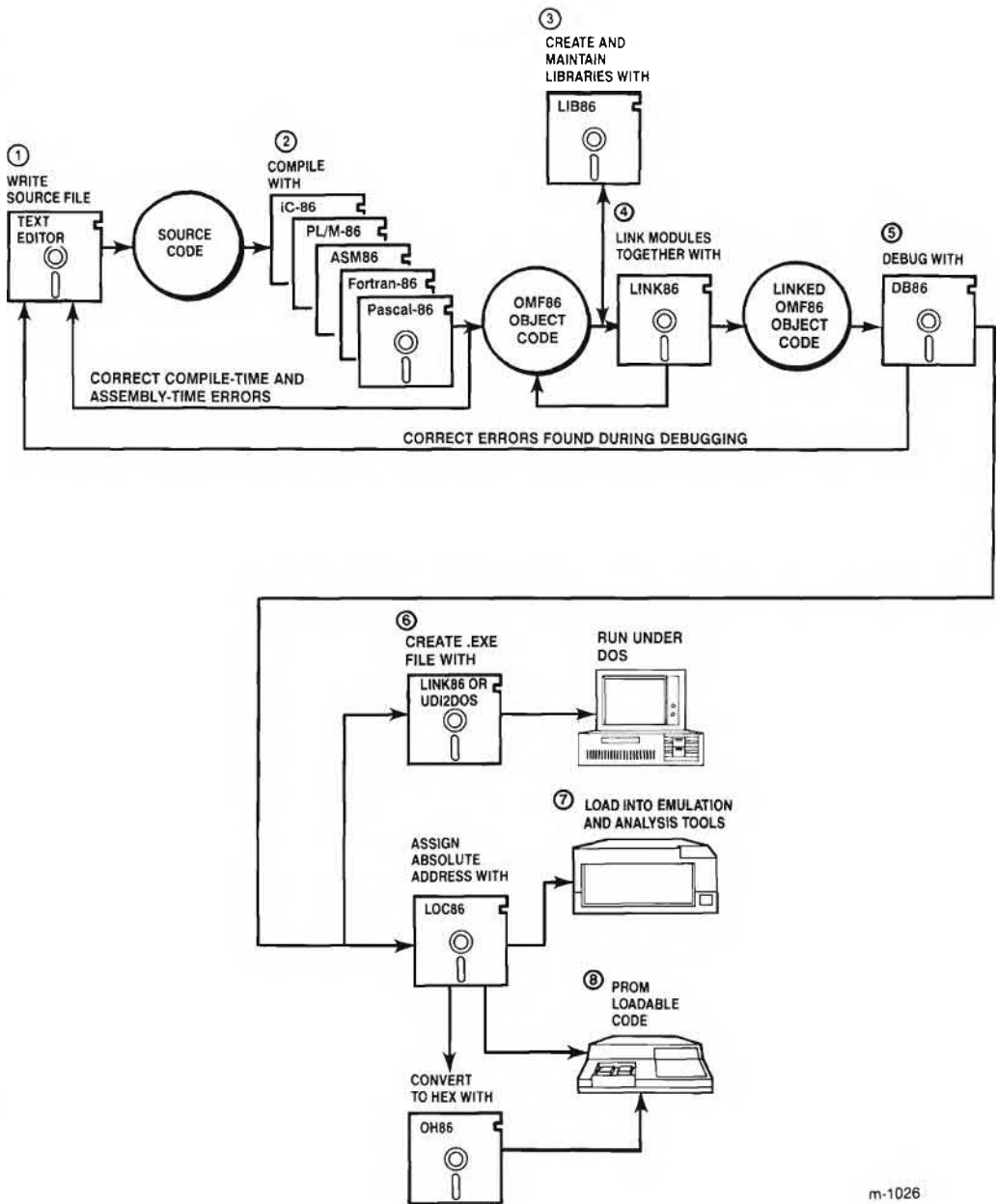


Figure 1-2 The Application Development Process

After software debugging and testing, you load the object code into system memory for execution and final debugging using one of the following processes:

- Configure the object code for DOS. You can use the /EXE option of LINK86, linking in the C libraries, to generate a .EXE file suitable for execution on DOS. Or you can use the operating-system-interface utility called UDI2DOS. This utility adds the DOS runtime support to linked object modules, transforming them into .EXE files that are suitable for execution on DOS. UDI2DOS is part of the ASM86 package. See step 6 in Figure 1-2. (The debugger requires .86 files and cannot use the .EXE files produced in this step.)
- Load the object code into a target system with an in-circuit emulator, such as the I²ICE™ in-circuit emulator or the ICE™ -186 in-circuit emulator. You can also use Intel's iPAT™ Performance Analysis Tool for symbolic performance and code coverage analysis. Assign absolute addresses with LOC86 before loading your code into an in-circuit emulator. See step 7 in Figure 1-2.
- Program it into ROM using LOC86 to assign absolute addresses to the object module generated by LINK86. The object code can be formatted by the compiler or assembler to be programmed into a PROM device using Intel's iPPS PROM programming software and the iUP-200 PROM Programmer. Or you can invoke OH86 to convert the absolute code into hexadecimal format as required by most PROM programming equipment. See step 8 in Figure 1-2.

1.3 Compatibility Issues

The DB86 debugger runs under DOS Version 3.0 or later. See Appendix A for detailed host-system requirements. Also, see Appendix C for tips on improving debugger performance on the host system.

The debugger works with OMF86 modules produced with Intel's LINK86, using one of the following Intel compilers or assemblers:

- iC-86 (Version 4.0)
- PL/M-86 (Version 3.1)
- ASM86 (Version 3.1)
- Fortran-86 (Version 3.0)
- Pascal-86 (Version 3.1)

See Appendix D for a detailed description of the level of support provided for each of these languages.

1.4 Trademarks

ICE, I²ICE, Intel, and iPAT are trademarks of Intel Corporation.

IBM and IBM Personal System/2 are registered trademarks of International Business Machines Corporation.

1.5 A Quick Tour of DB86

This section contains an example session to help you get started with the debugger. The sample session describes the debugger screen display and covers the following preliminary information:

- invoking the debugger
- getting help
- scrolling through source code
- setting a breakpoint
- executing the program
- showing user output
- exiting from the debugger

By going through this example, you will learn three ways to issue commands to the debugger.

Before running this example, you must install the debugger software following the instructions in Appendix A.

1.5.1 Invoking the Debugger

To invoke the debugger, change to the directory containing the debugger and example programs. Assuming that the debugger directory is named DB86, type:

```
C>cd \db86
```

Once you are in the debugger directory, use the batch file, TRYIT.BAT, to invoke the debugger.

```
C:\DB86>tryit
```

The TRYIT batch file contains a command that not only invokes the debugger, but also loads the EXAMPLE program and runs it under debugger control. The individual steps for invoking the debugger are described in more detail in Chapter 2. For now, use the batch file provided to get started quickly. The resulting screen looks like the following:

```

Debug Window Go Set View Browse Help
238
239 return(key); /* return with item number selecte
240 }
241
242 void far main()
243 {
244 /*=====MAIN=====
245 This is the main entry point for the entire EXAMPLE
246 program. If you are viewing this with the DB86 debugger
247 try using the Grey- and Grey+ keys to scroll the source
248 file in the view window. Grey* will restore the view
249 back to the current execution point (horizontal bar).
250 The F1 key brings help. Alt-<firstletter> for menu.
251 =====
252 int ni;
253 int done = FALSE;

Copyright 1983,1988 INTEL CORPORATION
*spath = .\,c,.lst
*load example.86
*go til main
[ Break at :MENU#243 ]
*
Mod: MENU Proc: MAIN Line: #243 BRK

```

1.5.2 The Debugger Screen

This section takes a short digression from the tour with the EXAMPLE program to describe the major elements of the debugger screen. The debugger screen is divided into at most seven framed areas called windows. See Figure 1-3. Some of these windows are optional; others are always present. Each window is used for a different purpose by the debugger. Only one window is active at any given time; commands can be issued in the active window. The active window is highlighted on color monitors; on monochrome monitors, the blinking cursor is shown in the active window. At sign-on, the active window is the command window.

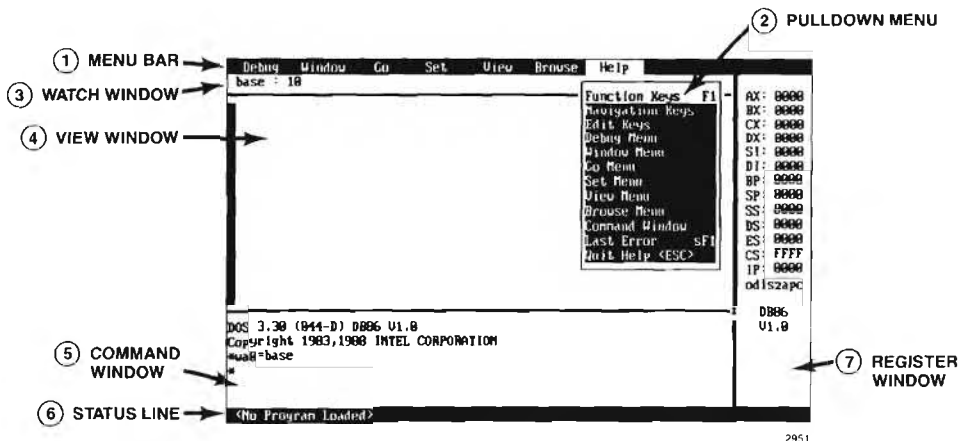


Figure 1-3 The Debugger Screen

The numbered elements of Figure 1-3 are:

1. The top line of the screen is called the menu bar. The menu bar lists seven menu names, each of which can be used to select a pull-down menu.

The menu bar can be turned off if you do not need it. See Appendix B for further information on the no-menu-bar (NOMB) control.

2. The Help Menu has been selected. Each pull-down menu has a list of commands on it. You can select commands from the list. Only one menu can be pulled down at a time. See Chapter 3 for further information on the menu system.

3. A Watch Window is open on the second line of the screen. The debugger automatically creates a Watch Window large enough to display all currently defined watch expressions. You can toggle the presence of the Watch Window by typing sF2, that is, hold down the Shift key while pressing the F2 key. Watch expressions allow you to observe data in your program. They can contain symbolic references from your program or debug variables (as shown in Figure 1-3). The current value of the watch expression is shown and is updated as your program executes.
4. The View Window contains the source code of the currently loaded program. The View Window is always present. The Break Status column is the leftmost column of the View Window. This column indicates where breakpoints are set and also contains the thumbmark, a pointer to the current line. When the View Window is active, the blinking cursor is attached to the thumbmark indicating the current line.
5. The Command Window is the area where you enter debugger commands at a command-line prompt. The Command Window is always present.

You can make the Command Window larger and the View Window smaller by typing the Ctrl-PgUp key; you can make the Command Window smaller and the View Window larger by typing the Ctrl-PgDn key.
6. The status line contains the current location in the program that is currently loaded and the reason for the last break in execution of that program. Debugger messages are also sometimes displayed on the status line. The status line is always present.
7. The Register Window contains the current value of the processor registers and flags. It can be toggled on and off by pressing the F2 key.

1.5.3 Entering Commands and Getting Help

There are three ways to enter commands in the debugger:

- Using the menu system, you can select the command that you want to execute. See Chapter 3 for more information on the menu system.
- You can immediately initiate common tasks with single key controls like function keys, navigational keys, or line-editing keys. See Chapter 4 for more information on single key controls.
- You can enter debugging commands at the prompt in the Command Window. See Chapter 5 for more information on debugging commands.

On-line help is available for each of these three groups of commands.

Continuing with the example, you have just invoked the debugger using the TRYIT batch file. Now, you can follow these steps to get help.

Press these keys in sequence:	To do this:
Alt-H	Pulls down the Help Menu. This menu allows you to get help on any other menu in the menu system. Additionally, you can get help on single key controls, debugger commands, and error messages.
D	Displays a help screen for the commands you can select from the Debug Menu.

Here is what the help screen looks like:

```
Debug Window Go Set View Browse Help
===== DEBUG MENU HELP =====
Load Program  Prompts for the name of the object file to be loaded
               and loads that program for debugging. You may
               specify drive and pathname before the file name.
               Normal line editing functions are available when you
               are typing the program name.
Reload        Clears all defined breakpoints, tracepoints, and
               watch expressions and then reloads the most recently
               loaded object file.
Source Path   Prompts for the pathname of the subdirectory in
               which DB86 should search for source display files.
               You may also specify up to ten file extensions,
               which DB86 uses when looking for source display
               files. For example: \mysourcedir,.c,.lst
DOS Shell     Temporarily exits DB86 and goes to a loaded DOS
               shell at which you may issue normal DOS commands.
               To return from DOS to DB86, type the EXIT command.
Exit DB86     Exits DB86 and returns to DOS.

               === PRESS ANY KEY TO CONTINUE ===

Co
* S
* 1
* g
* C
*
Mod: MENU Proc: MAIN Line: #243 BRK
```

**Press these keys
in sequence:**

To do this:

spacebar

Pressing any key returns to the Help Menu.

↓

You can also select items on a menu with the uparrow (↑) and downarrow (↓) keys. Downarrow (↓) moves the highlight bar down one item to select the Window Menu item.

Enter

Displays the help screen for the Window Menu.

spacebar

Pressing any key returns to the Help Menu.

You can continue displaying help screens for the menu system by pressing G, S, V, and B to get help on the Go, Set, View, and Browse Menus respectively. Press any key after each help screen is displayed to return to the Help Menu.

You can also get help on single key controls from the Help Menu by selecting Function Keys, Navigation Keys, or Edit Keys. If you try any of these, be sure to press a key after each help screen to return to the Help Menu.

The Help Menu also contains an overview of debugger commands. Select the Command Window item in the Help Menu to display this help screen.

You should be at the Help Menu now.

Press these keys in sequence:	To do this:
Esc	Exits from the Help Menu. The debugger returns to the Command Window with the cursor blinking at the prompt awaiting a command. The Esc key can be used to exit from any menu.
F1	The F1 function key displays a screen of help on all the function keys. The function key help screen is the same as the function key help screen on the Help Menu.
spacebar	Pressing any key returns to the Command Window at the command-line prompt.
Ctrl-PgUp (14 times)	Continue typing Ctrl-PgUp to expand the Command Window to its maximum size.

Enter the HELP command (type help at the prompt and press the Enter key):

```
*help
```

The debugger displays a list of topics on which you can get help:

```
Debug Window Go Set View Browse Help
238
239 return(key); /* return with item number selecte

*help
*
HELP is available for:
ASM          BASE          BREAKPOINT  BROWSEMENU  CALLS        CMDWINDOW
COMMANDS     COMMENTS  CONSTRUCTS  COUNT       CSTEP       DEBUGMENU
DEFINE       DEREFERENCE  DIR         DISPLAY     DO           EDIT
ENTRY        EXCEPTIONS  EXIT        GO          GOMENU      HELP
IF           INCLUDE     INVOCATION  ISTEP      KEYBOARD    LABELS
LINES        LIST        LOAD        LSTEP      MENU        MODIFY
MODULES      OBJECTS     OPERATORS   PORT        PROCEDURES  PSTEP
REALS        REGISTERS   REPEAT     RSTEP      SCOPE       SETMENU
SOURCE       STACK       TYPES       VARIABLES   VIEWMENU    WINDOWMENU
*
Mod: MENU Proc: MAIN Line: #243 BRK
```

To get help on any of these subjects enter the HELP command again followed by the subject. For example, to get help on the HELP command enter the following command line at the prompt:

```
*help help
```

The debugger displays help on the HELP command in the Command Window and awaits another command at the prompt. If the help information does not fit in the Command Window, the debugger pauses after displaying the text that does fit. You can press F to continue the output with no further pausing; L to display one line of output at a time; or P to continue displaying a page of text at a time.

1.5.4 Scrolling through Source Code

The next part of the example illustrates scrolling through source code.

Press these keys in sequence: **To do this:**

Ctrl-PgDn (14 times) Expands the View Window to its normal size.

At this point, the screen looks like the following:

```
Debug Window Go Set View Browse Help
238
239     return(key);                /* return with item number selecte
240 }
241
242 void far main()
243 {
244 /*=====MAIN=====
245 This is the main entry point for the entire EXAMPLE
246 program.  If you are viewing this with the DB86 debugger
247 try using the Grey- and Grey+ keys to scroll the source
248 file in the view window.  Grey* will restore the view
249 back to the current execution point (horizontal bar).
250 The F1 key brings help.  Alt-<firstletter> for menu.
251 =====
252     int ni;
253     int done = FALSE;
=====
IF          INCLUDE      INVOCATION  ISTEP      KEYBOARD   LABELS
LINES      LIST          LOAD        LSTEP      MENU       MODIFY
MODULES    OBJECTS        OPERATORS  PORT       PROCEDURES PSTEP
REALS      REGISTERS    REPEAT     RSTEP     SCOPE      SETMENU
SOURCE     STACK          TYPES      VARIABLES  VIEWMENU   WINDOWMENU
*
Mod: MENU Proc: MAIN Line: #243 BRK
```

The source code for the MAIN procedure is currently in the View Window. As shown on the status line at the bottom of the screen, the current module is the MENU module; the current procedure is MAIN; and the current line number is 243. The current execution point, line number 243, is highlighted.

**Press these keys
in sequence:**

To do this:

Grey - (several
times)

Scrolls the View Window up a line at a time to go back through the source code in the current module. Note that the thumbmark in the Break Status Column indicates the current line as you scroll through the source code.

Grey + (several
times)

Scrolls the View Window down a line at a time to move forward through the source code in the current module.

PgUp (several
times)

Scrolls the View Window up to go back through the source a screen at a time. Press this key several times to scroll to the start of the menu module, examining the source code as you do.

PgDn (several
times)

Scrolls the View Window down to go forward through the source a screen at a time. Press this key several times to scroll to the end of the menu module, examining the source as you do.

Grey *

Returns the source view and thumbmark to the current execution point (line 243 in this example).

F6

Switches to the View Window as the active window. (The Cycle Window item on the Window Menu also switches the active window; you would type Alt-W C to switch using the menu.) Several more keys are available for scrolling when the View Window is active. Notice that the blinking cursor is now attached to the thumbmark in the Break Status Column. The B appearing next to line 243 indicates that a breakpoint has been set on line 243. The breakpoint was set by the macro file EXAMPLE.MAC when you invoked the debugger. (The GO TIL MAIN line in the macro file actually set this breakpoint.) If you have a color monitor, the active window is indicated by the highlight color as well as the cursor. The Break Status Column is now highlighted instead of the Command Window.

Press these keys in sequence:	To do this:
Home	Jumps to the beginning of the current module.
End	Jumps to the end of the current module.
↑ (several times)	Moves the cursor up to go back through the source a line at a time.
↓ (several times)	Moves the cursor down to go forward through the source a line at a time.
→ (several times)	Scrolls the View Window right by one column.
← (several times)	Scrolls the View Window left by one column.
Ctrl-→	Scrolls the View Window right by 10 columns.
Ctrl-←	Scrolls the View Window left by 10 columns.

Besides scrolling through a single module, you can also browse to different source modules and then scroll through them as well. See Chapters 3 and 4 for a description of browse commands and browse keys (Ctrl-Home and Ctrl-End). See Chapter 2 for an example of browsing. Pressing the Grey * returns you home to the current execution point.

1.5.5 Setting a Breakpoint

Press these keys in sequence:	To do this:
Grey *	Returns to line 243, the current execution point.
PgDn and ↓ (until the cursor blinks on line 269)	Scrolls the View Window and moves the cursor until the second case statement at line 267 appears on the screen and the cursor lines up with line 269.

The screen appears as follows:

```
Debug Window Go Set View Browse Help
254 /*-----
255 Begin Main Menu Keyin Loop.
256 /*-----
257 while(!done) /* loop in main menu until done */
258 {
259 /* get menu item number selected */
260 ni = menu_select(mainxt,mainitems);
261
262 switch (ni)
263 {
264 case MENU_ITEM_ONE:
265 info_menu(); /* show EXAMPLE information screen
266 break;
267 case MENU_ITEM_TWO:
268 blimp_fly(); /* fly a blimp for fun and profit
269 break;
270
271 IF INCLUDE INVOCATION ISTEP KEYBOARD LABELS
272 LINES LIST LOAD LSTEP MENU MODIFY
273 MODULES OBJECTS OPERATORS PORT PROCEDURES PSTEP
274 REALS REGISTERS REPEAT RSTEP SCOPE SETMENU
275 SOURCE STACK TYPES VARIABLES VIEWMENU WINDOWMENU
276 *
277 mod: MENU Proc: MAIN Line: #243 BRK
```

Press these keys
in sequence:

To do this:

Alt-S B

Sets a breakpoint at line 269. Note that the F9 key is listed next to Breakpoint item on this menu. The most often used menu commands can be issued directly with a single key, as a shortcut to using the menu system. The single key is listed next to the corresponding menu item, so you can easily learn the more direct single key controls.

F9 (even number
of times)

Does the same thing as using the menu in the previous step. Toggles the breakpoint on and off. Be sure to leave it toggled on as shown in the screen below. The letter B appears in the Break Status Line next to line 269.

The screen appears as follows:

```
Debug Window Go Set View Browse Help
254 /*-----
255 Begin Main Menu Keyin Loop.
256 -----*/
257 while(!done) /* loop in main menu until done */
258 {
259 /* get menu item number selected */
260 mi = menu_select(maintxt,mainitems);
261
262 switch (mi)
263 {
264 case MENU_ITEM_ONE:
265 info_menu(); /* show EXAMPLE information screen
266 break;
267 case MENU_ITEM_TWO:
268 blimp_fly(); /* fly a blimp for fun and profit
269 break;
B
IF INCLUDE INVOCATION ISTEP KEYBOARD LABELS
LINES LIST LOAD LSTEP MENU MODIFY
MODULES OBJECTS OPERATORS PORT PROCEDURES PSTEP
REALS REGISTERS REPEAT RSTEP SCOPE SETMENU
SOURCE STACK TYPES VARIABLES VIEWMENU WINDOWMENU
*
Mod: MENU Proc: MAIN Line: #243 BRK
```

See Chapters 3, 4, and 5 for descriptions of other ways to set breakpoints. Also see Chapter 2 for further examples of setting breakpoints.

1.5.6 Executing the Program

**Press these keys
in sequence:** **To do this:**

F5 Executes the EXAMPLE program until the breakpoint set at line 269. (The Go Til Breakpoint item on the Go Menu also does this function; you would type Alt-G G to execute the EXAMPLE program using the menu.)

The screen is now controlled by the EXAMPLE program (i.e., the program loaded for debugging). The EXAMPLE program displays the following introductory screen and menu:

```

                WELCOME to the DB86 Debugger Example Program!

This EXAMPLE program uses several separately compiled and linked
modules that illustrate the DB86 debugger's ability to perform source
level debugging across several different program modules. The basic
idea is to use this EXAMPLE program itself as the object of a DB86
tryout session. First, read all the info screens in this program and
then exit back to DOS and try out DB86 by typing TRYIT. The names of
the modules help to indicate the major functions of the EXAMPLE program
that are selected from the main EXAMPLE menu. The MENU module contains
the main menu selection functions and the hex memory dump function.
The INFO module contains functions to select and display various info
screens. The BLIMP module contains the function to fly a blimp on the
screen. The EXAMP10 module contains the I/O primitive functions.

        1 - Show Various DB86 Information Screens
        2 - Fly a Blimp
        3 - Dump Memory in Hex
        4 - Exit this EXAMPLE Program

        Press the number of your selection:

```

**Press these keys
in sequence:**

To do this:

2

Selects the Fly a Blimp example from the menu. This example displays an animated blimp with a message about the debugger.

spacebar

Pressing any key continues the EXAMPLE program once the blimp message is done. But the next step for the EXAMPLE program is line 269 where you had set the breakpoint. The debugger takes control and returns to the debugger screen display with the highlight bar at line 269, the current execution point. At this point, execution of the EXAMPLE program is suspended, and the debugger is back in control.

The screen appears as follows:

```
Debug Window Go Set View Browse Help
254 /*
255 Begin Main Menu Keyin Loop.
256 -----
257 while(!done) /* loop in main menu until done */
258 {
259 /* get menu item number selected */
260 mi = menu_select(maintxt,mainitems);
261
262 switch (mi)
263 {
264 case MENU_ITEM_ONE:
265 info_menu(); /* show EXAMPLE information screen
266 break:
267 case MENU_ITEM_TWO:
268 blimp_fly(); /* fly a blimp for fun and profit
269 break;
```

IF	INCLUDE	INVOCATION	ISTEP	KEYBOARD	LABELS
LINES	LIST	LOAD	LSTEP	MENU	MODIFY
MODULES	OBJECTS	OPERATORS	PORT	PROCEDURES	PSTEP
REALS	REGISTERS	REPEAT	RSTEP	SCOPE	SETMENU
SOURCE	STACK	TYPES	VARIABLES	VIEWMENU	WINDOWMENU

```
*
Mod: MENU Proc: MAIN Line: #269 BRK
```

1.5.7 Showing User Output

Press these keys To do this:
in sequence:

F4 You can switch between the debugger screen and the screen from the program you are debugging. This way, you can observe output from the user program. (The Flip Screen item on the Window Menu also switches the screens; you would type Alt-W F to flip the screen using the menu.)

1.5.8 Exiting from the Debugger

**Press these keys
in sequence:**

To do this:

F6

Switches back to the Command Window.

At the prompt, enter the EXIT command to return to the operating system:

```
*exit
```

You can also exit using the Exit DB86 item on the Debug Menu by typing Alt-D E.

This command concludes the quick tour of the debugger. Chapter 2 has more extensive examples of debugger features, including a session where you will track down a bug in the EXAMPLE program you have just toured.

A DOS-executable version of the EXAMPLE program is provided in the EXAMPLE.EXE file on the distribution disk. You can run it by entering EXAMPLE at the DOS prompt:

```
C:\DB86>example
```

You may wish to run the program to see if you can notice what the bug is. Hint: The bug is not in the Fly-a-Blimp procedure.

Contents

Chapter 2 Debugging with DB86

2.1	Preparing a Program for Debugging.....	2-2
2.2	Invoking the Debugger	2-6
2.3	Loading a Program.....	2-8
2.4	Setting Breakpoints.....	2-10
2.4.1	Tracepoints.....	2-10
2.4.2	Breakpoints.....	2-12
2.4.3	Breaking on Access to Data	2-13
2.4.4	Other Breakpoint Commands	2-14
2.5	Executing a Program.....	2-15
2.6	Observing the Output of a Program	2-19
2.7	Examining and Modifying the Program	2-19
2.8	Examining and Modifying Data.....	2-22
2.8.1	Simple Ways to Display Data	2-22
2.8.2	Advanced Techniques for Displaying Data.....	2-23
2.8.2.1	Watch Expressions.....	2-24
2.8.2.2	Dereferencing Pointers	2-24
2.8.2.3	The EVAL Command.....	2-25
2.8.2.4	Dumping Memory.....	2-25
2.8.2.5	Assembly-level Display of Data	2-26
2.8.3	Modifying Data.....	2-26
2.9	Configuring the Debug Environment	2-27
2.10	Finding a Bug.....	2-28

()

()

()

Chapter 2 Debugging with DB86

This chapter describes typical debugging tasks and strategies for using the debugger to accomplish these tasks. Examples are included throughout to illustrate the steps involved. The tasks include:

- Preparing a program for debugging
- Invoking the debugger
- Loading a program
- Setting breakpoints
- Executing a program
- Observing the output of a program
- Examining and modifying a program
- Examining and modifying data
- Configuring the debug environment
- Finding a bug in a program

```
Debug Window Go Set View Browse Help
238 1
239 1      return(key);          /* return with i
240 1      }
241
242      void far main()
243 1      {
244 1      /*-----MAIN-----
245 1      This is the main entry point for the entire EXAM
246 1      program. If you are viewing this with the DB86
247 1      try using the Grey- and Grey+ keys to scroll the
248 1      file in the view window. Grey* will restore the
249 1      back to the current execution point (horizontal
250 1      The F1 key brings help. Alt-(firstletter) for n
251 1      -----
252 1      int ni;
253 1      int done = FALSE;
                                     DB86
                                     U1.0
DOS 3.28 (844-D) DB86 V1.0
Copyright 1983,1988 INTEL CORPORATION
*load example.86
*go til main
[ Break at :MENU#243 ]
*
Mod: MENU Proc: MAIN Line: #243                                ERR
```

Figure 2-1 DB86 Screen with EXAMPLE Program Loaded

2.1 Preparing a Program for Debugging

As shown in Figure 1-2 (see Chapter 1), the first three steps in preparing a program for debugging are editing the source file, compiling it, and linking it. If there is just a small amount of source text, you can place it all in a single file, compile it (using the compiler's DEBUG control), and link it (using the linker's BIND control).

Most often though, the source code is too elaborate to place in a single source file. In this case, edit separate source files, compile each one separately using the DEBUG control, and then link them together using the BIND control. Typically, a make file or a batch file can be used to automate the separate compiles and the linking.

To illustrate this process, see Figure 2-2. The batch file shown here compiles and links the EXAMPLE program used in the quick tour in Chapter 1. The EXAMPLE program consists of several modules:

sysint	is a low-level, interface subroutine written in ASM86. It provides an interface to the DOS system interrupts and connects the I/O functions in the exampio module to the operating system.
exampio	provides a set of I/O services which the other modules use to do screen and keyboard I/O.

The remaining modules make up the parts of the program that the user sees:

menu	is the main module and provides the C function main. This module includes the menu selection for each subroutine in the example. It also includes the subroutines called to do the memory dump (menu item 3) and to exit from the EXAMPLE program (menu item 4).
info	is called to display various information screens about the debugger (menu item 1).
blimp	is called to display the blimp on the screen (menu item 2).

Each module of EXAMPLE is a separate source file that is compiled or assembled as illustrated in Figure 2-2.

```

echo off
rem
rem -----
rem MAKEIT.BAT - This batch file performs the necessary compiles
rem and links to make the EXAMPLE.86 and EXAMPLE.EXE files for
rem debugging with the DB86 source-level, symbolic debugger. This
rem batch file assumes that you have the Intel ASM86, iC-86,
rem LINK86, and UDI2DOS utilities available in the current DOS
rem path and that you have the source files for all the example
rem modules in the current DOS directory.
rem
rem Do the Assembly-language module:
rem
    echo on
rem ===== asm86 sysint.a86
    asm86 sysint.a86 object(sysint.obj) debug
    echo off
    if errorlevel 1 goto end
rem
rem Do the C module compiles:
rem
rem The compiler invocation controls have the following meaning:
rem object - means produce an object file with the specified name.
rem ex - means extended keywords are recognized.
rem la - means compile with large memory model.
rem debug - means produce object file with debug information in it.
rem
    echo on
rem ===== ic86 exampio.c
    ic86 exampio.c object(exampio.obj) ex la debug
    if errorlevel 1 goto end
rem ===== ic86 menu.c
    ic86 menu.c object(menu.obj) ex la debug
    if errorlevel 1 goto end
rem ===== ic86 info.c
    ic86 info.c object(info.obj) ex la debug
    if errorlevel 1 goto end
rem ===== ic86 blimp.c
    ic86 blimp.c object(blimp.obj) ex la debug
    if errorlevel 1 goto end
    echo off

```

Figure 2-2 DOS Batch File for EXAMPLE Program: Part 1

```
rem
rem Now link all the object files together.
rem
    echo on
    rem ===== link86
    link86 callmain.obj, & <example.lrf
    echo off
rem
rem Finally, make a DOS-executable file.
rem
    echo on
    udi2dos example.86
rem
:end
```

Figure 2-2 DOS Batch File for EXAMPLE Program: Part 2

The batch file includes a step that illustrates linking the modules together with LINK86.

In the batch file, note that list files are not produced by the C compiler. For PL/M-86, Fortran-86, and Pascal-86 compilers, the debugger gets its source code for the View Window from the .LST file. So, when compiling with these languages, include the PRINT control and create a .LST file. The debugger can use C source files directly, so the debugger does not need the .LST files. The .LST extension is the default when the debugger attempts to find the source files. Thus, if you want to use the C source files during debugging, change the default extension to .C (or whatever extension you use for C source files). Use the Source Path entry in the Debug menu or use the SPATH command to make this change. See the EXAMPLE.MAC file for an example of using the SPATH command with C source files.

Another technique to note in the batch file is the use of a Link Response File (LRF). See Figure 2-3. The LINK86 command line from the batch file is as follows:

```
link86 callmain.obj, & <example.lrf
```

```
sysint.obj, & /* system interface function */
exampio.obj, & /* screen & keyboard I/O functions */
menu.obj, & /* main menu module for example program */
info.obj, & /* show selected help info */
blimp.obj & /* fly a blimp */
to example.86 bind ss(memory(+5000), stack(+5000))
```

Figure 2-3 Link Response File for EXAMPLE Program

The file CALLMAIN.OBJ is the first object file to be linked. CALLMAIN contains a single instruction of startup code (i.e., a call to the main procedure).

The file EXAMPLE.LRF contains the remainder of the LINK86 command line including the rest of the files to be linked, the output file specification, and the LINK86 controls BIND, SS, and STACK.

The BIND control tells the linker to produce a loadable program file (EXAMPLE.86) as output.

The SS and STACK controls make a larger stack segment. The default size of the stack segment created by LINK86 is sometimes not large enough if the C program makes use of I/O (e.g., as provided in the C libraries).

Do not use the NOTYPE control during compiles, assemblies, or links. Do not use the PURGE control during links. These controls make user-program symbols and data inaccessible during debugging.

The latest version of LINK86 can produce DOS-executable (.EXE) files directly. Thus, the UDI2DOS step shown in the batch file can also be accomplished with another invocation of the linker. Note that the .EXE files produced by the linker cannot be loaded and debugged with DB86.

If your program uses overlays, you must specify the NOGO (No Group Overlay) control on the final link.

Now that EXAMPLE.86 has been compiled and linked, it is ready to be debugged.

2.2 Invoking the Debugger

The next step in debugging the program is to invoke the debugger.

There are a number of command-line controls for the debugger. See Appendix B for more information on these controls. This section contains several typical examples of debugger invocation lines.

The simplest way to invoke the debugger is to enter the command name at the DOS prompt:

```
C:\DB86>db86
```

Then, use the menu system to load user programs to be debugged.

You can also invoke the debugger and load the user program from the command line:

```
C:\DB86>db86 myprog.86 cmdopt1 cmdopt2
```

Note that CMDOPT1 and CMDOPT2 are not debugger controls; they are controls for the MYPROG program. Collectively, they are referred to as the load file tail for the program being debugged. The load file tail is ignored by the debugger and preserved for use by the program being debugged. Thus, debugger controls must be specified on the command line before the filename of the loadable program. The name of the loadable program and its load file tail are the last items that can appear on the command line:

```
db86 db86opt userprog load_file_tail
```

The first token on the command line that is not identifiable as a debugger control is interpreted as the name of the user program to be loaded. All tokens after that are interpreted as the load file tail for the user program.

In the next example, a program is loaded from the command line and the debugger is set up for use with a 43-line EGA video adapter.

```
C:\DB86>db86 143 example.86
```

The L43 control requires an EGA video adapter; otherwise, it is ignored. With this option, the debugger displays 43 lines allowing you to see more source lines in the View Window. In this command line, the program EXAMPLE.86 is loaded.

In the next example, the MON2 debugger control sets up the debugger to use a second monitor: one for debugger screens and the other for user-program output. See Appendix C for more information on using a second monitor.

```
C:\DB86>db86 mon2 macro(example.mac)
```

The macro file EXAMPLE.MAC is also included on the command line in this example. It contains debugger commands that are automatically executed as if they had been entered at the prompt in the Command Window. Note that a load program is not specified on the command line. In fact, the EXAMPLE.MAC file has the LOAD command line in it. Macro files provide a convenient way to set up a debugging environment that is unique to a given program. In this case, EXAMPLE.MAC contains the following commands:

```
spath = .\,.c,.lst
load example.86
go til main
```

The SPATH command sets the source path to the current directory (.\) and defines two source file extensions (.C and .LST). The LOAD command loads the EXAMPLE program, and the GO command implicitly sets a breakpoint at the main function and begins executing the EXAMPLE program until that breakpoint is reached.

If a macro file is not specified on the command line, the debugger automatically looks for a default macro file named DB86.MAC. If this file is present, the debugger executes it as if it had been specified on the command line. The default macro file provides a convenient way to set up a global debugging environment that is not unique to a given program (e.g., commands to define the number base or to set the overlay break flag).

The default macro file can be disabled by using the NOMR control on the debugger command line:

```
C:\DB86>db86 nomr
```

2.3 Loading a Program

Previous examples have illustrated how to load a program directly on the debugger command line and indirectly by including the load command in a macro file that is specified on the command line.

Once the debugger has been invoked, you can still load a program to be debugged using the LOAD command at the debugger command-line prompt.

You can also load a program using the menu system by selecting the Load item from the Debug Menu (press Alt-D L).

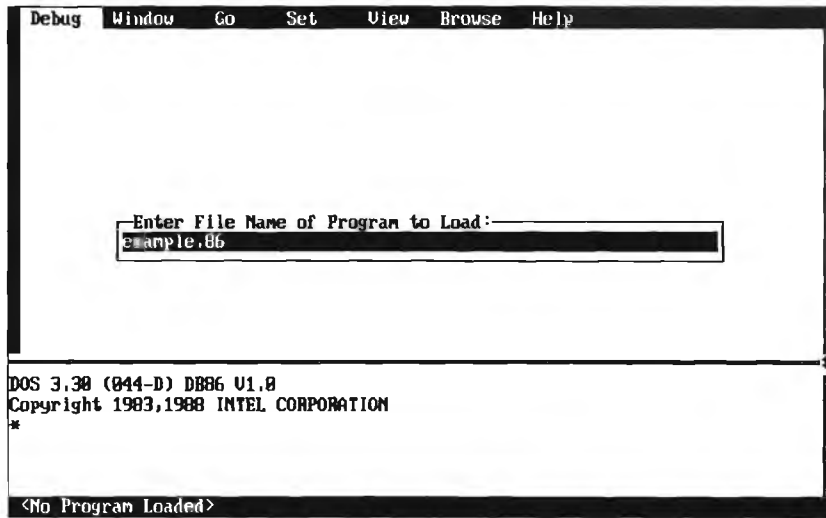
First, invoke the debugger as follows:

```
C:\DB86>db86
```

Press these keys in sequence:	To do this:
--	--------------------

Alt-D L	Pulls down the Debug Menu and selects the Load item.
---------	--

A pop-up menu is displayed on the screen for entering the name of the file to be loaded. Enter the filename EXAMPLE.86 as shown in the following screen. While you are entering the name, you can use a standard set of editing controls. See Chapter 4 for more information on line-editing key controls.



The EXAMPLE program is loaded. Note that if the program you are loading requires a load file tail, you can type it after the load filename.

Another way to load programs once you have invoked the debugger is to select the Reload item from the Debug Menu. This command (press Alt-D R) reloads the most recently loaded program. The most recent load file tail is preserved.

Press these keys in sequence:	To do this:
--	--------------------

Alt-D E	Exits the debugger and returns to DOS.
---------	--

2.4 Setting Breakpoints

Debugging involves controlling the execution of the user program: running it, stopping it, examining data while it is stopped, and then running it again.

To help with these activities, the debugger provides tracepoints, breakpoints, and watchpoints.

2.4.1 Tracepoints

A tracepoint is a location in the user program. When execution reaches that location, the program momentarily returns control to the debugger, which then displays an announcement in the Command Window. Then, the debugger resumes execution of the user program. Every time execution reaches the tracepoint, the announcement is displayed in the debugger Command Window.

A tracepoint can be defined using the menu system, using function keys, or using a command line. The Tracepoint item in the Set Menu (press Alt-S T) sets a tracepoint at the current source line. Scroll through the source moving the thumbmark until you reach the line where you want to set a tracepoint. Then, press Alt-S T to set it. The sF9 function key also sets a tracepoint at the current source line. The letter T appears in the Break Status Column next to the source line where the tracepoint is set.

In the Command Window, the TR command can be used to set a tracepoint at a specified location. With the TR command, you can also define an expression that is evaluated every time the tracepoint is reached. The result of the expression is included as part of the trace announcement. For example, invoke the debugger using the TRYIT batch file:

```
C:\DB86>tryit
```


Enter the following command at the Command Window prompt to set a tracepoint:

```
*tr0=:menu#265,'mi=',:menu.main.mi
```

In this example, the tracepoint is set at line 265 in the MENU module. This line is the first case in a switch statement. The variable `mi` is the menu item chosen; this item is used to select one of the cases. In other words, if the first menu item is selected, the variable `mi` is set to 1 and the first case in the switch statement is executed.

Press these keys in sequence:	To do this:
GO TIL #266	Begins executing the program.
1	Selects the subroutine that shows various information screens.
Esc	Escapes from the subroutine.

In the Command Window, the string `mi=` is displayed as part of the trace announcement followed by the value of the variable `mi`. The trace announcement displayed in this case follows:

```
*[ Trace at :menu#265 ] mi= +1
```

At a more advanced level, you can log Command Window activity to a file using the LIST command and, thus, capture the trace announcements, including the results of any expressions, for later examination. See the LIST command in Chapter 5 for more information.

Press these keys in sequence:	To do this:
Alt-D E	Exits the debugger and returns to DOS.

2.4.2 Breakpoints

A breakpoint is a location in the user program. When execution reaches that location, the program stops executing and control is restored to the debugger.

The debugger provides several kinds of breakpoints: temporary breakpoints, fixed breakpoints, and conditional breakpoints.

Temporary breakpoints support the point-and-shoot technique of debugging. Usually, they are set by scrolling to the source line where the breakpoint is to be set and then pressing the F9 key to set it. The F9 key also toggles a temporary breakpoint off if one is already set on the current source line. A temporary breakpoint is indicated by the letter B in the Break Status Column next to the source line where the breakpoint is set.

Using the menu system, you can set a temporary breakpoint by selecting the Breakpoint item in the Set Menu (press Alt-S B). Temporary breakpoints are cleared by the next GO TIL command.

The debugger provides more powerful control over the execution of a user program through fixed breakpoints and conditional breakpoints. Both of these kinds of breakpoints are set using commands in the Command Window.

A fixed breakpoint is defined with the FB command. These named breakpoints are debug objects and are permanently set. They can be enabled and disabled with the ENABLE and DISABLE commands. A fixed breakpoint is indicated by the letter F in the Break Status Column.

A conditional breakpoint is defined with the CB command. Like the fixed breakpoint, it is a named debug object. It also has a boolean expression defined with it to determine if the breakpoint is taken or not. If the specified expression is true when execution reaches the breakpoint, then the breakpoint is taken, and control is restored to the debugger. If the expression is false when execution passes the breakpoint, the breakpoint is not taken and user-program execution continues. A conditional breakpoint is indicated by the letter C in the Break Status Column.

For example, if you set the following conditional breakpoint at line 268 of the MENU module, a breakpoint is taken only if the program variable `mi` is equal to 2. Otherwise, execution continues and the breakpoint is not taken.

```
*cbl = :menu#268,mi==2
```

All breakpoints can be cleared by selecting the Remove All item in the Set Menu.

2.4.3 Breaking on Access to Data

The other breakpoint commands are associated with a location in memory, which you must specify when you set the breakpoint.

Often, however, there is a need to stop execution of the user program when a variable has reached a certain value. In this case, you typically do not know in advance the memory address or source-program location when this event occurs. In fact, most often, you are trying to find the program location where the data is being changed.

The debugger provides a strategy for breaking on access to data by combining a conditional breakpoint with auto stepping. This feature is commonly called a watchpoint.

A watchpoint is a way to stop program execution when a data item that you are interested in reaches a value that you specify.

First, set a conditional breakpoint defining an expression that returns a true if the data has a specified value and a false if the data does not have that value. For example, if you want to know when a variable is greater than 0, you would define a Boolean expression that returns a true when the variable is greater than 0 (e.g., `var > 0`).

Typically, you would set the breakpoint at the a location that is within the scope of the variable. For example, if it is a global variable, you can set the breakpoint at the start of the module; if it is a local variable, you can set the breakpoint at the start of the procedure where the variable is defined. The auto step feature is used for program execution. During auto stepping, the conditional breakpoint is evaluated at every step.

Then, select the Auto Step item from the Go Menu (press Alt-G A). When the auto stepping begins, the conditional expression is evaluated at every step. If the expression is true, a breakpoint is taken; if the expression is false, the user program continues auto stepping.

Thus, you can discover where in your program the data is changed to the specified value.

2.4.4 Other Breakpoint Commands

Temporary breakpoints can be cleared with the Clear At Cursor item or the Remove All item, both on the Set menu. The F9 key alternately sets and clears a breakpoint at the current line.

A temporary breakpoint is implicitly set by the GO TIL command after all other temporary breakpoints are cleared.

Fixed breakpoints, conditional breakpoints, and tracepoints can be enabled with the ENABLE command and disabled with the DISABLE command.

The Break Status Column in the View Menu displays any breakpoints that are currently set. This same information is provided by the DIR command.

The debugger has several built-in breakpoints. First, it can automatically break whenever an overlay is loaded, depending on the value of the OBREAK flag. See Chapter 5 for more information on the OBREAK flag.

The other built-in breakpoints are controlled by the value of the SYSINT flag. See Chapter 5 for more information on SYSINT. For example, the debugger can automatically break whenever the user program attempts to exit and return to the operating system. Thus, you can reload the program and continue debugging. Another built-in breakpoint controlled by SYSINT is the SysReq key. Pressing SysReq provides a way to break a runaway program. For instance, if the program is in an infinite loop, press SysReq to stop it and return to the debugger. The Ctrl-Break key does the same thing as the SysReq key.

See Chapter 5 for more information on FB, CB, TR, and GO TIL.

2.5 Executing a Program

Besides setting breakpoints, the most common debugging activity is executing a user program.

The Go Menu contains many of the go and step commands for executing a program. Most of these commands have corresponding function keys as well.

In the Command Window, you can use a more powerful version of the GO command. Various forms of step commands are available as well.

The most commonly used commands for executing a user program are the commands to step and to execute the program until the current line is reached (go til cursor).

With one debugging strategy, you scroll and browse through the source code in the View Window examining the code. At some point during scrolling and browsing, you read a source line that is a good candidate for the error. You then back up to a previous line and press the F7 key to execute to a point just before the point where the bug might appear. Following this strategy, you sequence through the program getting closer and closer to the point where the bug actually is.

The F7 key is the same as the Til Cursor Line item on the Go Menu (i.e., Go Til Cursor Line).

With another debugging strategy, you scroll and browse through the source code setting temporary breakpoints at several locations by pressing the F9 key. Then, you execute the program with the F5 key (the same as Go Til Breakpt item on the Go Menu). If execution reaches any of the breakpoints set in the source code, the program stops and returns to the debugger. At this point, you can check data and program output to determine if the bug has occurred yet, gradually closing in on the bug. This strategy is more of a shotgun approach. You start out with widely separated breakpoints and gradually narrow down to the bug by reducing the separation of the breakpoints.

The debugger supports both of these techniques; usually both strategies are combined when debugging a program.

The F8 key (line step) executes a single line of source at a time and the F10 key executes a single procedure at a time. Both of these commands are also used frequently during debugging.

If you are executing a program that gets caught in a loop or does not pass through a breakpoint location, you can always stop execution and return to the debugger with the SysReq key (or Ctrl-Break).

Type the following command at the DOS prompt to invoke the debugger:

```
C:\DB86>tryit
```

This command loads the EXAMPLE program and runs it until the temporary breakpoint at the main procedure is reached.

Press these keys in sequence:	To do this:
Grey + (to line 272)	Scrolls the View Window until the thumbmark is set on line 272.
F9	Sets a temporary breakpoint at this line.
F5	Executes the program until the breakpoint is reached.
3	Selects the third menu item to Dump Memory in Hex.
00000400	Specifies the address of 0000:0400 as the starting location of the memory dump.
Esc	Exits from the memory dump subroutine and takes the breakpoint returning to the debugger.
F5	Executes the program again.
3	Selects the Dump Memory in Hex menu item.
B0008000	Specifies the address of B000:8000 as the starting location of the memory dump. (This location is the address of the memory used by the CGA video adapter under DOS.)

Note that the screen display now looks like this:

```

==== Hexadecimal/ASCII Memory Dump ====
0000:0000 6C 05 EB 69 C6 EB C4 F8 FA C6 06 CF 02 00 C6 06 |--j-----
0000:0010 D0 02 FF C6 06 05 03 00 0F 06 1A 05 0F 06 1C 05 -----0--a--n-
0000:0020 0E 1E DE 02 0E 16 30 00 0B 26 2E 00 0B 6E BC 2E ---XXX--P--6-
0000:0030 A3 1E 05 58 58 58 00 02 F2 50 2E FF 36 1C 05 2E -6--P-----
0000:0040 FF 36 1A 05 2E A1 1E 05 CF 50 0B FE 0D 2D EE 0D -S-----P-----
0000:0050 3B F8 0B C7 73 07 2E 0A 05 EE 0D 32 E4 2E A3 D2 ;--s--z--
0000:0060 02 58 56 BE 9E 0D 09 70 0D 5E C3 26 01 7E 0D F6 -XU--x--^--
0000:0070 0F 73 0B 01 FB F0 0F 74 04 01 FB F0 0F C3 03 FB -S-----t-----
0000:0080 F8 C3 26 3B 5E 0D 77 26 E0 0A 00 72 1E 0B 3D 75 -a--^--u--r--u
0000:0090 0E 26 0B 76 0D 01 FE F6 0F 72 0C 0B FF EB 0C D1 -a--u--r-----
0000:00A0 EF D1 EF D1 EF D1 EF 01 E7 FF 0F 16 1F C3 26 C7 -----a-----
0000:00B0 46 1E FF FF 50 04 00 36 C6 06 F7 02 0B BF FF 0F F--P--6-----
0000:00C0 E8 0A FE 3C 03 F8 75 01 F9 5B C3 EB 77 00 72 DB ---<--u--X--u--r-
0000:00D0 0B 35 74 0C 51 B1 04 D3 E2 59 01 E6 0F 00 EB 14 -5t--Q--v-----
0000:00E0 26 01 7E 0D F6 0F 73 0A 01 E6 00 F0 01 E2 FF 0F a--s-----
0000:00F0 EB 02 33 F6 0B F2 09 35 36 C5 36 74 05 00 4C 05 --3--56-6t--L-

==== N - Next Screen; P - Previous Screen; ESCape:

```

Notice that the memory addresses being displayed do not correspond to the addresses specified; the addresses shown are in the lower 64K bytes of memory. If you want to, you can continue pressing F5 to run the program, select item 3, and try typing various address values to confirm this bug.

Press these keys in sequence:	To do this:
Esc	Exits from the memory dump subroutine and takes the breakpoint returning to the debugger.
Grey - (to line 103)	Scrolls back to line 103 in the source module.
F7	Executes the program until the current line (103 in this case).
3	Selects item 3 from the menu.

**Press these keys
in sequence:**

To do this:

F0008000

Specifies memory address F000:8000 as the starting address for the memory dump. (This location is in the BIOS ROM on the PC computer.)

The value entered (F000:8000) is stored in a variable called ptr. You can further confirm the bug by typing the name of the variable in the Command Window:

```
*ptr  
0000H:8000H
```

The debugger responds by displaying the value of ptr. Note that, although you entered F000:8000, the value of ptr is 0000:8000.

This is definitely a bug. Finding this bug is the topic of the last section of this chapter. First, there are a few more commands to cover.

**Press these keys
in sequence:**

To do this:

Alt-D E

Exits the debugger and returns to DOS.

2.6 Observing the Output of a Program

The user program that is being debugged may produce screen displays as output. The debugger provides a way to share the screen with the user program and avoid direct conflicts by allowing you to flip between the user program screen and the debugger screen.

The F4 key alternately flips between the debugger screen and the user program screen. The Flip Screen item on the Window Menu also accomplishes this function.

Screen flipping is accomplished by the debugger in different ways depending on the video adapter used in the system. See Appendix B for more information.

You can also run the debugger with two video adapters and two separate monitors. With this setup, you can simultaneously show the user program screen on one monitor and the debugger screen on the other. See Appendix C for more information.

2.7 Examining and Modifying the Program

The two main debugger commands for examining the source program are browsing and scrolling.

With scrolling, you can navigate through the current source module. See Chapter 4 for complete information on all the navigation keys.

With browsing, you change to another source module. You can browse through the set of all the modules in the program or you can browse according to the modules on the callstack.

Invoke the debugger with the TRYIT batch file:

```
C:\DB86>tryit
```

**Press these keys
in sequence:** **To do this:**

Ctrl-End (5 times) Starts browsing through the set of modules. Note that the status line changes to reflect the module that you are currently browsing through. Stop when you get to the exampio module. Ctrl-Home browses back through the module set in case you miss it.

Grey + (to line 116) Scrolls to line 116.

At this point, the screen looks like the following:

```
Debug Window Go Set View Browse Help
111 REGSTRUCT reg: /* register structure */
112
113 reg.bx = 0;
114 reg.dx = (row << 8) + col;
115 reg.ax = VIDEO_PUTCUR; /* video write char c */
116 video(&reg);
117 }
118
119 /*=====
120 clear screen.
121 =====
122 void far cls()
123 {
124 REGSTRUCT reg; /* register structure */
125
126 reg.bx = 0x0700;

Copyright 1983,1988 INTEL CORPORATION
*spath = ,\,c,,lst
*load example.86
*go til main
[ Break at :MENU243 ]
*
[Browsing] Mod: EXAMPIO Proc: TOUPPER Line: #24
```

Press these keys in sequence:	To do this:
F7	Executes the program until line 116 of the <code>exampio</code> module.
Alt-V C	Displays the current callstack. Note that the program is currently executing several levels deep in the callstack.
spacebar	Pressing any key continues.
Alt-B C	Turns on callstack browsing. Note that the status line at the bottom of the screen displays a message indicating that callstack browsing is turned on.
Ctrl-End (several times)	Browses through the modules in order as they appear on the callstack. Note that as you browse from module to module the module name on the status line changes. Note also that the View Window thumbmark is set to the returning location. Thus, you can easily back out of a series of calls and set a breakpoint to stop after those calls have completed and returned. Pressing Ctrl-Home takes you back through the callstack modules.
Grey *	Returns to the current execution point from browsing or callstack browsing.
Alt-D E	Exits from the debugger.

The Browse Menu also contains three search items (Find, Next Find, and Previous Find). The debugger searches the current source module for the specified string.

The Scope item on the Browse Menu allows you to browse directly to the module and procedure or line number that you specify instead of using the Ctrl-Home and Ctrl-End navigation keys.

At a more advanced level, you may need to debug at the assembly-language-instruction level. The debugger provides the ability to view source with disassembled code by pressing the F3 key. In addition to showing disassembled code, you can toggle on the Register Window and display the processor registers by pressing the F2 key.

Using the Command Window, you can enter the ASM command to dump memory as disassembled code. This command provides a similar view of your code as toggling on the disassembled display in the View Window.

To modify and patch the program, use the ASM command to assemble directly into memory. See Chapter 5 for more information on this powerful feature.

2.8 Examining and Modifying Data

Examining and modifying data are important aspects of debugging. Many different ways of looking at data are provided by the debugger.

2.8.1 Simple Ways to Display Data

At the simplest level, the Expanded Calls item on the View Menu provides a way to look at the local variables in procedures that appear on the callstack. The Locals item on the View Menu lists the values of variables in the current procedure.

In the Command Window, you can enter the symbol name and the debugger shows its value in the current number base. If the variable is not in the currently executing procedure, you may have to qualify the reference by prefixing the module name and/or procedure name.

You can prefix the variable name with the dot operator to show the address of the variable as opposed to its value.

For example, if `sysreg` is the name of a structure in a C program with a tag of `REGSTYPE`, then the following command shows the value of that structure:

```
*sysreg
REGSTYPE (structure
  AX  0200
  BX  0000
  CX  4848
  DX  0000
  SI  47D1
  DI  0010
  DS  1446
  ES  4540
)
```

With the dot operator, the debugger returns the address of the `sysreg` structure:

```
*.sysreg
4540H:1426H
```

The `DIR` command entered at the command line shows symbol names and types.

2.8.2 Advanced Techniques for Displaying Data

As described earlier, watchpoints and tracepoints both help you to monitor data. Watchpoints, using the conditional breakpoint and auto step features of the debugger, allow you to break when an expression is evaluated to true thus providing a further level of control over examining data. Tracepoints defined in the Command Window with an expression as part of the announcement provide yet another way to examine user-program data.

2.8.2.1 Watch Expressions

Watch expressions, defined at the command line with the WA command and then viewed in the Watch Window, provide an effective way to observe program data. The watch expression can contain program variables or conditions and is evaluated at every breakpoint, every step, and at every tracepoint. The value is automatically updated and displayed in the Watch Window, so you can monitor and observe changes in the program data.

2.8.2.2 Dereferencing Pointers

One of the most important advanced methods of displaying program data uses the technique of dereferencing a pointer. For example, if you wanted to view the sysreg structure, but did not know its name, you might still be able to display it using the dereferencing operator.

If you knew that a pointer named regp pointed to the sysreg structure, you could dereference the pointer to display the structure:

```
*regp
4741H:0157H
*regp^
REGSTYPE (structure
  AX  0200
  BX  0000
  CX  4848
  DX  0000
  SI  47D1
  DI  0010
  DS  1446
  ES  4540
)
*
```

The first command displays the value of the pointer (i.e., an address). The second command (regp^) displays the structure pointed to by the pointer.

The dereferencing operator can be used as part of expressions to provide a powerful method for displaying program data.

Dereferencing can be combined with other operations to select and view data. For example, if you want to display the value of a member of a structure but do not know the structure name, you can dereference a pointer to the structure and still access the structure member:

```
*regp^."ax  
0600
```

In this example, the value of `ax`, a member of a structure pointed to by `regp` is displayed. Note that the double quotes before `ax` are used to access `ax` from the user program instead of the reserved word `AX` which is the name of a processor register. Normally, the double quotes are not needed. See Appendix F for a list of reserved words.

2.8.2.3 The EVAL Command

Another advanced command for displaying data is the `EVAL` command. This command evaluates an expression that you specify and displays its value in all number bases and as a string. The expression can contain debug objects as well as program symbols and variable names. If you evaluate an address, the `EVAL` command displays the module and procedure the address is in. See Chapter 5 for more information.

2.8.2.4 Dumping Memory

Another feature that the debugger provides for examining data is the ability to dump memory through its `mtype` commands. By specifying a type, followed by an address (or symbol name that evaluates to an address), followed by a length, you can display blocks of memory using the `mtype` as a template for displaying the memory. For example, the following command displays eight bytes of memory as bytes starting at the address of line 168:

```
*byte #168 length 8
```

The following command displays three pointers from memory:

```
*pointer argv length 3
```

See Chapter 5 for more information on `mtype` commands.

2.8.2.5 Assembly-level Display of Data

At times, it may be necessary to debug at the assembly-language level. The debugger provides the F3 key to toggle disassembly display. And you can toggle on the Register Window (F2) to display registers and flags.

2.8.3 Modifying Data

The debugger provides an assignment command so that you can modify data in the program you are debugging.

For example, you can assign a new value to a loop counter. The new value might reset the counter to start the loop over again, or it might change the value so that the loop ends. In the Command Window, enter the symbol name followed by an equal sign and the new value. In the following example, *i* is a loop counter and you want to assign a new value to *i* so you can get out of the loop and continue executing elsewhere:

```
*i=9999
```

In another case, you can find a bug in a program where you have used the wrong pointer variable. By reassigning the correct value to the pointer, you can confirm the bug.

You can also change the values of registers since the debugger has built-in names for them. Note, however, that changing register values can produce unexpected results (e.g., if you change the instruction pointer or reset the stack pointer).

2.9 Configuring the Debug Environment

The debugger includes many features for configuring the debug environment. Macro files allow you to set up a global debug environment or to customize the environment for debugging a particular program.

Include files serve the same purpose as macro files. They are text files that contain debug commands. When you use the INCLUDE command to process one of these files, the commands are executed as if they were entered in the Command Window. With this facility, you can automate portions of the debugging effort.

The LIST command lets you save the Command Window screen display so you can keep a record of the debug session. You save the display to a text file which you can examine later. The logging feature also allows you to save trace announcements in a file to be examined later.

The debug environment can be further configured with command-line (invocation) controls that allow you to set preferences and take advantage of additional video adapters. See Appendix B for more information.

2.10 Finding a Bug

In this section, you can follow along with the steps taken to find the bug in the EXAMPLE program.

As discovered in a previous example, the bug is in the Dump Memory subroutine. No matter what you specify as the selector part of the address, the subroutine uses 0000. Thus, you cannot dump memory above the first 64K bytes.

First, invoke the debugger:

```
C:\DB86>tryit
```

**Press these keys
in sequence:**

To do this:

Grey + (to line
271)

Scrolls to line 271. Keep pressing the Grey + until the thumbmark is at line 271.

F7

Executes the EXAMPLE program starting at the current execution point until the current line (271 in this case) is reached. The program displays the main menu before the breakpoint is reached.

Another way to execute to line 271 involves two steps. First, use the F9 key to set a temporary breakpoint at the current line and, second, use the F5 key to execute to the breakpoint. The F7 key combines these two steps, setting a temporary breakpoint at the current line and then executing the program.

3

Selects the memory dump subroutine from the menu. Then, the program reaches line 271 and breaks.

At this point, the screen looks like the following:

```
Debug Window Go Set View Brouse Help
266      break:
267      case MENU_ITEM_TWO:
268          blimp_fly();          /* fly a blimp for fun and profit
269      break:
270      case MENU_ITEM_THREE:
271      menu_memdump();          /* dump memory in hex */
272      break:
273      case MENU_ITEM_FOUR:
274      case MENU_EXIT:
275          done = TRUE;          /* ESCape key pressed */
276      break:
277      default:
278          beep();              /* invalid selection, so beep */
279      break:
280      }
281      }
```

```
Copyright 1983,1988 INTEL CORPORATION
*spath = .\,c,.lst
*load example.86
*go til main
[ Break at :MENU243 ]
*
Mod: MENU Proc: MAIN Line: #271 BRK
```

**Press these keys
in sequence:**

To do this:

F9 (several times
leaving the
breakpoint off)

Toggles the temporary breakpoint off and on. As the breakpoint is toggled on and off, the letter B appears and disappears in the Break Status Column next to line 271. Note that the Break Status Column is to the left of the View Window.

F8

Steps to the next source line in the program which is inside the menu_memdump subroutine. The F10 key also steps, but it steps over a procedure and not into it. The F10 key executes the called subroutine and breaks after it returns while F8 breaks at the first instruction inside the subroutine.

The View Window is updated to display the new current location:

```
Debug Window Go Set View Browse Help
86
87 /*-----
88 menu_memdump - ask for start paragraph and dump memory in hex.
89 -----
90 void menu_memdump()
91 {
92     int done = FALSE;
93     char *ptr;
94     unsigned int key;
95
96     cls();
97     putcur(12,5);
98     stout("=== HEX/ASCII Memory Dump ===");
99     putcur(13,5);
100    stout("Enter Starting Address in Hex (eg, 0000:0000): ");
101    ptr = inptr();

Copyright 1983,1988 INTEL CORPORATION
*spath = ,N,.C,.lst
*load example.86
*go til main
[ Break at :MENU#243 ]
*
Mod: MENU Proc: MENU_MEMDUMP Line: #91 STEP
```

Press these keys
in sequence:

To do this:

Alt-V C

Displays the current callstack. Note that, in this case, the program is not inside the menu_memdump procedure yet. Thus, the callstack is empty and the debugger returns a brief error message.

spacebar

Pressing any key continues.

sF1

Displays additional help on the last error message. In most cases, this command provides a more extensive help message than the message shown when the error occurs. In effect, this command is a way to ask for more help on the most recent error. It is the same as selecting the Last Error item on the Help Menu.

spacebar

Pressing any key continues.

F8

Steps again to line 92.

**Press these keys
in sequence:** **To do this:**

F10 Steps to line 96.
Alt-V C Displays the callstack again. This time, there
are several procedures listed on the stack.

The callstack screen looks like the following:

```
Debug Window Go Set View Browse Help
(0) :MENU#96 MENU_MENDUMP
(1) :MENU#272 MAIN
(2) :CALLMAIN#5

=== PRESS ANY KEY TO CONTINUE ===

* Mod: MENU Proc: MENU_MENDUMP Line: #96 PSTEP
```

**Press these keys
in sequence:** **To do this:**

spacebar Pressing any key continues.
F10 (2 times) Steps over subroutines that clear the screen and
position the cursor. There is no need to step
through these since they probably have nothing
to do with the bug.

**Press these keys
in sequence:**

To do this:

Grey + (3 times to
line 101)

Scrolls the thumbmark to line 101, past a few more lines of code that are not relevant to the bug. Line 101 contains an assignment to the variable ptr. The ptr variable contains the starting address that you specify for the memory dump subroutine. This line may be related to the bug that was discovered.

F7

Executes until the current line (line 101) is reached.

F4

Flips to the user-program screen to show that the menu screen has been cleared and now a prompt for a starting memory address is being displayed. You have not yet typed the start address value and the assignment to the variable ptr has not been done yet.

The memory dump screen looks like the following:

```
==== HEX/ASCII Memory Dump ====  
Enter Starting Address in Hex (eg, 0000:0000):
```

**Press these keys
in sequence:**

To do this:

F4

Flips back to the debugger screen.

Grey + (2 times to
line 103)

Scrolls the thumbmark to line 103, the next
source line.

Alt-G T

Uses the menu system to execute the user
program until the current line (line 103) is
reached (the same as the F7 key used above).

An alternate way to execute to line 103 involves
two steps. First, set a breakpoint at line 103
and, second, execute until the breakpoint is
reached. The Breakpoint item on the Set Menu
can be used to set the breakpoint at line 103,
and the Go Til Breakpt item on the Go Menu
can be used to execute until that breakpoint is
reached (the same as using the F9 key followed
by the F5 key).

At this point in the example, the user program is in control. The user
program displays the prompt for the starting address and waits for
you to enter the starting address for the hexadecimal memory dump.

**Press these keys
in sequence:**

To do this:

B0008000

Specifies the address of B000:8000 as the
starting address of the memory dump. The user
program then reaches line 103 and returns
control to the debugger.

At this point, the ptr variable should have been assigned the start
address that you specified (B000:8000). Type the following command
at the prompt in the Command Window to display the value of the
ptr variable:

```
*ptr  
0000H:8000H
```

Note that even though you specified B000 as the segment selector, the ptr variable contains 0000 for this value. So, the bug must occur at line 101 of the source code:

```
101 ptr = inptr();
```

This line assigns the ptr variable the value returned by the function inptr. Somewhere within the inptr function, the value that you specified is getting lost.

To find out where the inptr function is, type the following command at the prompt in the Command Window:

```
eval .inptr procedure  
:EXAMP10.INPTR
```

As shown, inptr is in the exampio module. To take a look at the source code, browse to this module. Instead of using the Ctrl-End navigation key, use the Scope item on the Browse Menu; it lets you browse to a known location.

**Press these keys
in sequence:**

To do this:

Alt-B S

Selects the Scope item on the Browse Menu. This item prompts for the name of a module and procedure.

:exampio.inptr
Enter

Browses to the specified module and procedure.

The screen display looks like the following:

```

Debug Window Go Set View Browse Help
315
316 /*=====
317     inptr - input a seg:off pointer from user and return it to caller.
318     =====
319 char *inptr()
320 {
321     char *ptr;
322     unsigned long dw;
323     unsigned int nib;
324
325     nib = inhnib();
326     dw = (long)(nib << 20);
327     nib = inhnib();
328     dw |= (long)(nib << 24);
329     nib = inhnib();
330     dw |= (long)(nib << 28);

*ptr
0000H:0000H
*eval .inptr procedure
:EXAMP10.INPTR
*
[Browsing] Mod: EXAMP10 Proc: INPTR Line: #320

```

You can scroll through the code from line 316 to line 346 to see what the `inpтр` subroutine is doing. First, it inputs a segment:offset address (8 characters long) a character at a time in the `nib` variable and accumulates these characters in the `dw` variable by shifting them left. Then, at the end, the `dw` variable is converted to a pointer and assigned to the `ptr` variable. This `ptr` value is then returned to the caller.

The next step in finding the bug is to start executing this subroutine.

Press these keys
in sequence:

To do this:

Grey + or Grey -
(to line 326)

Scrolls the thumbmark to line 326.

F7

Executes the memory dump program until line 326, starting where it left off before at line 103 in the menu module. At that point, you had typed in the starting memory address. Now, the memory dump program displays the memory at that location.

The user program is now in control and the screen display looks like the following:

```
==== Hexadecimal/ASCII Memory Dump ====
0000:0000 6C 05 EB 69 C6 EB C4 F8 FA C6 06 CF 02 00 C6 06  |---i-----|
0000:0010 D0 02 FF C6 06 05 03 00 0F 06 1A 05 0F 06 1C 05  |-----0--8,--n-|
0000:0020 0E 1E DE 02 0E 16 30 00 0B 26 2E 00 EB 6E BC 2E  |---XXX---P,-6---|
0000:0030 A3 1E 05 50 50 50 00 02 F2 50 2E FF 36 1C 05 2E  |-6---,----P-----|
0000:0040 FF 36 1A 05 2E A1 1E 05 CF 50 00 FE 0D 2D EE 0D  |:---S-,---Z-,--|
0000:0050 3B F0 00 C7 73 07 2E 0A 05 EE 0D 32 E4 2E A3 D2  |-XU-----x-^--8-~-|
0000:0060 02 50 56 BE 9E 0D E0 70 BD 5E C3 26 01 7E 0D F6  |-S-----t-----|
0000:0070 0F 73 0B 01 F0 F0 0F 74 04 01 F0 F0 0F C3 03 F0  |--8:^-u8---r--=u|
0000:0080 F0 C3 26 30 5E 0D 77 26 E0 0A 00 72 1E 00 3D 75  |--8-U-----r-----8-|
0000:0090 0E 26 0B 76 0D 01 FE F6 0F 72 0C 0B FF EB 0C D1  |F---P-6-----|
0000:00A0 EF D1 EF D1 EF D1 EF 01 E7 FF 0F 16 1F C3 26 C7  |---<--u--X--u--r-|
0000:00B0 0B 35 74 0C 51 01 04 D3 E2 59 01 E6 0F 00 EB 14  |-5t-Q-----Y-----|
0000:00C0 26 01 7E 0D F6 0F 73 0A 01 E6 00 F0 01 E2 FF 0F  |8-~-S-----|
0000:00D0 EB 02 33 F6 00 F2 09 35 36 C5 36 74 05 00 4C 05  |--3---56-6t--L-|

==== N - Next Screen: P - Previous Screen: ESCape:

```

**Press these keys
in sequence:**

To do this:

Esc	Escapes from the memory dump program and returns to the main menu of the EXAMPLE program. The memory dump program is completed.
3	Selects the memory dump program again from the menu, displaying the prompt for a starting address. To reach line 326 where the breakpoint was set so you can continue debugging, you must run the memory dump program again.
F	Specifies the first character of the segment offset. Then, the program reaches line 326 and breaks.

At the prompt in the Command Window, enter the following command to check the value of the nib variable:

```
*nib  
000F
```

The nib variable inputs the key you typed at the keyboard. As shown, nib is getting the correct value as you typed it.

**Press these keys
in sequence:**

To do this:

F8	Steps to the next line of the program where the value in the nib variable is shifted left and assigned to the dw variable.
----	--

At the prompt in the Command Window, enter the following command to show the value of the dw variable:

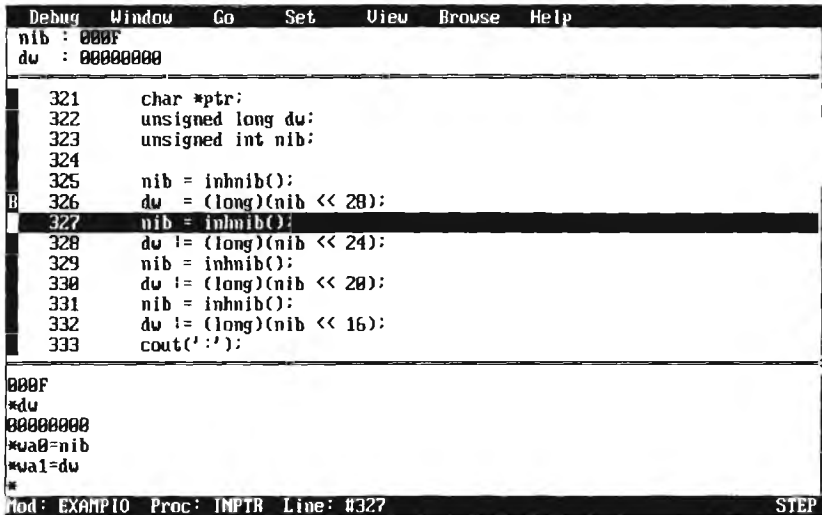
```
*dw  
00000000
```

This variable should be F0000000, so something happened during the assignment.

To observe this condition in further detail, set up a Watch Window for nib and dw by entering the following commands at the prompt in the Command Window:

```
*wa0=nib
*wa1=dw
```

A Watch Window is opened below the menu bar and the values of the two watch expressions are displayed.



The screenshot shows a debugger's Watch Window. At the top, there is a menu bar with the following items: Debug, Window, Go, Set, View, Browse, Help. Below the menu bar, the current values of the watch expressions are displayed: nib : 000F and dw : 00000000. The main area of the window shows a list of source code lines with their corresponding line numbers. The lines are: 321 char *ptr; 322 unsigned long dw; 323 unsigned int nib; 324 325 nib = inhNib(); 326 dw = (long)(nib << 28); 327 nib = inhNib(); 328 dw |= (long)(nib << 24); 329 nib = inhNib(); 330 dw |= (long)(nib << 20); 331 nib = inhNib(); 332 dw |= (long)(nib << 16); 333 cout('<'); The line numbers 327 and 328 are highlighted. At the bottom of the window, the current values of the watch expressions are displayed: 000F, *dw, 00000000, *wa0=nib, *wa1=dw, and *. At the very bottom, the debugger's status bar shows: Mod: EXAMP10 Proc: IMPTR Line: #327 STEP

**Press these keys
in sequence:**

To do this:

F10

Steps over the next line of the program and flips to the user-program screen, displaying the prompt again. The memory dump program is ready to accept the next character of the segment offset.

9

Specifies the next character of the segment offset. Notice in the Watch Window that the nib variable is correctly set to 9 but dw remains at 0.

F10 F10 8
F10 F10 7
F10 F10 F10 1
F10 F10 2
F10 F10 3
F10 F10 4

Steps through the program as you specify the remaining characters of the address. After each step, observe the values of nib and dw in the Watch Window. The nib variable is getting the correct value as you type it, but dw never accumulates these values until it gets to the lower word (i.e., the offset part of the address). The segment part of the address remains set to 0.

F10 (2 times)

Executes the next two steps until you reach line 345.

At the prompt in the Command Window, type the following command to check the value that was assigned to the ptr variable, confirming that it is incorrect:

```
*ptr  
0000H:1234H
```

Press these keys
in sequence:

To do this:

F5 Executes the memory dump program to display
Esc the memory specified; returns to the main menu
3 of the EXAMPLE program; selects the memory
F dump subroutine from the menu; and specifies
the first character of the segment offset.

You already discovered that the bug first occurs
in line 326. To look at this bug more closely
requires assembly-language debugging. The
commands you just typed execute the memory
dump program again until line 326 is reached.
The breakpoint is still set at this line from a
previous step.

F2 Turns on the Register Window and displays
F3 disassembled code in the View Window. Note
the current execution point is the first
assembly-language statement that makes up line
326.

The View Window now displays the disassembled code for each source
line:

Debug	Window	Go	Set	View	Browse	Help
nib : 000F						AX: 000F
dw : 00310020						BX: 1434
325 nib = inhNib();						CX: 0046
	79E1H:061FH	0E		PUSH CS		DX: 1D31
	79E1H:0620H	EB41FF		CALL \$-00BCH ; A=0		SI: 0000
	79E1H:0623H	8946F6		MOV [BP-0AH],AX		DI: 0000
	326 dw = (long)(nib << 28);					BP: 146C
	79E1H:0626H	8B46F6		MOV AX,[BP-0AH]		SP: 1462
	79E1H:0629H	B11C		MOV CL,1CH ; +28T		SS: 775A
	79E1H:062BH	D3E0		SHL AX,CL		DS: 1D31
	79E1H:062DH	BA0000		MOV DX,0		ES: 775A
	79E1H:0630H	8946FB		MOV [BP-08H],AX		CS: 79E1
	79E1H:0633H	8956FA		MOV [BP-06H],DX		IP: 0626
	327 nib = inhNib();					odIszpac
	79E1H:0636H	0E		PUSH CS		DB86
00000000						U1.0
*wa0=nib						
*wa1=dw						
*ptr						
0000H:1234H						
*						
mod: EXAMPIO Proc: INPTR Line: #326						BRK

**Press these keys
in sequence:**

To do this:

F8 (2 times)

Executes the next two assembly-language instructions. The line step command steps through the source lines as they are displayed in the View Window. If assembly-language instructions are shown, this command steps one instruction at a line. If high-level, source code is displayed, it steps one source statement at a time.

Notice the values of registers in the Register Window as you step. The character F that you typed is moved into the AX register and the shift counter of ICH (28T) is moved into CL.

F8

Executes the next assembly-language instruction, which is the instruction that shifts the value left 28 times. Notice, however, that the shift left is occurring on AX, a 16-bit register. If you shift a 16-bit value left 28 times, the value is shifted out of the register. And as observed, the AX register changes to 0.

F8 (3 times)

Executes the next two assembly-language instructions. Notice that the value of dw in the Watch Window changes to 0.

Re-examine the source line number 326:

```
dw = (long)(nib << 28);
```

This line is intended to convert nib, an unsigned int, into a long, a 32-bit unsigned variable. After conversion, it should shift the value left 28 times. What is actually happening is that nib is shifted left 28 times (and lost) and, after being shifted, it is converted to a 32-bit long and assigned to dw. The last three assembly-language instructions make this conversion and assignment.

To accomplish its intended purpose, the source line should read as follows:

```
dw = (long)nib << 28;
```

The parentheses around the nib << 28 expression cause it to be evaluated first before the conversion. Taking these parentheses out solves the problem and allows the conversion to take place first.

As you scroll through the View Window, note that this same error occurs in lines 328, 330, 332, 335, 337, and 339. The bug does not have any effect in lines 335, 337, and 339, the offset part of the address because, in these lines, the value is shifted left less than 16 times. In these cases, nib is long enough for the shift operation.

Now, that the bug has been found, enter the following command to exit from the debugger:

```
*exit
```

In the development applications process, illustrated in Figure 1-2, the next step is to edit the source files with a text editor and re-compile and re-link to create a corrected object file that you can continue to test and debug.

The remaining chapters in this manual provide reference information on the menu system, the keyboard controls, and the commands that can be entered in the Command Window.

Contents

Chapter 3 DB86 Menu System

3.1	Introduction	3-1
3.1.1	Starting the Menu System	3-2
3.1.2	Selecting Commands from the Menus	3-2
3.1.3	Exiting from the Menu System.....	3-3
3.2	The Debug Menu (Alt-D).....	3-4
3.3	The Window Menu (Alt-W).....	3-7
3.4	The Go Menu (Alt-G).....	3-11
3.5	The Set Menu (Alt-S)	3-15
3.6	The View Menu (Alt-V).....	3-17
3.7	The Browse Menu (Alt-B).....	3-19
3.8	The Help Menu (Alt-H)	3-23



3.1 Introduction

The debugger menu system is represented on the screen by the Menu Bar across the top line of the display. Once started, this menu system offers an easy way to do common debugging tasks. You do not have to remember command syntax or even command names. Just select the commands from the menus.

Generally, the menu system proves most useful when you are learning the system. Once you know how to use the system, you can turn the Menu Bar off and use function keys or commands most of the time.

This chapter describes the menu system. Each menu item is described separately.



Figure 3-1 Debugger Screen with Help Menu

In the following descriptions of the menus, cross-references to similar keyboard controls and encyclopedia entries are listed after the menu item description. Keyboard controls are described in Chapter 4 while encyclopedia entries are in Chapter 5.

3.1.1 Starting the Menu System

There are two ways to activate the menu system:

Alt-M activates the menu system. The entire Menu Bar is shown in the highlight color on color monitors. The first pulldown menu name (Debug) is shown in reverse video on all monitors.

To select a pulldown menu, use the leftarrow (←) and rightarrow (→) keys to highlight the desired menu in the Menu Bar. Press enter to select the highlighted pulldown menu. The selected menu is pulled down on the screen under the menu name to show the items available on that menu.

Alt-x activates the menu system and directly selects the specified menu. *x* is the first letter of the name of a pulldown menu. The leftarrow (←) and rightarrow (→) keys can then be used to cycle through the neighboring menus after one has been selected.

3.1.2 Selecting Commands from the Menus

There are two ways to select an item from a pulldown menu:

- Use the uparrow (↑) and downarrow (↓) keys to highlight the item desired on the pulldown menu and press the enter key to select the highlighted item. You can also use the Home and End keys to jump to the first and last menu items, respectively.
- Type the first character of the name of the item as it appears on the pulldown menu.

Any menu item followed by ellipses (...) prompts for additional information. Enter the information requested in the pop-up window to complete the function. A standard set of line-editing controls can be used when entering information at the prompt. See Chapter 4 for more information.

3.1.3 Exiting from the Menu System

When the menu system is active, press the ESC key to exit from it.

If you are entering text at a prompt after you have selected a menu item, press ESC once to clear the prompt and then press ESC again to exit from the menu system.

If you have already selected a menu item, the debugger automatically exits from the menu system when the function is complete. The exception is the Help Menu; the debugger returns to the Help Menu after displaying the requested help. To exit from the menu system when you are in the Help Menu, press ESC or explicitly select the Quit Help menu item.

3.2 The Debug Menu (Alt-D)

The following screen shows the Debug Menu:



The Debug Menu contains utility commands for initializing the debug environment, running DOS commands, and exiting from the debugger.

Load Program ... prompts for the object file to be loaded for debugging. Drive and pathnames can be specified with the filename. For example, you can specify a file as follows:

```
C:\PROGS\MYPROG.86
```

After the program is loaded, the debugger searches for the corresponding source file in the drive and directory specified by the current Source Path. If the source file is found, it is displayed in the View Window. Otherwise, assembly language is disassembled from the memory where the program was loaded.

Keyboard Control: none
Encyclopedia Entry: LOAD

Reload clears all defined breakpoints, tracepoints, and watch expressions and reloads the object file that was most recently loaded.

Keyboard Control: none
Encyclopedia Entry: LOAD

Source Path ... prompts for the pathname of the directory where the debugger can find source files for display. Optionally, you can specify up to 10 extensions separated by commas. For example, if the source files have an extension of .LST or .C and are in the directory PROG on drive C, you can specify the following Source Path:

C:\PROG,.C,.LST

The debugger searches the path for source files with the same extensions as those specified in the Source Path, and in the order they are listed. The names of the source files (less extensions) are assumed to be the same as the debug module name (i.e., those names seen by the the DIR module command). See Appendix C for further information. If you specify a path with no extensions, the previously defined extensions are used. If no extensions have been defined, the default, .LST, is used. New extensions replace previously defined extensions.

Keyboard Control: none
Encyclopedia Entry: SPATH

DOS Shell

temporarily exits from the debugger, displays the DOS prompt, and lets you enter DOS commands. For example, you can edit changes in a source file with a text editor. The display screen flips to the user-program screen (page 0 on CGA) which is used for output from DOS commands. Enter EXIT at the DOS prompt to return to the debugger. The complete status of the debug session, including the screen display, is restored.

Keyboard Control: none

Encyclopedia Entry: !

Exit DB86

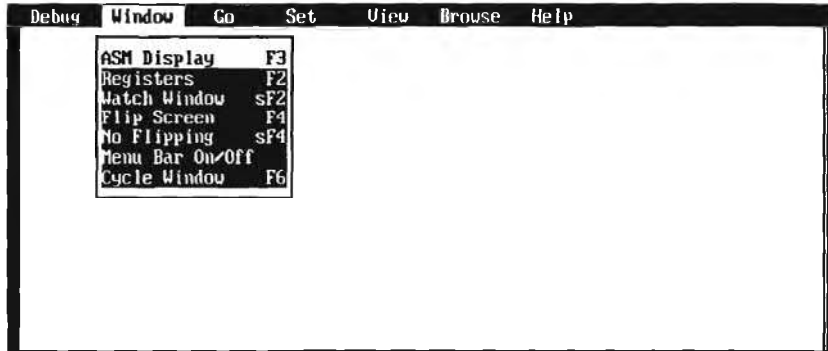
exits from the debugger and returns to DOS.

Keyboard Control: none

Encyclopedia Entry: EXIT

3.3 The Window Menu (Alt-W)

The following screen shows the Window Menu:



The Window Menu contains commands that control the debugger screen display and windows.

ASM Display toggles the View Window between program source display (the default) or source display plus the assembly language for each source line (i.e., disassembly display).

Keyboard Control: F3

Encyclopedia Entry: ASM

Registers toggles the display of the processor registers and flags in the Register Window that appears on the right side of the display screen. The Register Window stays on until it is toggled off. All register contents are shown in hexadecimal. The flags are shown as an eight-character value (e.g., ODisZAPc) where each letter stands for one of the processor flags as follows:

O - Overflow	Z - Zero
D - Direction	A - Auxiliary Carry
I - Interrupt Enable	P - Parity
S - Sign	C - Carry

If the letter is uppercase, the corresponding flag is set (logic 1); if the letter is lowercase, the corresponding flag is clear (logic 0).

The Register Window is updated when the user program stops (e.g., at breakpoints or during stepping).

Keyboard Control: F2

Encyclopedia Entry: FLAGS, REGS, REGS87

Watch Window

toggles the display of the Watch Window appearing between the Menu Bar and the View Window. The Watch Window stays on until toggled off. The current values of watch expressions are displayed here. Watch expressions are defined with the WA command and can hold the values of user-program variables. The Watch Window is updated when the user program stops (e.g., at breakpoints or during stepping).

Keyboard Control: sF2

Encyclopedia Entry: WA

Flip Screen

toggles to show the output screen from the user program. The entire debugger screen is temporarily replaced with the output screen of the user program that is currently being debugged. Press any key to restore the debugger screen. If screen flipping has been toggled off with the No Flipping menu item, Flip Screen turns it back on.

Keyboard Control: F4

Encyclopedia Entry: none

No Flipping

toggles screen flipping off until a flip is explicitly requested again with the F4 key. Whenever the user program begins executing, the debugger saves its display screen and allows the user program to control the display screen. When the user program stops, the debugger restores its display screen saving the user-program, display screen. This screen flipping can cause the display screen to blink annoyingly (e.g., when you are in Auto Step mode). This switching takes place even when the user program does not output to the display screen. In this case, the flipping can be turned off safely with the No Flipping option. Do not use this function if the program being debugged outputs to the display screen. In this case, the user program and the debugger will conflict with one another overwriting the display screen.

Keyboard Control: sF4

Encyclopedia Entry: none

Menu Bar On/Off

toggles the display of the Menu Bar at the top of the screen. With the Menu Bar off, an extra line of display is used for the View Window. When off, you can still access the menu system in the same way as if the Menu Bar were on. The Menu Bar stays off until toggled back on, except that it is temporarily shown whenever you access the menu system.

Keyboard Control: none

Encyclopedia Entry: none

Invocation Control: NOMB

Cycle Window

toggles debugger control between the View Window and the Command Window. The system cursor blinks in the window currently in control. Additionally, color monitors show the window currently in control with the highlight color. When in the Command Window, you can enter debugger command lines. When in the View Window, you can use the keyboard controls to view source code and to do many common debugging tasks.

Keyboard Control: F6

Encyclopedia Entry: none

3.4 The Go Menu (Alt-G)

The following screen shows the Go Menu:



The Go Menu contains commands that execute the user program under debugger control.

With the Auto Step, Step, and Procedure Step functions in the Go Menu, the user program can step by source-level statement or by assembly-language statement, determined by the display in the View Window. If source only is shown in the View Window, stepping is by numbered, executable, source line. If assembly language is shown in the View Window, then stepping is by assembly-language instruction. Assembly language can be shown in the View Window when the source is not available or when the disassembly display is toggled on with the ASM Display item in the Window Menu.

With the Call Step item, stepping is to the next CALL instruction whether or not the CALL aligns with a source-level statement. Likewise, with the Return Step item, stepping is to the next RETURN instruction.

Til Cursor Line	<p>clears previous temporary breakpoints and sets a new temporary breakpoint at the current cursor line of the user program shown in the View Window. Then this menu item executes the user program from the current execution point until an enabled fixed or conditional breakpoint is reached or until the temporary breakpoint is reached. The temporary breakpoint remains active until the next Go Til Cursor Line command or until breakpoints are cleared. The debugger beeps if you are not on an executable source line when you issue this command.</p> <p>Keyboard Control: F7 Encyclopedia Entry: GO TIL</p>
Go Til Breakpoint	<p>executes the user program from the current execution point, keeping all currently defined breakpoints and tracepoints.</p> <p>Keyboard Control: F5 Encyclopedia Entry: GO</p>
Keep Fixed	<p>executes the user program from the current execution point, clearing temporary breakpoints, but keeping all fixed breakpoints, conditional breakpoints, and tracepoints.</p> <p>Keyboard Control: sF7 Encyclopedia Entry: GO</p>
Forever	<p>clears all breakpoints and tracepoints and executes the user program from the current execution point. The only way to stop the program before it reaches completion is with the SysReq or Ctrl-Break keys.</p> <p>Keyboard Control: sF5 Encyclopedia Entry: GO FOREVER</p>

Auto Step

steps through the user program starting at the current execution point. Auto stepping continues until a fixed, conditional, or temporary breakpoint is reached or until any keyboard key is pressed. Conditional breakpoints and watch expressions are updated at every step.

Keyboard Control: none

Encyclopedia Entry: LSTEP, ISTEP

Step

executes one step of the user program at the current execution point, retaining all breakpoints.

Keyboard Control: F8

Encyclopedia Entry: LSTEP, ISTEP

Call Step

executes the user program from the current execution point until one step beyond the next encountered CALL instruction. In other words, Call Step steps into the called procedure or function, easing the task of nesting into the next procedure or function that will be called.

Keyboard Control: sF8

Encyclopedia Entry: CSTEP

Procedure Step

steps through a called procedure. If you are currently on a CALL instruction this menu item executes the called procedure or function and stops upon return from it. If you are not on a CALL instruction, this menu item is the same as Step.

Keyboard Control: F10

Encyclopedia Entry: PSTEP

Return Step

executes the user program from the current execution point until one step beyond the next RETURN instruction (i.e., execution stops after the next RETURN instruction that is encountered). Thus, Return Step nests out of the called procedure or function to the calling procedure or function. If an intervening CALL instruction is encountered, it is stepped over.

Keyboard Control: sF10

Encyclopedia Entry: RSTEP

3.5 The Set Menu (Alt-S)

The following screen shows the Set Menu:



The Set Menu contains commands that set and clear breakpoints.

Breakpoint sets a temporary breakpoint at the program location indicated by the current cursor in the View Window. A maximum of 10 temporary breakpoints can be set at the same time. Temporary breakpoints are shown in the View Window by highlighting the source display line and displaying the letter B next to the source display line in the Break Status Column at the left side of the View Window.

Keyboard Control: F9

Encyclopedia Entry: GO TIL

Tracepoint sets a tracepoint at the program location indicated by the current cursor in the View Window. When execution reaches a tracepoint, an announcement message is displayed in the Command Window and the Watch Window is updated. A maximum of ten tracepoints can be set at the same time.

Keyboard Control: sF9

Encyclopedia Entry: TRn

Clear at Cursor

clears any breakpoint or tracepoint at the current cursor location in the View Window.

Keyboard Control: none

Encyclopedia Entry: none

Remove All

clears all breakpoints and tracepoints that are currently defined. Watch expressions are retained.

Keyboard Control: none

Encyclopedia Entry: none

3.6 The View Menu (Alt-V)

The following screen shows the View Menu:



The View Menu contains commands that display information in a pop-up window that overwrites the debugger screen. If the information exceeds the size of this window, a pause prompt appears on the Status Line, and you can press any key to continue the display. At the end of the display, press any key to restore the debugger screen.

Calls displays the call ancestry relative to the current execution point in the user program. The call ancestry is the nested list of procedures that have called one another down to the currently executing procedure.

Keyboard Control: none

Encyclopedia Entry: CALLS

Expanded Calls	<p>displays the call ancestry relative to the current execution point in the user program. Additionally, this menu item displays the local variables allocated on the stack for each procedure.</p> <p>Keyboard Control: none Encyclopedia Entry: CALLS EXP</p>
Locals	<p>displays the current value of variables defined local to the current execution point in the user program.</p> <p>Keyboard Control: none Encyclopedia Entry: LOCALS</p>
Modules	<p>displays the names of all modules in the program currently loaded for debugging.</p> <p>Keyboard Control: none Encyclopedia Entry: DIR MODULE</p>
Debug Status	<p>displays the status of all breakpoints, tracepoints, and watch expressions that are currently defined. Any user-defined debug objects and their respective values are also shown.</p> <p>Keyboard Control: none Encyclopedia Entry: DIR DEBUG</p>

3.7 The Browse Menu (Alt-B)

The following screen shows the Browse Menu:



The Browse Menu contains commands that change the scope to another module or return to the home scope (i.e., the current execution point).

Home Scope restores the View Window and the current scope to the current execution point in the user program. After browsing to other modules or scrolling through the current module, this function returns you to the home scope. The current execution point (the location in the user program to which the current instruction pointer is pointing) is highlighted with the reverse video execution bar in the View Window.

Keyboard Control: Grey *

Encyclopedia Entry: none

Scope ...

prompts for a scope expression in a pop-up window and changes the scope context to the specified location. The expression entered at the prompt should evaluate to a location in the user program being debugged. For example, the following symbolic reference evaluates to line number 40 in the DEMO module:

```
:DEMO#40
```

Changing the scope explicitly using this menu function is another way to browse to a different source module. At any point when the user program stops execution, the scope is set to the same location as CS:IP. This location, the current execution point in the user program, is also known as the home scope. When you browse to another module, you can scroll through its source code in the View Window; view any variables that are local to the new scope; or set breakpoints at the new location.

Keyboard Control: none

Encyclopedia Entry: SCOPE

Calls

initiates callstack browsing. This menu item allows you to browse through modules using the Ctrl-Home and Ctrl-End keys according to the call ancestry relative to the current execution point. The Ctrl-Home key browses toward the top of the call stack and Ctrl-End browses toward the end of the call stack (toward the most distant calling ancestor). Normally, Ctrl-Home and Ctrl-End do not browse through modules according to the call ancestry, but instead browse through all modules in the user program. The next go or step command that is issued restores normal module browsing. Local variables become active as you browse to each level and may be viewed by the Locals item in the View Menu.

Keyboard Control: none

Encyclopedia Entry: none

Find ...

prompts for a search string and searches the source code displayed in the View Window starting at the current cursor location. Only the source for the current module is searched. If a match is found, the surrounding source is displayed in the View Window and the thumbmark is placed on the line where the match occurred. If no match is found, the debugger beeps and displays a message on the Status Line.

Keyboard Control: none

Encyclopedia Entry: none

Next Find

searches forward from the current cursor location for the next occurrence of the search string that was specified for the most recent Find.

Keyboard Control: none
Encyclopedia Entry: none

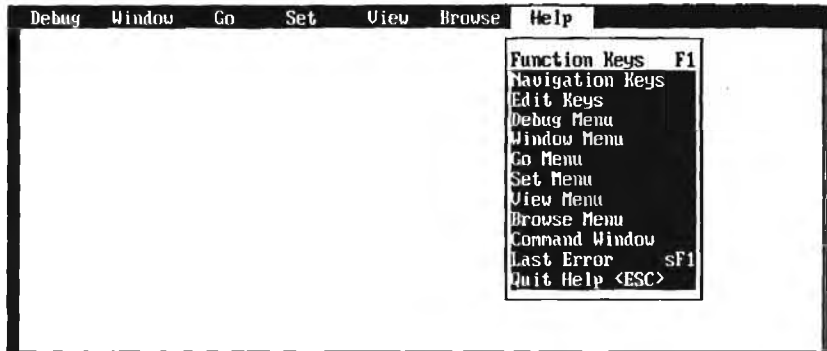
Previous Find

searches backward from the current cursor location to find the previous occurrence of the search string that was specified for the most recent Find.

Keyboard Control: none
Encyclopedia Entry: none

3.8 The Help Menu (Alt-H)

The following screen shows the Help Menu:



Help Menu Items

Each item on the Help Menu displays a help screen on the related subject. The help screen overwrites the current debugger screen. Press any key to restore the debugger screen and return to the Help Menu. As a convenience, you are returned to the help menu after viewing a help topic. The Quit Help item is the same as pressing the Esc key and exits the menu system.

Keyboard Control: F1, sF1

Encyclopedia Entry: HELP



Contents

Chapter 4 DB86 Keyboard Controls

4.1	Introduction	4-1
4.2	Navigational Key Controls.....	4-1
4.3	Line-editing Key Controls	4-3
4.4	Function Key Controls.....	4-5
4.5	Other Keyboard Controls.....	4-8



4.1 Introduction

This chapter describes the single key controls available in the debugger: navigational keys, line-editing keys, function keys, and other control keys. These single key controls provide a responsive level of interacting with the debugger and play an important role in making the debugger easy to use.

Keyboard templates that can be placed on your keyboard are supplied with the debugger and contain reference information on debugger keyboard controls.

4.2 Navigational Key Controls

The navigational key controls work only when the View Window is active unless otherwise noted.

↑	moves the cursor up one line.
↓	moves the cursor down one line.
←	scrolls the View Window left by one column.
→	scrolls the View Window right by one column.
Ctrl-←	scrolls the View Window left by ten columns.
Ctrl-→	scrolls the View Window right by ten columns.
PgUp	scrolls the View Window up one page. A page is equal to the current size of the View Window with a one line overlap. PgUp works either in the View Window or the Command Window.
PgDn	scrolls the View Window down one page. A page is equal to the current size of the View Window with a one line overlap. PgDn works either in the View Window or the Command Window.

- Grey - scrolls the View Window up one line to move back to the beginning of the source code. The cursor stays at the current window location and does not follow the data being viewed. Grey - works either in the View Window or the Command Window.
- Grey + scrolls the View Window down one line to move forward to the end of the source code. The cursor stays at the current window location and does not follow the data being viewed. Grey + works either in the View Window or the Command Window.
- Grey * restores the scope to the current execution point in the user program. After browsing to different modules or scrolling through the current module, use the Grey * to quickly return to the home scope. Grey * works either in the View Window or the Command Window. Grey * is the same as the Home Scope item on the Browse Menu.
- Home jumps to the start of the current source module.
- End jumps to the end of the current source module.
- Ctrl-Home browses to the previous module. Usually, the window is cued to the start of the previous module in the module list. However, if the Calls item on the Browse Menu has been set, Ctrl-Home browses to the next procedure toward the top of the callstack. Ctrl-Home works either in the View Window or the Command Window.
- Ctrl-End browses to the next module. Usually, the window is cued to the start of the previous module in the module list. However, if the Calls item on the Browse Menu has been set, Ctrl-End browses to the next procedure toward the end of the callstack, i.e., toward the most distant calling ancestor. Ctrl-End works either in the View Window or the Command Window.

4.3 Line-editing Key Controls

Both command-line input and user responses to prompts can be edited using the same set of line-editing keys.

In addition to editing the current command line, the debugger maintains a command-line buffer so previously issued commands can also be recalled, edited, and re-issued.

Command-line editing can be done only when the Command Window is active.

Editing of user responses to prompts can be done until the Enter key is pressed.

The following keys can be used only when editing command lines in the Command Window.

- | | |
|--------|---|
| Ctrl-E | re-issues the last command stored in the command-line history buffer. |
| ↑ | clears the current command line and restores the previously issued command line from the history buffer. You can edit and re-issue this command line. Use the uparrow (↑) key to cycle backward to the beginning of the command-line history buffer. At the beginning of the history buffer, the uparrow (↑) key causes a beep. |
| ↓ | clears the current command line and restores the next command in the history buffer. You can edit and re-issue this command line. Use the downarrow (↓) key to cycle forward to the end of the command-line history buffer. At the end of the history buffer, the downarrow (↓) key causes a beep. |

The following keys can be used in both cases to edit command lines or user responses.

←	moves the cursor left by one character.
→	moves the cursor right by one character.
Home	restores the cursor to the left end of the current line.
End	restores the cursor to the end of the current line.
Ins	toggles between insert and overstrike editing modes.
Del	deletes the character under the screen cursor.
Ctrl-F	deletes the character under the screen cursor (same as Del).
Grey ←	backspaces and deletes the character to the immediate left of the current screen cursor.
Ctrl-X	deletes all characters from the current screen cursor to the beginning of the line.
Ctrl-A	deletes all characters from the current screen cursor to the end of the line.
Ctrl-Z	deletes the entire current line (same as ESC).
Esc	deletes the entire current line (same as Ctrl-Z).
Enter	submits the currently edited line for execution.

4.4 Function Key Controls

The function keys work when the Command Window is active or when the View Window is active.

Every function key duplicates a menu item. The following descriptions summarize the function key and provide a cross-reference to the menu item which describes the function in more detail. Cross-references to related command encyclopedia entries are also listed.

F1 provides help on function keys and other keyboard controls.

Menu/Item: Help/Function Keys
Encyclopedia Entry: HELP

F2 toggles the Register Window on or off.

Menu/Item: Window/Registers
Encyclopedia Entry: FLAGS, REGS, REGS87

F3 toggles the View Window between program source display and disassembly display.

Menu/Item: Window/ASM Display
Encyclopedia Entry: ASM

F4 toggles to show the debugged program's output screen.

Menu/Item: Window/Flip Screen
Encyclopedia Entry: none

F5 executes the user program from the current execution point, keeping all currently defined breakpoints and tracepoints.

Menu/Item: Go/Til Breakpoint
Encyclopedia Entry: GO

- F6 toggles debugger control between the View Window and the Command Window.
Menu/Item: Window/Cycle Window
Encyclopedia Entry: none
- F7 clears previous temporary breakpoints; sets a new temporary breakpoint at the current line; and executes the user program from the current execution point.
Menu/Item: Go/Til Cursor Line
Encyclopedia Entry: GO TIL
- F8 executes one step of the user program.
Menu/Item: Go/Step
Encyclopedia Entry: LSTEP, ISTEP
- F9 toggles a temporary breakpoint on and off at the program location indicated by the current cursor in the View Window.
Menu/Item: Set/Breakpoint
Encyclopedia Entry: none
- F10 if on a CALL instruction, executes the called procedure or function and stops upon return from it. Otherwise, executes one step of the user's program (same as F8).
Menu/Item: Go/Procedure Step
Encyclopedia Entry: PSTEP
- sF1 provides additional help on the most recent error message. In effect, this function is a way to get more help.
Menu/Item: Help/Last Error
Encyclopedia Entry: HELP *En*

- sF2 toggles the on-screen presence of the Watch Window.
Menu/Item: Window/Watch Window
Encyclopedia Entry: WA
- sF4 toggles screen flipping off or on.
Menu/Item: Window/No Flipping
Encyclopedia Entry: none
- sF5 clears all breakpoints and tracepoints and executes the user program from the current execution point.
Menu/Item: Go/Forever
Encyclopedia Entry: GO FOREVER
- sF7 clears temporary breakpoints and executes the user program keeping fixed breakpoints, conditional breakpoints, and tracepoints.
Menu/Item: Go/Keep Fixed
Encyclopedia Entry: GO
- sF8 executes the user program stepping into the next called procedure or function that is encountered.
Menu/Item: Go/Call Step
Encyclopedia Entry: CSTEP
- sF9 sets a tracepoint at the program location indicated by the current cursor in the View Window.
Menu/Item: Set/Tracepoint
Encyclopedia Entry: TR n
- sF10 executes the user program until one step beyond the next RETURN instruction encountered, thus nesting out of the called procedure or function.
Menu/Item: Go/Return Step
Encyclopedia Entry: RSTEP

4.5 Other Keyboard Controls

In addition to the function keys, there are several other keys that perform debugging functions.

The Ctrl-Home and Ctrl-End keys allow browsing from module to module. The Grey * key restores the home scope. The Grey - and Grey + keys move the View Window over the source code, scrolling through the source. All of these keys are described in detail in Section 4.2.

The following keys are also available to control the debugger:

- Ctrl-PgUp moves the window divider up, making the View Window smaller by one line and the Command Window larger by one line. Ctrl-PgUp works either in the View Window or the Command Window.
- Ctrl-PgDn moves the window divider down, making the View Window larger by one line and the Command Window smaller by one line. Ctrl-PgDn works either in the View Window or the Command Window.
- SysReq restores control to the debugger when the user program is executing. For example, if the GO FOREVER command is issued and you want to stop execution before the program ends or if the program is caught in an infinite loop, you can force control back to the debugger with the SysReq key.
- Ctrl-Break same as SysReq when the user program is in control; otherwise, when the debugger is in control, same as Ctrl-C.
- Ctrl-C cancels the current command.

If the debugger is displaying more information than will fit in the window being used, the following keys are available to control the display.

- F(ast mode) continues output without pausing after each screen.
- L(ine mode) displays one line of output and pauses again.
- P(age mode) displays one window of output and then pauses again.

Chapter 5 DB86 Command Language Encyclopedia

5.1	How to Use this Chapter.....	5-1
5.2	Functional Overview of Commands.....	5-3
	! Shell Escape.....	5-10
	ASM.....	5-11
	BASE.....	5-15
	Breakpoints.....	5-17
	CALLS.....	5-19
	CB.....	5-22
	Command Entry.....	5-24
	Control Constructs.....	5-28
	COUNT.....	5-30
	CSTEP.....	5-32
	Data types.....	5-33
	Debug Variables.....	5-35
	Debugger Commands.....	5-36
	DEFINE.....	5-37
	DIR.....	5-39
	DO.....	5-44
	ENABLE/DISABLE.....	5-46
	EVAL.....	5-49
	EXIT.....	5-52
	Expressions.....	5-53
	FB.....	5-62
	FLAGS.....	5-64
	GO.....	5-66
	HELP.....	5-68
	IF.....	5-71
	INCLUDE.....	5-73
	ISTEP.....	5-74
	Lexical Elements.....	5-76
	LIST/NOLIST.....	5-79
	LOAD.....	5-80
	LSTEP.....	5-82
	Memory Access.....	5-84
	Menu System.....	5-87
	OBREAK.....	5-88
	PORT.....	5-89
	Pseudovariables.....	5-91
	PSTEP.....	5-92

Registers and Flags.....	5-94
REGS.....	5-95
REGS87.....	5-99
REPEAT	5-102
RSTEP.....	5-104
SCOPE.....	5-106
SETMOD.....	5-108
SPATH.....	5-110
STACK.....	5-112
Stepping Commands.....	5-113
SYSINT.....	5-114
TR	5-116
User-program Symbols.....	5-118
WA.....	5-119
WPORT	5-121

Chapter 5

DB86 Command Language Encyclopedia

This chapter contains a brief functional overview of debugger commands and provides comprehensive reference descriptions for the commands and related topics in alphabetical order.

5.1 How to Use this Chapter

The command entry shown on the following page is an example of an entry in this encyclopedia. The entry includes command syntax, with definitions for keywords, delimiters, and variables; discussion of how the command is used; examples; and cross-references to related commands and to menu system commands and keyboard controls that perform a similar function. Topical entries may include additional subject headings, as appropriate.

```
Debug Window Go Set View Browse Help

#help
#
HELP is available for:

ASM          BASE          BREAKPOINT   BROWSEMENU   CALLS        CMDWINDOW
COMMANDS     COMMENTS     CONSTRUCTS   COUNT        CSSTEP      DEBUGMENU
DEFINE       DEREFERENCE   DIR          DISPLAY       DO           EDIT
ENTRY        EXCEPTIONS    EXIT         GO           GOMENU      HELP
IF           INCLUDE       INVOCATION   ISTEP        KEYBOARD    LABELS
LINES       LIST          LOAD         LSTEP        MENU        MODIFY
MODULES      OBJECTS       OPERATORS    PORT         PROCEDURES  PSTEP
REALS       REGISTERS     REPEAT      RSTEP        SCOPE       SETMENU
SOURCE      STACK         TYPES        VARIABLES    VIEWMENU    WINDOWMENU

<No Program Loaded>
```

Figure 5-1 DB86 Command Language Help Screen

COMMAND NAME

A short statement about the function the command performs

Syntax

The syntax section gives the correct syntax for the command, including all modifiers and variables that are available. The notational conventions used in command syntax are listed on the inside front cover of this book.

Discussion

The discussion section defines the command and explains how it is used.

Examples

The examples section gives one or more examples of the command, showing how it might be used in a debugging environment. The debugger prompt precedes user input, just as it would if you were using the system. User input appears in bold type and system output appears in regular type. Debugger keywords are in all caps. An example may include comments enclosed in slash-asterisk delimiters (e.g., /* comment */).

The following example shows how user input is differentiated from system output. In this example, **BASE** is user input; **HEX** is debugger output.

```
*BASE  
HEX
```

Most command and topic entries include examples.

Cross-references

The cross-references section refers you to related entries in the command language encyclopedia and to any menu-system items or keyboard controls that perform a similar function. Most command and topic entries include cross-references.

Encyclopedia Entries

The commands section lists other commands or topics in this command language encyclopedia.

Menu/Item

The menu/item section first lists the menu in which you will find the item, and then the item itself. The item performs a function similar to the command that is the subject of this entry.

Keyboard Control

The keyboard control section lists the keyboard control that performs a function similar to the command that is the subject of this entry.

5.2 Functional Overview of Commands

Tables 5-1 through 5-7 group debugger commands according to function and give a brief definition of each command and topic covered in the Command Language Encyclopedia.

Table 5-1 DB86 Control Constructs

Entry	Description
COUNT	Groups and executes commands a specified number of times
DO	Groups and executes commands
IF	Groups and conditionally executes commands
REPEAT	Groups and executes commands forever, while, or until an exit condition is met

Table 5-2 DB86 Debug Environment Commands

Entry	Description
! Shell Escape	Accesses DOS system commands from the debugger environment
BASE	Displays or establishes the default base for numeric constants
DEFINE	Defines a global or local debug variable
DIR	Displays a listing of symbolic debug information
EVAL	Calculates and displays the results of an expression
EXIT	Terminates the debug session and returns control to the host operating system

Table 5-2 DB86 Debug Environment Commands (continued)

Entry	Description
HELP	Provides on-line operating assistance
INCLUDE	Executes command input from a text file
LIST/NOLIST	Opens or closes a debug session log file
OBREAK	Is a pseudovisible that determines whether or not a breakpoint occurs when a user-program overlay is loaded
SYSINT	Is a pseudovisible that determines whether or not breakpoint checking occurs when a DOS interrupt is encountered during program execution

Table 5-3 DB86 Processor Status Commands

Entry	Description
FLAGS	Displays or changes 8086/8088 flags
PORT	Is a pseudovisible that displays or changes the contents of a specified byte wide I/O port
REGS	Displays or sets 8086/8088 microprocessor registers
REGS87	Displays or sets 8087 emulator or numerics processor registers
WPORT	Is a pseudovisible that displays or changes the contents of word-wide ports

Table 5-4 DB86 Source Display Commands

Entry	Description
SCOPE	Sets a new debugger browse context or displays the current one
SETMOD	Correlates an executable module with a source display file
SPATH	Displays or sets the path and default extensions used to find the source display files of compiled modules

Table 5-5 DB86 Memory Access Commands

Entry	Description
ASM	Displays 8086 and 8087 machine code using assembler mnemonics or patches mnemonics line by line to memory
CALLS	Displays procedure calls on the stack
LOAD	Reads an Intel-86, load-time locatable, OMF file into memory
STACK	Displays values from the top of the stack

Table 5-6 DB86 Execution and Watch Commands

Entry	Description
CB	Sets conditional breakpoint or displays the current conditional breakpoints
CSTEP	Steps into the next procedure call encountered
ENABLE/DISABLE	Enables or disables a specified breakpoint, tracepoint, or watch expression; or enables or disables all breakpoints, tracepoints, or watch expressions
FB	Sets fixed breakpoint or displays current fixed breakpoints
GO	Starts program execution and controls breakpoints and tracepoints
ISTEP	Steps through a program one machine instruction at a time
LSTEP	Steps to next high-level language line in program
PSTEP	Steps to the next high-level language line in the program, stepping over procedure calls
RSTEP	Steps out of the current procedure until after the next return encountered, stepping over calls
TR	Sets new tracepoint or displays current tracepoints
WA	Sets a new watch expression or displays a current watch expression

Table 5-7 DB86 Topical Entries

Entry	Description
Breakpoints	Set breakpoints, tracepoints, or watch expressions; or display the current breakpoint, tracepoint, or watch expression
Command Entry	Includes command-line interface, continuation lines, multiple command lines, comments, editing, and display controls
Control Constructs	Group commands and control program execution
Data Types	Are the standard types, used in commands and displays, with which the debugger is familiar
Debug Variables	Store temporary values during a debug session
Debugger Commands	Are the primary commands in the debugger command-language
Expressions	Are one or more numbers, debug variables, or functions separated by operators
Lexical Elements	Are names, numbers, strings and special character delimiters recognized by the debugger
Memory Access	Makes it possible to display or change debug variables or program variables in user-program memory
Menu System	Gives access, through pulldown menus, to many debugger commands

Table 5-7 DB86 Topical Entries (continued)

Entry	Description
Pseudovariables	Affect the operation of the debugger
Registers and Flags	Set the contents of a specified register or flag; display the contents of all registers or flags or a specified register or flag
Stepping Commands	Step program execution by called procedure, machine instruction, language instruction, procedure, or return instruction
User-program Symbols	Are accessible for display

! Shell Escape

Accesses DOS system commands
from the debugger environment

Syntax

```
!{DOS_command}
```

Where:

DOS_command is a DOS command line that you wish to execute.

Discussion

The ! shell escape command provides access to DOS system commands from the debugger environment, but does not save the current DB86 screen. You can run any program or command executable from the DOS command level, as long as there is enough RAM.

Example

In this example, the ! command is used to execute the DOS SET command, which displays DOS environment variables that are currently set:

```
*!SET  
COMSPEC = C:\COMMAND.COM  
:WORK: = D:\  
PROMPT = $P$G
```

Cross-references

Encyclopedia Entry

EXIT

Menu/Item

Debug/DOS Shell

ASM

Displays 8086 and 8087 machine code using assembler mnemonics or patches mnemonics line by line to memory

Syntax

```
ASM aexpr [= 'asm-86'[, 'asm-86', ...]]
```

Where:

aexpr is an address, an expression that evaluates to an address, or a range of addresses specified as *addr* TO *addr* or *addr* LENGTH *n*, where *addr* is an address or an expression that evaluates to an address and *n* is the specified number of instructions.

asm-86 is any valid 8086 instruction.

Discussion

The ASM command displays a single instruction or specified set of instructions in the user program in assembler mnemonics. It also enables you to write assembly-code patches to memory.

Disassembly

The disassembly format depends on whether you use ASM with a single address, a range of addresses or a partition. To display the first instruction at a given address, use the form ASM *addr*. To display all instructions that start within a certain range, use the form ASM *addr* TO *addr*; an instruction is displayed if its first byte is within the range, even if subsequent bytes are not. To display an exact number of instructions, use the form ASM *addr* LENGTH *n*.

When you use an absolute address to specify a partition, the absolute address is converted into *segment:offset* form before disassembly begins.

ASM (continued)

Disassembled instructions are displayed in columns; from left to right, they are address, hexadecimal, object-code values, opcode mnemonics, and operands (if any). For example, here is line 266 from the MENU module in a sample user program called EXAMPLE.86:

```
*ASM #266
4FA7H:0415H E94F00      JMP  :MENU #281
```

Because the debugger can determine the length of lines, procedures, and modules within the user program, you can limit the display of disassembled instructions to those of interest. For example, to see only the instructions in line number 37 of your program, enter the command `ASM #37`. Likewise, to see only the instructions associated with a procedure called `PROC_A`, enter `ASM PROC_A`.

Patching Code

You can also use the `ASM` command to assemble code into memory. For example, suppose you wanted to replace the code at line number 50 in the user program with the instruction sequence `MOV AX,50H` and then `PUSH AX`. The following command sequence would do this:

```
*ASM #50 = 'MOV AX,50H', 'PUSH AX'
```

When you are assembling instructions into memory, make sure that their addresses do not conflict with addresses of other instructions that you want to keep.

If an invalid instruction is encountered during disassembly, double question marks (??) are displayed in place of an instruction.

NOTE

You can use `ASM` interchangeably with its alias, `SASM`.

Examples

1. In this example, ASM disassembles ten lines of assembly code, starting at the address of the current execution point.

```
*ASM $ LENGTH 10
:PLMTST
1D3D:0000h 8BEC          MOV BP,SP
1D3D:0002H FB           STI
1D3D:0003H E84700       CALL $+004AH
#54
1D3D:0006H 803E1000E     CMP BYTE PTR 0010H,0EH ; 14T
1D3D:000BH 7403         JZ $+0005H
1D3D:000DH E92300       JMP $+0026H
#56
1D3D:0010H C70602004000  MOV WORD PTR 0002H,0040H ;
64T
#57
1D3D:0016H 813E02002800  CMP WORD PTR 0002H,0028H ; 40T
1D3D:001CH 7703         JA $+0005H
1D3D:001EH E90900       JMP $+000CH
```

2. In this example, ASM assembles the PUSHF and CALL instructions starting at address 2089H.

```
*ASM 2089H = 'PUSHF',
**'CALL 31A0H'
*
```

3. In this example, ASM assembles two MOV instructions, starting at address CS:31A0H.

```
*ASM CS:31A0H = 'MOV AX,40', 'MOV ES,AX'
*
```

ASM (continued)

Cross-references

Encyclopedia Entries

CALLS
LOAD
Memory Access
STACK

Menu/Item

Window/ASM Display

Keyboard Control

F3 Toggle ASM display

BASE

Displays or establishes the default base for numeric constants

Syntax

```
BASE = [BINARY    2T  
        DECIMAL  10T  
        HEX      16T  
        expr]
```

Where:

BINARY defines binary as the current radix.

DECIMAL defines decimal as the current radix.

HEX defines hexadecimal as the current radix.

expr is an expression that evaluates to 2 for binary, 10 for decimal, or 16 for hexadecimal.

Discussion

The BASE command displays the default number base (radix) for numeric constants. Display of user-program variables and debug variables is also governed by the BASE command. Integers and floating-point numbers are always displayed in decimal.

When used with the assignment statement, BASE =, BASE sets the default base to binary, decimal, or hexadecimal. During input, you can override the default base setting of a constant with an explicit base suffix: Y for binary, T for decimal, H for hexadecimal (e.g., 10H).

You can also use BASE as an expression within other commands and as a variable (e.g., *variable* = BASE). The type of the BASE pseudovvariable is BYTE.

The default BASE is hexadecimal.

BASE (continued)

Examples

1. In this example, BASE displays the current base.

```
*BASE  
HEX
```

2. In this example, BASE changes the default base to decimal. The suffix T ensures that the base is changed to decimal. Without the suffix, the specified base is interpreted as a number in the current base. For example, if the current base is hexadecimal, the command BASE = 10 does not change the base to decimal since the 10 is interpreted as a hexadecimal value.

```
*BASE = 10T  
*BASE  
DECIMAL
```

Cross-reference

Encyclopedia Entry

Expressions

Breakpoints

Set breakpoints, tracepoints, or watch expressions; or display the current breakpoint, tracepoint, or watch expression

Discussion

Breakpoint commands set breakpoints, tracepoints, or watch expressions; or display the current breakpoint, tracepoint, or watch expression. Breakpoint commands and the functions they perform are listed in Table 5-8.

Table 5-8 DB86 Breakpoint Commands

Breakpoint Command	Action
<i>CBn = aexpr, bexpr</i>	Sets a conditional breakpoint
<i>FBn = addr</i>	Sets a fixed breakpoint
<i>TRn = aexpr[expr-list]</i>	Sets a tracepoint
<i>WAn = aexpr, bexpr</i>	Sets a watch expression

Breakpoint status for the current execution line is shown in the Break Status Column in the View Window. The debugger indicates different types of breaks with the following symbols. An enabled breakpoint is shown in uppercase; a disabled (or defined in a nonresident overlay, but inactive) breakpoint is shown in lowercase. A hidden breakpoint is one that is set within the range of a program line but is not exactly on the line boundary; it is always shown as +.

- B Temporary breakpoint
- F Fixed breakpoint
- + Hidden breakpoint
- T Tracepoint
- C Conditional breakpoint
- E Exception breakpoint (used internally by the debugger)

Breakpoints (continued)

Cross-references

Encyclopedia Entries

CB	GO	RSTEP
CSTEP	ISTEP	TR
ENABLE/DISABLE	LSTEP	WA
FB	PSTEP	

Menus/Items

Go Menu
Set Menu

Keyboard Controls

F5	Go, all breakpoints
sF5	Go forever
F7	Go til cursor line
F9	Set temporary breakpoint
sF9	Set tracepoint

Syntax

CALLS [EXP]

Where:

EXP includes the stack resident variables for each procedure in the display.

Discussion

The CALLS command displays the current chain of procedure calls on the stack, giving you a dynamic view of procedure call nesting. Fully qualified references to procedures are listed. The first reference is to the current execution point. The second reference is to the return point, to which execution control will return when the current procedure exits.

CALLS operates correctly only when the assembly prologue of the current procedure has been executed. That is, the instruction pointer must be pointing to the first executable statement of the current procedure, rather than to the entry point of the procedure.

If the EXP control is used, the stack resident variables for each procedure in the call chain are also displayed.

NOTE

The CALLS command does not operate correctly if the nesting sequence includes a procedure written in assembly language.

You can use CALLS interchangeably with its alias, CALLSTACK.

CALLS (continued)

Examples

1. In this example, CALLS displays the procedure calls that are on the stack when get_char occurs. As indicated in the example, the nesting level is shown for each procedure followed by the address of the procedure, followed by the procedure name.

```
*LOAD dc.86
*GO TIL get_char
[Break at :DC#74]
*CALLS
[0] :DC#74 READ_CHAR
(1) :DC#85 PROCÈSS_TOKEN
(2) :DC#102 LEX_SCAN
(3) :DC#80 PROCÈSS_TOKEN
(4) :DC#255 PARSE_LINE
(5) :DC#600
```

2. In this example, the CALLS command displays an expanded listing that shows the variables for each procedure:

```
*calls exp
[0] :EXAMPIO#126 CLS
      REG REGS (structure
      AX 02CA
      BX 4F29
      CX 0020
      DX 4FA0
      SI 0003
      DI 01E3
      DS 4F29
      ES 0116
)
(1) :MENU#97 MENU_MEMDUMP
      DONE +0
      PTR 4FA7H:0157H
      KEY 0003
(2) :MENU#272 MAIN
      MI +3
      DONE +0
(3) :CALLMAIN#5
```

Cross-references

Encyclopedia Entries

Memory Access
STACK

Menus/Items

View/Calls
View/Expanded Calls

CB

Sets a conditional breakpoint or displays the current conditional breakpoint

Syntax

CB[*n*] [= *aexpr*, *bexpr*]

Where:

- n* is a number, from 0 to 3 inclusive, assigned to a debugger conditional breakpoint.
- aexpr* is an expression representing the address at which the Boolean expression is evaluated.
- bexpr* is a Boolean expression that is evaluated at the address defined by *aexpr*. If the Boolean expression is true (i.e., Least-Significant Bit (LSB) = 1), the breakpoint is taken; otherwise, user-program execution continues.

Discussion

The conditional breakpoint command, CB, sets a conditional breakpoint at the specified address: when the address is encountered and the condition is true, program execution stops. Up to four conditional breakpoints are available: CB0 through CB3.

You can enable all or specified conditional breakpoints with the ENABLE command and disable them with the DISABLE command. All breakpoints are cleared with the GO FOREVER command. The Clear at Cursor and Remove All items on the Set Menu remove conditional breakpoints.

In Auto Step mode, the debugger evaluates a conditional breakpoint at each program step. This feature allows you to test for program conditions at every point in the program flow rather than at a unique address only.

To show the current definition of a conditional breakpoint, enter CB*n*. Use the DIR CB command to see the currently defined conditional breakpoints.

Examples

1. In this example, CB breaks at line 15 in module :MYMOD when count_var is greater than 5.

```
*CBO = :MYMOD#15,count_var > 5
```

2. In this example, CB breaks at line number 30 of the module :YOURMOD if xy equals 5 or if mtop is less than 25 hexadecimal.

```
*CB1 = :YOURMOD#30, (xy == 5) OR (mtop < 25H)
```

3. In this example, CBO is enabled.

```
*ENABLE CBO
```

Cross-references

Encyclopedia Entries

Breakpoint Commands	FB	PSTEP
CSTEP	GO	RSTEP
DIR	ISTEP	TR
ENABLE/DISABLE	LSTEP	WA

Menus/Items

Go/Til Cursor Line
 Set/Breakpoint
 Set/Clear at Cursor
 Set/Remove All

Keyboard Controls

F7 Go til cursor line
 F9 Set temporary breakpoint

Command Entry

Includes command-line interface, continuation lines, multiple command lines, comments, editing, and display control

Command-line Interface and Windowed Menu System

You can issue debugger commands through the command-line interface at the bottom of the screen or through the windowed menu system at the top of the screen.

When you start the debugger, you see the asterisk (*) prompt in the Command Window, waiting for input. You can enter commands at the prompt or toggle control to the View Window or the menu system. Type Alt-M (for Menus) and, using the arrow keys, move to the menu you wish to use. See Chapter 3 for more information on the menu system. Press the F6 key to change control to the View Window. See Chapter 4 for more information on the controls available in the View Window.

Continuing Command-lines

You can continue a command line to the next line by typing an ampersand (&) at the end of the line. The continuation character is most useful when you want to type a lengthy sequence of commands before any of the commands are executed.

You do not need the ampersand at the end of a command line if the line ends before the command is complete. The debugger can tell when a command is complete, and it requests additional input, with a double asterisk (**) prompt, if you have not finished a command. Be careful not to leave out a necessary continuation character; it is possible to type a complete, valid command before you have finished entering the command you had in mind.

You can use ampersands within strings; they are not be interpreted as continuation characters.

Multiple Command Lines

You can type multiple commands on a single line, separating each command from the next with a semicolon (;).

Comments

You can include comments, enclosing them in comment brackets: /* and */. A comment can appear anywhere a space, tab, or new line is allowed.

Editing

You can edit command lines typed at the command-line prompt, using the debugger line-editing controls. Line-editing controls are summarized in Table 5-9. See Chapter 4 for a complete description of line-editing controls.

When you have completed a debugger command-line (and with the cursor anywhere in the command line) press Enter to enter the command into the system.

When the debugger has more information to display in a given window than room in which to display it, you can control the display using Fast mode (F), Line mode (L), or Page mode (P). Display output control keys and the actions they perform are listed in Table 5-10.

Command Entry (continued)

Table 5-9 DB86 Command-line Editing

Key	Action
Grey ←	Deletes character to the left (backspace)
Del or Ctrl-F	Deletes character at the cursor
Ctrl-X	Deletes line to left of cursor
Ctrl-A	Deletes line to right of cursor
Ctrl-Z	Deletes current line
←	Moves cursor one character left
→	Moves cursor one character right
↑	Restores previous command line from history buffer
↓	Scans to next command line in history buffer
Home	Goes to the start of current command line
End	Goes to the end of current command line
Esc	Cancels current command line
Ctrl-C or Ctrl-Break	Cancels command in progress
Ctrl-E	Re-executes previous command

Command Entry (continued)

Table 5-10 Display Output Control Keys

Key	Action
F(ast mode)	Continues output without pause
L(ine mode)	Displays one line of output at a time
P(age mode)	Displays one window of output at a time

Cross-references

Encyclopedia Entry

Menu System

Menus/Items

Help/Edit Keys

Help/Navigation Keys

Keyboard Controls

Edit Keys

Navigation Keys

Control Constructs

Group commands and control program execution

Discussion

Control constructs, paired commands such as DO ... END and IF ... END[IF] permit you to set up conditions for controlling program execution. Debugger control constructs and the actions they perform are listed in Table 5-11.

Table 5-11 DB86 Control Constructs

Control Construct	Action
COUNT ... END[COUNT]	Executes a loop a specified maximum number of times
COUNT ... WHILE ... END[COUNT]	Executes a loop a specified maximum number of times while a specified condition is met
COUNT ... UNTIL ... END[COUNT]	Executes a loop a specified maximum number of times, until a specified condition is met
DO ... END	Executes the commands between the DO and the END, grouping the set of commands into a unit
IF ... THEN ... END[IF]	Executes a set of commands between THEN and END if the specified condition is true
IF ... THEN ... ELSE ... END[IF]	Executes a set of commands between THEN and ELSE if the specified condition is true; otherwise, executes the commands between ELSE and END

Control Constructs (continued)

Table 5-11 DB86 Control Constructs (continued)

Control Construct	Action
REPEAT ... END[REPEAT]	Executes a set of commands between REPEAT and END forever
REPEAT ... WHILE ... END[REPEAT]	Executes a set of commands between REPEAT and END as long as the specified condition is met
REPEAT ... UNTIL ... END[REPEAT]	Executes a set of commands between REPEAT and END until the specified condition is met

Cross-references

Encyclopedia Entries

COUNT ... ENDCOUNT
DO ... END
IF ... ENDIF
REPEAT ... ENDREPEAT

COUNT ... ENDCOUNT

Group and execute commands
a specified number of times

Syntax

```
COUNT expr  
  {debugger_commands}  
  [WHILE bexpr]  
  [UNTIL bexpr]  
END[COUNT]
```

Where:

expr is an expression that specifies how many times the commands between COUNT and END are to be executed.

debugger_commands are one or more debugger commands except HELP, INCLUDE, and LOAD.

bexpr is a Boolean expression, evaluated to either true (Least-Significant Bit (LSB) = 1) or false (LSB = 0).

WHILE *bexpr* evaluates the Boolean expression to determine when execution stops (i.e., when the WHILE expression is false or the COUNT is reached).

UNTIL *bexpr* evaluates the Boolean expression to determine when execution stops (i.e., when the UNTIL expression is true or the COUNT is reached).

Discussion

The COUNT ... ENDCOUNT construct executes debugger commands a specified maximum number of times. The COUNT expression is converted to a word value and is evaluated only once, when the statement is first encountered.

You can nest COUNT blocks; the number of dots before the command-line prompt indicates the nesting level.

Example

In this example, COUNT ... ENDCOUNT steps through the program as LSTEP would, until either 100 lines have been executed or the execution point is at line number 30.

```
*COUNT 100t  
.*LSTEP  
.*WHILE $ < > #30  
.*ENDCOUNT
```

Cross-references

Encyclopedia Entries

Control Constructs
Debugger Commands
DO ... END

Expressions
IF ... ENDIF
REPEAT ... ENDREPEAT

CSTEP

Steps into the next procedure call encountered

Syntax

CSTEP

Discussion

The CSTEP command steps by machine instruction using the ISTEP command until the next CALL instruction is encountered. It then steps into the procedure call using one ISTEP instruction and displays the next executable line.

CSTEP allows you to step from the current execution point and nest into the next encountered procedure call.

Cross-references

Encyclopedia Entries

Breakpoint Commands	GO	RSTEP
CB	ISTEP	TR
ENABLE/DISABLE	LSTEP	WA
FB	PSTEP	

Menu/Item

Go/Call Step

Keyboard Control

sF8 Call step

Data Types

Are the standard types, used in commands and displays, with which the debugger is familiar

Discussion

Data types, the standard types with which the debugger is familiar, are listed in Table 5-12. You cannot create new types within the debugger. User-defined variables, which have been defined in a user program at compilation, cannot refer to debug variables.

Table 5-12 DB86 User-program Types

Type	Description
Unsigned	
ADDRESS	16-bit quantity in current base
BYTE	8-bit quantity
DWORD	32-bit quantity in current base
SELECTOR	16-bit quantity representing a segment (paragraph), in current base
WORD	16-bit quantity in current base
Signed	
INTEGER	16-bit quantity in Base 10
LONGINT	32-bit quantity in Base 10
SMALLINT	8-bit quantity in Base 10
EXTINT	64-bit quantity in Base 10
Composite	
ARRAY	Composite
STRUCTURE	Composite
Floating-point	
LONGREAL	64-bit floating-point number
REAL	32-bit floating-point number
TEMPREAL	80-bit floating-point number representing intermediate real values

Data Types (continued)

Table 5-12 DB86 User-program Types (continued)

Type	Description
Pointer POINTER	32-bit selector:offset pair
Boolean BOOLEAN	True (LSB = 1) or False (LSB = 0)
Character CHAR	ASCII character string
Assembler ASM	Assembler language mnemonic

Cross-references

Encyclopedia Entries

- Debug Variables
- DEFINE
- Expressions
- User-program Symbols

Debug Variables

Store temporary values during a debug session

Discussion

Debug variables can store temporary values during a debug session.

To display the value of a debug variable, simply type *name*, where *name* is the name of the variable.

To define a new debug variable, use the DEFINE command.

To change a debug variable, type *name* = .

Cross-references

Encyclopedia Entries

Data Types

DEFINE

DIR

Debugger Commands

Are the primary commands in the debugger command-language

Discussion

Debugger commands, the primary commands in the debugger command-language, enable you to set breakpoints, tracepoints, and watch expressions; control program execution; and perform a variety of debugging functions.

Debugging commands are available through the command-line interface, the menu system, and the keyboard.

Cross-references

Encyclopedia Entries

- Command Entry
- HELP
- Menu System

Menu/Item

- Help Menu

DEFINE

Defines a global or local debug variable

Syntax

```
DEFINE [GLOBAL] type name [= expr]
```

Where:

GLOBAL specifies that *name* is global, whether or not it appears in any DO ... END constructs.

type specifies the data type of the debug variable. The following list presents debugger data types:

ADDRESS	LONGINT
ASM	LONGREAL
BOOLEAN	POINTER
BYTE	REAL
CHAR	SELECTOR
DWORD	SMALLINT
EXTINT	TEMPREAL
INTEGER	WORD

name is the user-specified *name* for the debug variable.

expr is an expression that assigns an initial value to the variable.

Discussion

The DEFINE command defines a debug variable. The variable is global if defined at the the outer command level (that is, if it is not enclosed in a DO ... END construct) or if you use the GLOBAL option. Otherwise, the variable is local, and so is known only within the DO ... END block in which it appears.

The debug variable has the same properties as a program variable of the same type, except that it cannot be prefaced with a dot.

DEFINE (continued)

You can assign an initial value to the debug variable. If no value is given, the variable has a default value of zero (or is a string of zero length) and its Boolean value is set to false. You can assign a new value to a debug variable as long as the new value does not change the type.

Example

In this example, DEFINE defines two variables, a pointer and a string.

```
*DEFINE pointer x = :LISTMOD #50
*EVAL x LINE
:LISTMOD #50
*DEFINE char s = 'This is a string'
*s
*This is a string
```

Cross-references

Encyclopedia Entries

- Data Types
- Debug Variables
- DO ... END
- Expressions

DIR

Displays a listing
of symbolic debug information

Syntax

```
DIR [PUBLIC] [ [ mtype | stype ]  
               [ module-name ] [ mtype | stype | LINE | SETMOD ]  
               MODULE  
               DEBUG  
               break-type ] ]
```

Where:

PUBLIC lists all public symbols.

mtype qualifies the display of symbols so that only debug variables of memory type *mtype* are displayed. The following list presents *mtype* variables:

ARRAY	LONGREAL
BOOLEAN	POINTER
BYTE	PROCEDURE
CHAR	REAL
DWORD	RECORD
ENUMERATION	SET
FILE	SMALLINT
INTEGER	TEMPREAL
LONGINT	WORD

See Table 5-13 for language-specific information.

stype qualifies the display of user symbols so that only variables of special user program type *stype* are displayed. ARRAY, RECORD, PROCEDURE, and LABEL are all *stype* variables.

module-name names a specific module whose public symbols are to be listed. The current module is used when *module-name* is not specified.

LINE displays the line numbers of executable statements. These line numbers are generated at compile time.

DIR (continued)

MODULE	displays the names of all modules currently loaded in the symbol table.
SETMOD	lists all correlations between modules of the loaded program and list files; these correlations are set by the user and by the debugger. The display includes module name and path name.
DEBUG	Displays the current definitions of breakpoints, tracepoints, watch expressions, and debug variables.
<i>break-type</i>	is one of the following breakpoints, tracepoints, or watch expressions: CB, FB, TR, WA.

Discussion

The DIR command, without options, displays a listing of symbols in the current module. Symbols are indented to show their scope.

If the symbol type is a structure, the elements of the structure are displayed. If the symbol type is an array, the array bounds are displayed. Dynamic and base attributes are also displayed.

If the module has not yet been accessed by the debugger and the SPATH extension list has not been applied, the module extension is shown as an asterisk (*) (e.g., TEST1.*).

If all SPATH extensions have been applied to the file and no list file match is found, "None" is displayed.

To display the active values of breakpoints, tracepoints, or watch expressions, type *break-type*.

Table 5-13 DB86 mtypes and their Language-specific Names

mtype	ASM-86	PL/M	FORTTRAN-86	iC-86	Pascal-86
byte	BYTE	BYTE		UNSIGNED SHORT	BYTE
ADDRESS		ADDRESS		<i>*var</i>	
word	WORD	WORD		UNSIGNED	WORD
dword	LONG	DWORD		UNSIGNED LONG	
real	REAL	REAL	REAL*4	REAL	REAL
smallint			INTEGER*1	SIGNED CHAR	
integer		INTEGER	INTEGER*2	INTEGER	INTEGER
longint		LONGINT	INTEGER*4	LONG	LONGINT
char		CHAR	CHARACTER	UNSIGNED CHAR	CHAR
boolean			LOGICAL*1		BOOLEAN
array		ARRAY	ARRAY	ARRAY	ARRAY
structure		STRUCTURE		STRUCT	RECORD
proce- dure		PROCEDURE	FUNCTION	FUNCTION	PROCE- DURE

DIR (continued)

Table 5-13 DB86 mtypes and their Language-specific Names (continued)

mtype	ASM-86	PL/M	FORTRAN-86	iC-86	Pascal-86
pointer		POINTER	POINTER	<i>far*var</i>	@ <i>var</i>
long-real	QWORD		REAL*8	double	
temp-real	TBYTE				
set					SET
file					FILE
enumeration				enum	ENUMERATION

Example

In this example, DIR displays a listing showing the correlation between each module to be debugged (first column) and its list file (second column).

```
*DIR SETMOD
ECRYPTION    \\list.dir\ECRYPTION.LST
MYPROG       \\list.dir\MYPROG.LST
YOURPROG     \\list.dir\YOURPROG.LST
TEST1        \\list.dir\TEST1.*
CHKFIL       None
```


Cross-references

Encyclopedia Entries

Data Types
SETMOD

Menus/Items

View/Locals
View/Modules
View/Debug

DO ... END

Group and execute commands

Syntax

```
DO
  {debugger_commands}
END
```

Where:

debugger_commands can be any debugger commands except HELP, INCLUDE, and LOAD.

Discussion

The DO ... END construct groups one or more debugger commands in a block and executes the commands.

You can nest DO blocks; the number of dots before the * prompt indicates the nesting level.

Example

In this example, the DO ... END construct groups a series of commands, which test the value of a variable, var1, after a breakpoint occurs at line number 50.

```
*DO
.*GO TIL #50
.*IF var1 == 25T THEN
..*'TRUE'           /*display TRUE on console*/
..*ELSE
..*'FALSE'         /*display FALSE on console*/
..*ENDIF
.*END
[Break at :mod #50]
TRUE
```

Cross-references

Encyclopedia Entries

Control Constructs
COUNT ... ENDCOUNT
Debugger Commands
IF ... ENDIF
REPEAT ... ENDREPEAT

ENABLE/DISABLE

Enables or disables a specified breakpoint, tracepoint, or watch expression; or enables or disables all breakpoints, tracepoints, or watch expressions

Syntax

```
[ENABLE ]  
[DISABLE] [break-type-list]
```

Where:

break-type-list lists either all or specified conditional breakpoints, fixed breakpoints, tracepoints, or watch expressions. The items in the list are separated by commas. The following list presents debugger breakpoints, tracepoints, and watch expressions:

- Conditional Breakpoints

CB0
CB1
CB2
CB3

- Fixed Breakpoints

FB0
FB1
FB2
FB3
FB4
FB5

- Tracepoints

TR0 TR5
TR1 TR6
TR2 TR7
TR3 TR8
TR4 TR9

ENABLE/DISABLE (continued)

- Watch Expressions

WA0
WA1
WA2
WA3
WA4
WA5

Discussion

The ENABLE command enables all conditional breakpoints, fixed breakpoints, tracepoints, or watch expressions. To enable a particular breakpoint, tracepoint, or watch expression, enter ENABLE CB*n*, FB*n*, TR*n*, or WA*n*.

The DISABLE command disables all conditional breakpoints, fixed breakpoints, tracepoints, or watch expressions. To disable a particular breakpoint, tracepoint, or watch expression, enter DISABLE CB*n*, FB*n*, TR*n*, or WA*n*. All conditional and fixed breakpoints and tracepoints are cleared with the GO FOREVER command.

Disabled breakpoints and tracepoints do not halt the user program that is executing under debugger control. Disabled watch expressions are not shown in the Watch Window.

Use the DIR command to show currently defined conditional breakpoints (DIR CB), fixed breakpoints (DIR FB), tracepoints (TR), and watch expressions (WA).

Examples

1. In the following example, all watch expressions are enabled.

```
*ENABLE WA
```

2. In the following example, fixed breakpoint number 3 and conditional breakpoint number 1 are disabled.

```
*DISABLE FB3, CB1
```

ENABLE/DISABLE (continued)

Cross-references

Encyclopedia Entries

Breakpoint Commands	FB
CB	TR
DIR	WA
GO FOREVER	

Menus/Items

Go/Forever
Go/Til Breakpoint
Go/Keep Fixed

Keyboard Controls

F5	Go, all breakpoints
sF5	Go forever
sF7	Go, fixed breakpoints

EVAL

Calculates and displays
the result of an expression

Syntax

```
EVAL expr [OV ov-name] [LINE  
PROCEDURE  
SYMBOL]
```

Where:

expr is an expression.

OV *ov-name* is the name of an overlay. If you wish to evaluate to a symbolic reference in an overlay that is not present, OV forces the symbolic reference to *ov-name*. *ov-name* is assigned by the user at link-time when linking with LINK86.

LINE displays a line number, in the form *module-name#line-number*. If address mapping was not exact, this message can include *+offset*, where *offset* is the difference in bytes between *module-name#line-number*. If the debugger has no line number information for the module, offset is calculated from the start of the module.

PROCEDURE displays a procedure, in the form *module-name.procedure-name*. This message can include *+offset*, as described in LINE, above.

SYMBOL displays the symbol associated with an address specified in *expr*, in the form *module_name.procedure_name.symbol*. This message can include *+offset*, as described in LINE, above. When *expr* evaluates to a module with no labels, or with no labels before the expression, SYMBOL displays the module name.

EVAL (continued)

Discussion

The EVAL command calculates and displays the result of an expression. When used with LINE, EVAL calculates and displays the module name and line number closest to *expr*. When used with PROCEDURE, EVAL calculates and displays the module name and procedure closest to *expr*. When used with SYMBOL, EVAL calculates and displays the label or data variable closest to *expr*.

Most results are displayed in binary, decimal, hexadecimal, and ASCII. Negative integers are displayed with negative signs. ASCII is displayed as a string enclosed in apostrophes ('*string*'); any nonprinting characters are displayed as periods (.). Floating-point calculations are displayed as 10 bytes, in hexadecimal form. Real types with no calculations are displayed in the number of bytes appropriate to their type; that is, a real is displayed as 4 bytes, a longreal as 8 bytes, and a tempreal as 10 bytes.

Examples

1. In this example, EVAL calculates and displays the current execution point.

```
*EVAL $ LINE
:MOD_2#3
*
```

2. In this example, EVAL determines in which procedure the address 1024:40 is.

```
*EVAL 1024:40 PROCEDURE
:MOD_2.PROC_ONE+5
*
```

3. In this example, EVAL evaluates the result of a combination of program variable expressions and a constant.

```
*EVAL 13 + (var1 - j)
01101110Y 110T 6EH 'n'
```


4. In this example, EVAL determines which procedure in overlay OVERLAY_1 corresponds to address 1234:56.

```
*EVAL 1234:56 OV OVERLAY_1 PROCEDURE  
:MOD6.LOAD  
*EVAL DS:1234 SYMBOL  
:INIO.FIELD_IDX
```

Cross-reference

Encyclopedia Entry

Expressions

EXIT

Terminates the debug session
and returns control
to the host operating system

Syntax

EXIT

Discussion

The EXIT command closes all open files, terminates the debug session, and returns to the operating system.

Example

In this example, EXIT exits the debugger and returns to DOS.

```
*EXIT  
C>
```

Cross-references

Encyclopedia Entry

! DOS Shell

Menus/Items

Debug/Exit DB86
Debug/DOS Shell

Expressions

Are one or more numbers, debug variables, or functions separated by operators

Discussion

An expression is a single value or a combination of operands (one or more numbers, debug variables, strings, or control variables) separated by operators.

Operands

Debugger operands include the following:

- Debug variables
- Keywords
- Line numbers
- Numbers
- Pseudovariables BASE, PORT, WPORT
- Real numbers
- User program symbols

Operators

The debugger recognizes the operators shown in Table 5-14. The operators are grouped according to precedence, with the unary operator class having the highest precedence and arithmetic operator class having the lowest precedence. The operators are also grouped according to precedence within each class.

MOD is valid with Boolean operands and real numbers.

Expressions (continued)

Table 5-14 Operators

Operator	Function
Arithmetic Operators	
+	unary and binary plus
-	unary and binary minus
*	multiplication
/	division
MOD	modulo (remainder)
Boolean Operators	
NOT	Boolean NOT
AND	Boolean AND
OR	Boolean OR
XOR	Boolean XOR
Relational Operators	
< >	not equal
>	greater than
<	less than
< =	less than or equal to
> =	greater than or equal to
= =	equals

Types and Their Valid Operators

Table 5-15 shows which operators are valid for which types.

Table 5-16 shows which binary operators are valid between operands of different types.

Table 5-15 Types and their Valid Operators

Type	Valid Operators Unary	Binary
Unsigned	+, -, NOT	All operators
Signed	+, -	All operators
Floating-point	+, ,, -	Unary and binary, relational operators
Pointer	None	+, -, relational operators
Boolean	NOT	Boolean, relational operators
Character	None	None

Expressions (continued)

Table 5-16 Disparate Types and their Valid Binary Operators

Type	Type	Valid Binary Operators
Unsigned	Signed	All operators
	Floating-point	Unary and binary, relational operators
	Pointer	Relational operators, +, -
	Boolean	Boolean, relational operators
	Character	All operators
Signed	Unsigned	All operators
	Floating-point	Unary and binary, relational operators
	Pointer	Relational operators, +, -
	Boolean	Boolean, relational operators
	Character	None
Floating-point	Unsigned	Unary and binary, relational operators
	Signed	Unary and binary, relational operators
	Pointer	Relational operators, +, -
	Boolean	Boolean, relational operators
	Character	None

Expressions (continued)

Table 5-16 Disparate Types and their Valid Binary Operators (continued)

Type	Type	Valid Binary Operators
Pointer	Unsigned	Relational operators, +, -
	Signed	Relational operators, +, -
	Floating-point	Relational operators, +, -
	Boolean Character	None None
Boolean	Unsigned	Boolean, relational operators
	Signed	Boolean, relational operators
	Floating-point	Boolean operators
	Pointer	None
	Character	None
Character	Unsigned	All operators
	Signed	None
	Floating-point	None
	Pointer	None
	Boolean	None

Expressions (continued)

Expression Typing

When the debugger evaluates expressions that contain different data types, it automatically performs any necessary type conversions, as shown in Table 5-17. The debugger also performs conversions when a particular type is required by the syntax (e.g., in `COUNT expr`, *expr* becomes a word).

5-17 Type Conversions in Expressions

Expression	Resulting Type
Unsigned	
<i>operator</i> Unsigned	DWORD
<i>operator</i> Signed	LONGINT
<i>operator</i> Floating-point	TEMPREAL
<i>operator</i> Pointer	POINTER
<i>operator</i> Boolean	BOOLEAN
<i>operator</i> Character	DWORD
Signed	
<i>operator</i> Unsigned	LONGINT
<i>operator</i> Signed	LONGINT
<i>operator</i> Floating-point	TEMPREAL
<i>operator</i> Pointer	POINTER
<i>operator</i> Boolean	BOOLEAN
<i>operator</i> Character	Incompatible; error message
Floating-point	
<i>operator</i> Unsigned	TEMPREAL
<i>operator</i> Signed	TEMPREAL
<i>operator</i> Floating-point	TEMPREAL
<i>operator</i> Pointer	POINTER
<i>operator</i> Boolean	BOOLEAN
<i>operator</i> Character	Incompatible; error message

Table 5-17 Type Conversions in Expressions (continued)

Expression	Resulting Type
Pointer	
<i>operator</i> Unsigned	POINTER
<i>operator</i> Signed	POINTER
<i>operator</i> Floating-point	POINTER
<i>operator</i> Pointer	DWORD
<i>rel operator</i> Pointer	BOOLEAN
<i>operator</i> Boolean	Incompatible; error message
<i>operator</i> Character	Incompatible; error message
Boolean	
<i>operator</i> Unsigned	BOOLEAN
<i>operator</i> Signed	BOOLEAN
<i>operator</i> Floating-point	BOOLEAN
<i>operator</i> Pointer	Incompatible; error message
<i>operator</i> Boolean	BOOLEAN
<i>operator</i> Character	Incompatible; error message
Character	
<i>operator</i> Unsigned	DWORD
<i>operator</i> Signed	Incompatible; error message
<i>operator</i> Floating-point	Incompatible; error message
<i>operator</i> Pointer	Incompatible; error message
<i>operator</i> Boolean	Incompatible; error message
<i>operator</i> Character	Incompatible; error message

Expressions (continued)

Typing in an Assignment Operation

When the debugger combines different types in an assignment operation, it automatically performs any necessary type conversions, as shown in Table 5-18. In direct mapping conversions, the debugger expands the source operand to maximum precision and then truncates it to the destination type if necessary. In tests, the result is true if the least-significant bit is 1, false if the least-significant bit is 0.

5-18 Type Conversions in Assignments

Assignment	Resulting Type
Unsigned =	
Unsigned	Direct mapping
Signed	Direct mapping
Floating-point	Fixed point
Pointer	Direct mapping
Boolean	Incompatible; error message
Character	Direct mapping
Signed =	
Unsigned	Direct mapping
Signed	Direct mapping
Floating-point	Fixed point
Pointer	Direct mapping
Boolean	Incompatible; error message
Character	Incompatible; error message
Floating-point =	
Unsigned	Floating-point
Signed	Floating-point
Pointer	Incompatible; error message
Boolean	Incompatible; error message
Character	Incompatible; error message

Table 5-18 Type Conversions in Assignments (continued)

Assignment	Resulting Type
Pointer =	
Unsigned	Direct mapping
Signed	Incompatible; error message
Floating-point	Incompatible; error message
Boolean	Incompatible; error message
Character	Incompatible; error message
Boolean =	
Unsigned	Test
Signed	Test
Floating-point	Test
Pointer	Incompatible; error message
Character	Incompatible; error message
Character =	
Unsigned	Direct mapping
Signed	Incompatible; error message
Floating-point	Incompatible; error message
Pointer	No conversion necessary
Boolean	Incompatible; error message

Cross-references

Encyclopedia Entry

- Data Types
- Debug Variables
- Pseudovariables
- User-program Symbols

FB

Sets a fixed breakpoint
or displays a current
fixed breakpoint

Syntax

FB[*n*] [= *aexpr*]

Where:

n is a number, from 0 to 5 inclusive, assigned to a debugger breakpoint.

aexpr is an expression that evaluates to the address at which program execution is to be halted.

Discussion

The fixed breakpoint command, FB, sets a new breakpoint at the specified address. This breakpoint remains in effect regardless of a GO TIL condition. Up to six fixed breakpoints are available: FB0, FB1, FB2, FB3, FB4, and FB5.

You can enable fixed breakpoints with the ENABLE command and disable them with the DISABLE command. All breakpoints are effectively disabled with the GO FOREVER command.

For the current definition of a fixed breakpoint, enter FB*n*.

Use the DIR FB command to display the current definitions of all fixed breakpoints.

Examples

1. In this example, FB displays the current definition of fixed breakpoint number 0.

```
*FBO  
:vst1.vst1_enter_module
```

2. In this example, FB sets a breakpoint at line number 37 of the current module.

```
*FB1 = #37
```

Cross-references

Encyclopedia Entries

Breakpoint Commands	GO
CB	TR
DIR	WA
ENABLE/DISABLE	

FLAGS

Displays or changes
8086/8088 flags

Syntax

```
[ FLAGS  
  flag-name ] [= bexpr]
```

Where:

bexpr is an expression that represents a Boolean value.

flag-name is one of the following flag bits:

OFL	Overflow flag
DFL	Direction flag
IFL	Interrupt flag
SFL	Sign flag
ZFL	Zero flag
AFL	Auxiliary flag
PFL	Parity flag
CLF	Carry flag

Description

The FLAGS command displays or sets 8086/8088 flags.

Use the FLAGS form to display the flag word in the current radix. To display the flag word in another radix, supply the desired base suffix.

The flag word is shown in the Register Window as an 8-letter value (e.g., ODisZAPC) in which each letter stands for one of the processor flags listed above. An uppercase letter indicates that the flag is set (logic 1) and a lowercase letter indicates that the flag is not set (logic 0).

Use the *flag-name* form to reference the desired flag. To set all the flags (i.e., the flag word) to a different value, use the FLAGS = *expr* form.

Examples

1. In this example, FLAGS displays the 8086/8088 flags in binary; BASE sets the radix to 2.

```
*BASE = 2
*FLAGS
1111000101100101
```

2. In this example, FLAGS sets the Carry flag to true.

```
*CFL = TRUE
```

Cross-references

Encyclopedia Entries

- Expressions
- FLAGS
- Registers and Flags
- REGS
- REGS87

Menu/Item

- Window/Registers

Keyboard Control

- F2 Toggle Register Window

GO

Starts program execution
and controls breakpoints and tracepoints

Syntax

```
GO [FOREVER  
   TIL expr [, expr1 ... expr9 ] ]
```

Where:

- FOREVER** clears all active breaks and executes the program from the CS:IP (\$) to the end of the program.
- TIL** executes the program from the CS:IP (\$) until a specified address is reached. The comma (,) separating expressions represents an OR function.
- expr* is an expression that can be converted into an address in the object code being debugged.

Discussion

The GO command begins execution of the user program at the LOAD address or from the previous breakpoint.

To set temporary breakpoints, use the GO TIL *expr* {, *expr1* ... *expr9*} form; in this way, you can set up to ten temporary breakpoints. Temporary breakpoints remain in effect until you specify another GO TIL or GO FOREVER.

To change the execution address, assign a value to dollar sign (\$), a pseudovisible that represents the current execution point.

NOTE

When you change the execution address with \$, you may invalidate the run-time stack as a consequence of disrupting the normal flow of the program.

To execute the program up to a certain line number, enter GO TIL *#line-number*. Both the executable line and its context are displayed.

Examples

1. In this example, GO executes the user program until CS:2090H or CS:3090H is encountered.

```
*GO TIL CS:2090H , CS:3090H
```

2. In this example, GO clears all active breaks and executes the user program from beginning to end.

```
*GO FOREVER
```

3. In this example, GO clears all existing temporary breakpoints and then sets a temporary breakpoint at line number 39.

```
*GO TIL #39
```

Cross-references

Encyclopedia Entries

Expressions
LOAD

Menus/Items

Go/Forever
Go/Keep Fixed
Go/Til Breakpoint
Go/Til Cursor Line

Keyboard Controls

F5 Go, all breakpoints
sF5 Go forever
F7 Go til cursor line
sF7 Go, fixed breakpoint
F9 Set temp breakpoint

HELP

Provides on-line operating assistance

Syntax

```
HELP [help-item]  
     [  
     E  
     n  
     En  
     ]
```

Where:

- help-item* specifies the topic for which help is desired. Help topics are listed below.
- E displays the expanded error message for the last error.
- n* displays the error message number *n*, with no expanded text. *n* must be a decimal number.
- En* displays the expanded error message number *n*.

Discussion

The HELP command provides on-line operating assistance by displaying helpful information in the command window on a variety of topics:

ASM	ENTRY	MODULES
BASE	EXCEPTIONS	OBJECTS
BREAKPOINT	EXIT	OPERATORS
BROWSEMENU	GO	PORT
CALLS	GOMENU	PROCEDURES
CMDWINDOW	HELP	PSTEP
COMMANDS	IF	REALS
COMMENTS	INCLUDE	REGISTERS
CONSTRUCTS	INVOCATION	REPEAT
COUNT	ISTEP	RSTEP
CSTEP	KEYBOARD	SCOPE
DEBUGMENU	LABELS	SETMENU
DEFINE	LINES	SOURCE
DEREFERENCE	LIST	STACK
DIR	LOAD	TYPES
DISPLAY	LSTEP	VARIABLES
DO	MENU	VIEWMENU
EDIT	MODIFY	WINDOWMENU

Example

In this example, HELP provides on-line information about the EXIT command.

***HELP EXIT**

To exit the debugger, type EXIT. DB86 ends the debugging session, closes all files (both the debugger's and the user's), returns all allocated memory (both the debugger's and the user's), and returns to the host operating system.

HELP (continued)

Cross-references

Encyclopedia Entry

Debugger Commands

Menu/Item

Help Menu

IF ... ENDIF

Group and conditionally
execute commands

Syntax

```
IF bexpr THEN
  {debugger_commands}
ELSE {debugger_commands}
ENDIF
```

Where:

bexpr is a Boolean expression, which evaluates to true (Least-Significant Bit (LSB) = 1) or false (LSB = 0).

debugger_commands can be any debugger commands except HELP, INCLUDE, and LOAD.

Discussion

The IF ... ENDIF construct groups one or more debugger commands in a block and executes the commands.

IF ... ELSE executes debugger commands when *bexpr* is false. When *bexpr* is false and there is no ELSE clause, no commands are executed.

IF ... ENDIF (continued)

Example

In this example, the IF ... ENDIF construct sets up and executes the following condition: if var_1 equals 12, then perform an LSTEP from the current line. Otherwise, set a breakpoint at line number 101 and begin program execution.

```
*
*var_1
0
*IF var_1 == 12 THEN
.*LSTEP
.*ELSE GO TIL #101
.*ENDIF
[Step at #57]
```

Cross-references

Encyclopedia Entries

COUNT ... ENDCOUNT
Debugger Commands
DO ... END
Expressions
REPEAT ... ENDREPEAT

INCLUDE

Executes command input
from a text file

Syntax

```
INCLUDE filename [NOLIST]
```

Where:

filename is the name of the file to be included.

NOLIST is the debugger command that suppresses the listing of the included file to the terminal.

Discussion

The INCLUDE command redirects command input from a separate text file, treating the input as part of the current listing. Make sure that INCLUDE is the last command to appear on a line.

You can nest INCLUDE commands; the level of nesting permitted is determined by the debugger memory manager.

Example

In the following example, INCLUDE includes a command file that initializes the source path for debug list files and also assigns specific list files to two of the program modules.

```
*INCLUDE exampl.inc  
*SPATH = ..\list.dir  
*SETMOD :module_a to first.lst  
*SETMOD :module_b to second.lst
```

Cross-reference

Encyclopedia Entry

LIST/NOLIST

ISTEP

Steps through a program one machine instruction at a time

Syntax

ISTEP

Discussion

The ISTEP command executes the machine instruction at the current execution point, displays the next machine instruction, and then stops.

NOTE

If your PC has a resident 8087 microprocessor, ISTEP executes two instructions instead of one when it is stepping through an instruction which either alters the segment register or ends in a wait cycle (opcode 9B). Execution of two instructions is due to interactions between the 8087 and 8086/8088 processors and their communication protocols.

Example

In this example, ISTEP steps and displays the processor registers at the next line in the program.

*ISTEP

0021:0045H 890E0E00

MOV WORD PTR 000EH,CX

Cross-references

Encyclopedia Entries

Breakpoint Commands	FB	RSTEP
CB	GO	TR
CSTEP	LSTEP	WA
ENABLE/DISABLE	PSTEP	

Menus/Items

Go/Auto Step
Go/Return Step
Go/Call Step
Go/Procedure Step
Go/Return Step
Go/Step

Keyboard Control

F8 Step Execution

Lexical Elements

Are names, numbers, strings, and special character delimiters recognized by the debugger

Discussion

Lexical elements are names, numbers, strings, and special character delimiters recognized by the debugger. These elements are defined below.

Names

Names include keywords, symbols in the user program, and debug symbols. Keywords are symbolic elements of the DB86 command language and cannot be used in any other context. See Appendix F for the list of keywords reserved for DB86. You can refer to debug symbols as simple, unqualified names.

A name must begin with any of the following:

- uppercase or lowercase letter
- at symbol (@)
- underscore (_)
- question mark (?)

A name can also be made up of any of the following:

- at symbol (@)
- underscore (_)
- question mark (?)
- dollar sign (\$)
- decimal digit

With one exception, the break character (dollar sign, \$) is not significant in identifiers and is discarded on input, as is consistent with PL/M. However, the exception is the pseudovisible \$, which represents the current program counter (PC).

Numbers

Numbers are 8-, 16-, or 32-bit binary quantities, depending on the number of significant digits. If you have an 8087 numeric processor or an 8087 emulator program loaded, you can also use real numbers, in binary, decimal, or hexadecimal radix. Real numbers are floating-point numbers in scientific notation, and can range from 8.43E-37 to 3.38E+38.

Signed numbers and real numbers are made possible by the syntax for expressions.

The base of a number or real number is set with the BASE command. To override the current base setting, enter one of the following suffixes:

- *nY* Binary
- *nT* Decimal
- *nH* Hexadecimal
- *nK* Decimal multiple of 1024

The unusual base specifiers, especially Y and T, are required because of the use of B as a hexadecimal digit. That is, it would otherwise be unclear whether 1B is a hexadecimal number or a binary number with a base override.

Strings

A string is one or more characters enclosed in apostrophes ('). You can create a string up to 254 characters long, not counting the apostrophe delimiters. You can use an apostrophe as part of a string by enclosing it in quotation marks (that is, ""). You can use both uppercase and lowercase letters in a string.

Strings that are separated by one or more logical blanks (including spaces, tabs, or returns) are concatenated to form a single string. This feature makes it possible to break strings over line boundaries.

Special Character Delimiters

The debugger recognizes the delimiters listed in Table 5-19.

Lexical Elements (continued)

Table 5-19 Special Character Delimiters

Character(s)	Function
space	logical blank
tab	logical blank
return	line terminator
ampersand ... return	continuation line indicator
semicolon (;)	command separator
apostrophe (')	string delimiter
dot (.)	infix symbol qualification delimiter
colon (:)	module name prefix
comma (,)	list element separator
caret (^)	pointer dereferencer
/* ... */	comment sequence
double quote (")	user symbol override character

Cross-reference

Encyclopedia Entry

BASE

LIST/NOLIST

Opens or closes
a debug session log file

Syntax

```
[LIST filename ]  
[NOLIST
```

Where:

filename is the name of the file in which information is to be listed.

Discussion

The LIST command creates a debug session log by opening a file and recording a history of all interactions in the command window between the debugger and the user. Output includes prompts, input line echos, and error messages.

The NOLIST command closes the current list file. There are two other ways to close the current list file:

1. Open another list file, using the LIST command;
2. Exit DB86.

Examples

1. In this example, LIST opens a debug session log file on the A drive.

```
*LIST a:aug1.88
```

2. In this example, NOLIST closes the current list file.

```
*NOLIST
```

LOAD

Reads an Intel-86, load time locatable, OMF file into memory

Syntax

```
LOAD filename [load-file-tail]
```

Where:

filename names the file to be loaded into memory.
load-file-tail specifies the command invocation controls for the program to be loaded.

Discussion

The LOAD command reads an INTEL-86, load-time-locatable, OMF file into memory, making it available to the debugger for debugging. Symbolic information is included in the file if you used the DEBUG option at compilation time.

If the source path has not been explicitly set by the user with the SPATH command, it is set to the directory from which the debugger is loaded.

Example

In this example, LOAD loads MYPROG.86 into memory; /s/m is passed to the program as a load file tail.

```
*LOAD \debug\mydir\myprog.86 /s/m
```

Cross-references

Encyclopedia Entry

SPATH

Menus/Items

Debug/Load Program
Debug/Reload
Debug/DOS Shell
Debug/Exit DB86
Debug/Source Path

LSTEP

Steps to the next high-level language line in the program

Syntax

LSTEP

Discussion

The LSTEP command executes the high-level language line at the current execution point and then stops.

You can use LSTEP even if the current execution point is not at the beginning of a source-level statement; LSTEP steps to the next statement number in the program.

NOTE

If your PC has a resident 8087 microprocessor, ISTEP executes two instructions instead of one when it is stepping through an instruction which either alters the segment register or ends in a wait cycle (opcode 9B). Execution of two instructions is due to interactions between the 8087 and 8086/8088 processors and their communication protocols.

Example

In this example, LSTEP steps at two consecutive language statements.

```
*LSTEP  
[Step at :SWAPARR#17]  
*LSTEP  
[Step at :SWAPARR#18]
```


Cross-references

Encyclopedia Entries

Breakpoint Commands	FB	RSTEP
CB	GO	TR
CSTEP	ISTEP	WA
ENABLE/DISABLE	PSTEP	

Menus/Items

Go/Auto Step
Go/Step

Keyboard Control

F8 Step execution

Memory Access

Makes it possible to display or change debug variables or program variables in user-program memory

Discussion

The debugger's memory access features enables you to display or change debug variables or program variables in user-program memory.

Displaying Variables

To display the value of a debug or program variable within the program block containing the current execution point or the block containing the variable as a public symbol, simply type the name of the variable and press Enter.

To display the address of a variable within the current program block, use the dot operator (.) as a prefix to the variable name as in *.variable*. The current program block is determined by the SCOPE pseudovariable.

To display the item the pointer is pointing to (i.e., to dereference the pointer), type *pointer-name^*. Pointer dereferencing is available only for languages that provide type information describing pointer item relationships (such as iC-86).

To display the value of a variable outside the current scope, provide a fully or partially qualified reference, which establishes a path from the module level to the desired symbol.

The syntax for a fully qualified reference is as follows:

:module-name.procedure-name.symbol-qualifier.symbol-name.

The syntax for a partially qualified reference is the same, but shorter: you can omit the outermost module and procedure specifications, depending on the current scope.

Modifying Variables

To change the value of a debugger or program variable, type the variable name, the assignment operator (=), and the new value: *variable = new-value*. *new-value* must be the same type as the original value or coercible to it. See the Expressions topic in this chapter for rules governing type coercion during assignment.

If the variable being displayed or modified is stack resident and is not active at the time of reference then <INACTIVE> is displayed.

If a variable name conflicts with a DB86 keyword (e.g., BASE), use the double-quote prefix (") to force DB86 to look up the variable in the user symbol table.

Examples

1. In this example, the debugger displays the values of three consecutive integers, starting at INTARRAY:

```
*integer .INTARRAY LENGTH 3  
+15 -47 +1024
```

2. In this example, the debugger changes the value of the user variable bob, which is defined as an integer.

```
*bob  
+215  
*bob = 8  
*bob  
+8
```

3. In this example, the debugger displays the ten bytes starting at .var.

```
*BYTE .var LENGTH 10  
1047:3333 01 23 77 00 11 22 33 44 55 66 77 88 99 AA CC
```

4. In this example, the debugger initializes an array of words to zero.

```
*WORD .word_array length 50 = 0
```

Memory Access (continued)

5. In this example, the debugger moves a block of memory from one area to another.

```
*BYTE .array1 LENGTH 20 = BYTE .array2 LENGTH 20
```

6. In this example, the debugger displays an inactive, stack-resident variable:

```
*ABC  
<INACTIVE>
```

7. In this example, you use the double-quote prefix to force DB86 to lookup the variable BASE in the user symbol table instead of the current number base:

```
**"base = 37T  
*"base  
37  
*base  
DECIMAL
```

Cross-references

Encyclopedia Entries

Expressions
Data Types
SCOPE

Menu System

Gives access, through pulldown menus, to many debugger commands

Discussion

The menu system gives access, through pulldown menus, to many often-used debugger commands. The menu system is represented on the screen by the Menu Bar across the top line of the display. You do not have to remember command syntax or even command names; just select the commands from the menus.

There are two ways to activate the menu system:

- Type Alt-M. Use the leftarrow (←) and rightrightarrow (→) keys to highlight the desired menu in the Menu Bar. Press Enter to select the highlighted pulldown menu.
- Type Alt-x, where x is the first letter of the name of a pulldown menu.

To select a menu item, use the uparrow (↑) and downarrow (↓) keys until the desired item is highlighted. Press Enter to select the highlighted menu item. Or type the first character of the desired item.

To exit the menu system, press Esc.

Cross-references

Encyclopedia Entry

Debugger Commands

Menu/Item

Alt-M or Alt-x, where x is the first letter of the name of a pulldown menu.

OBREAK

Is a pseudovvariable that determines whether or not a breakpoint occurs when a user-program overlay is loaded

Syntax

```
OBREAK = [ TRUE ]  
         [ FALSE ]
```

Discussion

The Boolean pseudovvariable OBREAK controls whether or not execution of the user program stops when an overlay load is detected. The default value is true, which means that execution breaks each time an overlay load request is performed.

When OBREAK is false, overlay loads are transparent; that is, they do not perceptibly interrupt program execution.

NOTE

The debugger's overlay loading function simulates the UDI DQ\$OVERLAY function. You must use the NOGROUPOVERLAYS (NOGO) option of LINK86 when you are linking user programs that contain overlays and are to be debugged using DB86.

Example

In this example, OBREAK is set to false.

```
*OBREAK = FALSE  
*OBREAK  
FALSE
```

Cross-reference

Encyclopedia Entry

Pseudovvariables

PORT

Is a pseudovvariable that displays or changes the contents of a specified byte wide I/O port

Syntax

PORT (*port-number*) [= *data*]

Where:

PORT (*port-number*) displays the contents of the user I/O port specified by *port-number* in the current base. *port-number* is a number or expression that evaluates to a number in the current base from 0000H to 0FFFFH.

data is any byte of data entered in the current base. *data* is written to the specified port.

Discussion

The pseudovvariable PORT displays the contents of a specified byte-wide I/O port. When used with the assignment operator (=), PORT changes the contents of a port.

When used as a pseudovvariable in a debugger expression, PORT represents the current value of the byte port.

If you try to write data longer than a byte (e.g., a word), PORT uses only the least-significant byte. Use the WPORT pseudovvariable for word length data.

PORT (continued)

Examples

1. In this example, PORT reads I/O port number 2. The base is hexadecimal.

```
*PORT(2)
99
```

2. In this example, PORT writes a decimal value to I/O port number 2. The T base selector overrides the current base and designates 50 as a decimal number.

```
*PORT(2) = 50T
*PORT(2)
32
```

Cross-references

Encyclopedia Entries

Pseudovariables
WPORT

Pseudovariables

Affect the operation
of the debugger

Discussion

The pseudovariables OBREAK, SYSINT, PORT and WPORT affect the operation of the debugger. OBREAK controls how the debugger acts when an overlay load request is performed. SYSINT controls whether or not the debugger monitors certain DOS interrupts during program execution. PORT and WPORT give access to the contents of byte wide and word wide I/O ports. To display the value of one of these pseudovariables, just type *pseudo-var*, where *pseudo-var* is OBREAK, PORT, SYSINT, or WPORT.

The pseudovariable SCOPE points to the current execution point or changes it to your specification.

Cross-references

Encyclopedia Entries

BASE	PORT
Data Types	SCOPE
Debug Variables	SYSINT
OBREAK	WPORT

PSTEP

Steps to the next high-level language line in the program, stepping over procedure calls

Syntax

PSTEP

Discussion

If the current execution point resides at a CALL instruction, PSTEP executes the called procedure and stops upon return from it. PSTEP treats the procedure or function as if it were a single statement, executing it completely before returning control at the next statement. The next statement to be executed is displayed.

NOTE

If your PC has a resident 8087 microprocessor, ISTEP executes two instructions instead of one when it is stepping through an instruction which either alters the segment register or ends in a wait cycle (opcode 9B). Execution of two instructions is due to interactions between the 8087 and 8086/8088 processors and their communication protocols.

Example

In this example, PSTEP steps through a call to a procedure. Before PSTEP is executed, the current CS:IP (\$) is at line number 40 in the module myprog. Line 40 consists of a call to a procedure. When PSTEP is executed, the entire procedure is executed.

***PSTEP**

[Step at :myprog #41]

Cross-references

Encyclopedia Entries

CSTEP
GO
ISTEP
LSTEP
RSTEP

Menu/Item

Go/Procedure Step

Keyboard Control

F10 Step over procedure

Registers and Flags

Set the contents of a specified register or flag; display the contents of all registers or flags or a specified register or flag

Discussion

Registers and flags set the contents of a specified register or flag. They also display the contents of all registers or flags or a specified register or flag. Registers and flags are displayed in hexadecimal.

Cross-references

Encyclopedia Entries

Expressions
FLAGS
REGS
REGS87

Menu System

Window/Registers

Keyboard Control

F2 Toggle Register Window

REGS

Displays or sets 8086/8088
microprocessor registers

Syntax

```
[REGS  
  86/88_register-name [= expr]  
]
```

Where:

expr is an expression that sets the contents of the specified 8086/8088 register.

86/88_register-name displays the contents of one of the following 8086/8088 registers, in hexadecimal format:

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

Discussion

The REGS command displays the contents of all registers, or the contents of a specified 8086/8088 register, during a particular program state. Definitions of 8086/8088 registers are shown in Table 5-20.

REGS (continued)

Table 5-20 8086/8088 Registers

Keyword	Description	Data Type
Data Registers		
AX	Accumulator register pair	WORD
AH	Accumulator high byte	BYTE
AL	Accumulator low byte	BYTE
BX	B register pair	WORD
BH	B register high byte	BYTE
BL	B register low byte	BYTE
CX	C register pair	WORD
CH	C register high byte	BYTE
CL	C register low byte	BYTE
DX	D register pair	WORD
DH	D register high byte	BYTE
DL	D register low byte	BYTE

Table 5-20 8086/8088 Registers (continued)

Keyword	Description	Data Type
Pointer/Index Registers		
SI	Source index	WORD
DI	Destination index	WORD
BP	Base pointer	WORD
SP	Stack pointer	WORD
Segment Registers		
CS	Code segment	WORD
DS	Data segment	WORD
ES	Extra segment	WORD
SS	Stack segment	WORD
Instruction Pointer		
IP	Instruction pointer	WORD

REGS (continued)

Examples

1. In this example, *register-name* AX sets the contents to 12C0 and displays the new value.

```
*AX = 12C0
```

```
*AX
```

```
12C0
```

2. In this example, REGS displays the current contents of all registers.

```
*REGS
```

```
AX=0004H    BX=003AH    CX=0000H    DX=0002H  
SI=39E7H    DI49F2H    BP0104H    SP=49F2H  
SS=39E7H    DS=49F2H    ES=0104H    CS=49F2H  
IP=03A2H  
FLAGS:ZFL OFL PFL
```

Cross-references

Encyclopedia Entries

Expressions
FLAGS
Registers and Flags
REGS87

Menu/Item

Window/Registers

Keyboard Control

F2 Toggle Register Window

REGS87

Displays or sets 8087 emulator or
numerics processor registers

Syntax

```
[REGS87  
  87_register-name [= expr ]
```

Where:

expr is an expression that sets the contents of the specified 80x87 math coprocessor unit.

87_register-name displays the contents of one of the following 8087 registers, in hexadecimal format:

ST0	ST7
ST1	FIO
ST2	FCW
ST3	FSW
ST4	FTW
ST5	FIA
ST6	FDA

Discussion

The REGS87 command displays the contents of all registers, or the contents of a specified 8087 register, during a particular program state. The 8087 coprocessor may be either the 8087 emulator or an 8087 numeric processor; the debugger automatically recognizes the coprocessor type at load time. Definitions of 8087 registers are shown in Table 5-21.

NOTE

If the program being debugged does not use real math, the contents of the 8087 registers are undefined. In this instance, the debugger prevents you from displaying or setting 8087 registers.

REGS87 (continued)**Table 5-21 8087 Registers**

Keyword	Description	Data Type
ST0	Stack Register 0	Tempreal
ST1	Stack Register 1	Tempreal
ST2	Stack Register 2	Tempreal
ST3	Stack Register 3	Tempreal
ST4	Stack Register 4	Tempreal
ST5	Stack Register 5	Tempreal
ST6	Stack Register 6	Tempreal
ST7	Stack Register 7	Tempreal
FIO	Instruction Opcode	Word
FCW	Control Word	Word
FSW	Status Word	Word
FTW	Tag Word	Word
FIA	Instruction Address	Dword
FDA	Data Address	Dword

Example

In this example, REGS87 displays the current contents of all registers (assuming that a floating-point user program is loaded).

```
*REGS87
F10  :0624          FCW  :037F
FSW  :0000          FTW  :FFFF
FIA  :0006E229     FDA  :00000D62
ST0  :+3.1415926535000000 ST1 :+2.0000000000000000
ST2  :+0.0000000000000000 ST3 :+3.0000000000000000
ST4  :+4.0000000000000000 ST5 :+3.0000000000000000
ST6  :+2.0000000000000000 ST7 :+1.0000000000000000
```

Cross-references**Encyclopedia Entries**

- Expressions
- FLAGS
- Registers and Flags
- REGS

Menu/Item

- Window/Registers

Keyboard Control

- F2 Toggle Register Window

REPEAT ... ENDREPEAT

Groups and executes commands forever, while, or until an exit condition is met

Syntax

```
REPEAT
  {debugger_commands
  WHILE bexpr
  UNTIL bexpr}
ENDREPEAT
```

Where:

<i>debugger_commands</i>	can be any debugger commands except HELP, INCLUDE, and LOAD.
<i>bexpr</i>	is a Boolean expression that evaluates to true (Least-Significant Bit (LSB) = 1) or false (LSB = 0).
WHILE <i>bexpr</i>	executes while <i>bexpr</i> is true. Execution stops when the WHILE <i>bexpr</i> is false.
UNTIL <i>bexpr</i>	executes until <i>bexpr</i> is true.

Discussion

The REPEAT ... ENDREPEAT construct repeatedly executes a block of commands until the specified exit condition is reached.

Example

In this example, REPEAT ... ENDREPEAT line steps the program until line number 50 is reached.

```
*REPEAT
.*LSTEP
.*UNTIL $ == #50
.*ENDREPEAT
```

Cross-references

Encyclopedia Entries

COUNT ... ENDCOUNT
Debugger Commands
DO ... END
Expressions
IF ... ENDIF

RSTEP

Steps out of the current procedure until after the next return encountered, stepping over calls

Syntax

RSTEP

Discussion

The RSTEP command executes the user program from the current execution point until one step beyond the next RETURN instruction encountered. RSTEP steps through any intervening CALL instructions.

RSTEP provides the ability to return execution context to the procedure that called the current procedure. RSTEP is implemented as a series of ISTEP commands; therefore, execution is not at full machine speed.

Example

In this example, RSTEP returns control to the procedure that called the current procedure.

```
*CALLS  
[0] :MOD #27 CO  
(1) :MOD #41  
*RSTEP  
[Step at :MOD #41]
```

Cross-references

Encyclopedia Entries

CSTEP
GO
ISTEP
LSTEP
PSTEP

Menu/Item

Go/Return Step

Keyboard Control

sF10 Return step

SCOPE

Sets a new debugger browse context or displays the current one

Syntax

SCOPE [= *aexpr*]

Where:

aexpr is an expression that evaluates to an address.

Discussion

The pseudovariable SCOPE defines the current scope context. The scope context is a memory location in the user program that is known to the debugger and that determines what user symbols and variables are visible to the debugger. It also determines what source information is shown in the View Window. After any steps or breakpoints, the scope is automatically set to the current execution point (CS:IP). When the debugger is in control, the user is free to change the scope to another location in the program (i.e., browsing).

Use SCOPE = *aexpr* to change the current scope context and display a window of data starting at that point.

You can use the scope context to help qualify a symbol in the user program. The current scope determines how much information you must provide when you are looking up a user-program symbol. The debugger searches for the symbol first in the innermost program block, and then in the next (ancestor) block, continuing the search until the symbol is found or the search fails in the outermost block.

For example, suppose execution has halted in the procedure GETCHAR in the module SCANNER. You can refer to any variable in GETCHAR just by typing its name. But suppose execution has halted outside GETCHAR. To refer to a variable within GETCHAR, you must also supply a qualifier (e.g., GETCHAR.TOKEN). To refer to a variable in another module, you must supply the module name and any enclosing procedure names; use a dot to separate procedure names from each other (e.g., :SCANNER.GETSTR.GETCHAR.CH).

SCOPE (continued)

Even if the current execution point is not inside of GETCHAR, you can set SCOPE to a point inside GETCHAR and, thus, allow direct reference to variables like CH, which are visible only inside of GETCHAR. SCOPE can easily be set back home (i.e., to CS:IP) by the command SCOPE = \$.

NOTE

You can use SCOPE interchangeably with its alias, NAMESCOPE.

Example

In this example, SCOPE locates var_1, which is known in MOD_2 but not in MOD_1.

```
*SCOPE = :MOD_1
*var_1
VAR_1
ERROR #12
Symbol not known in current context.
*SCOPE = :MOD_2
*var_1
15
```

Cross-references

Encyclopedia Entries

- Pseudovariables
- User-program Symbols

Menu/Item

- Browse/Scope ...

Keyboard Control

- Grey *

SETMOD

Correlates an executable module with a source display file

Syntax

```
SETMOD :module-name TO filename
```

Where:

module-name is the name of the module for which debug information is obtained and stored in the virtual symbol table.

filename is the name of the file used for source display. It may be a listing file generated during compilation or it may be the source file.

Discussion

The SETMOD command correlates the name of an executable module with the name of a corresponding source or listing file. The debugger uses this file while displaying the user program in the View Window.

The source display file is derived as follows:

1. The debugger looks up, in the virtual symbol table, the name of the module being debugged or the name of the module you have specified.
2. The debugger checks to see if you set the module with SETMOD.
3. If you set the module with SETMOD, the debugger derives the full path name from the filename plus the SPATH setting.
4. If you have not set the module, SPATH is prefixed to the module name and the first extension specified by SPATH is appended to the module name.
5. The debugger keeps trying each extension supplied by SPATH until a file is successfully opened or the SPATH extension list is exhausted.

Since the default extension in the SPATH extension list is .LST, the default module name is *module-name*.LST.

Examples

In this example, DIR displays the current setting for program module ecription, SETMOD assigns FILE1.C to ECRYPTION, and DIR confirms the new setting.

```
*DIR :ecryption SETMOD
ECRYPTION \TEST.DIR\ECRYPTIO.LST
*SETMOD :ecryption TO file1.c
*DIR :ecryption SETMOD
ECRYPTION \TEST.DIR\FILE1.C
```

2. In this example, SETMOD assigns FORTRAN.LST to the program MYPROG.

```
*SETMOD myprog TO fortran.lst
```

Cross-references

Encyclopedia Entries

DIR
SPATH

Menu/Item

Debug/Source Path

SPATH

Displays or sets the path and default extensions used to find the source display files of compiled modules

Syntax

```
SPATH [= pathname [,ext0, ... ext9]]
```

Where:

pathname specifies the relative or absolute path to the directory containing source display files in the format `\directory-name\directory-name`. *directory-name* can also be one of the DOS relative directories (. and ..).

ext is the extension or extension list for *pathname*. You can list up to ten extensions.

Discussion

The SPATH command displays the current path to the directory containing source display files for compiled and linked modules. The source display filenames are not part of the path name. The default path is the path from which the user program was loaded and the default extension list is .LST.

To set a new path, use `SPATH = pathname`.

If you have not explicitly assigned a file to a module with SETMOD, the debugger attempts to apply a module name as a prefix and each list extension as a suffix until a file is found or the extension list is exhausted.

The debugger can have only one path at a time. Therefore, we recommend that, before beginning the debug process, you place all related source display files in a single directory and use SPATH to set the path to that directory. The grouping of source display files by directory ensures that the debugger has source list text information for all the necessary source display files.

Examples

1. In this example, SPATH sets the source path to `..\..\prog\display`.

```
*SPATH = ..\..\prog\display
```

2. In this example, SPATH sets the directory to `\prog\display\window`, and also initializes the suffix search list to `.C,.LST`.

```
*SPATH = \prog\display\window, .C,.LST
```

3. In this example, SPATH displays the current path and extension list.

```
*SPATH  
\PROG\IO_DISPLAY\WINDOW  
Extensions: .C .LST
```

Cross-references

Encyclopedia Entries

LOAD
SETMOD

Menu/Item

Debug/Source Path

STACK

Displays values from the top of the stack

Syntax

```
STACK [expr]
```

Where:

expr is an expression that specifies the number of values to be displayed.

Discussion

The STACK command displays either the first value or the specified number of values from the top of the stack.

Example

In this example, STACK displays the top 16 values on the stack, in hexadecimal format.

```
*BASE = HEX
*STACK 10
01A0 01A4 00AF 5DD2 50CB C033 5050 FF50
14AF 005C 14B7 005C 01A4 14BF 005C 14C7
```

Cross-reference

Encyclopedia Entry

Expressions

Stepping Commands

Step program execution by called procedure, machine instruction, language instruction, procedure, or return instruction

Discussion

Stepping commands provide control over program execution. You can step into the next called procedure (CSTEP), by machine instruction (ISTEP), by line instruction (LSTEP), through a procedure (PSTEP), or out of a procedure (RSTEP).

Cross-references

Encyclopedia Entries

CSTEP
ISTEP
LSTEP
PSTEP
RSTEP

Menu/Item

Go Menu

Keyboard Controls

F8 Execute one step of user program
sF8 Step into next procedure call encountered
F10 Step over procedure calls to next line
sF10 Step out of current procedure

SYSINT

Is a pseudovvariable that determines whether or not a breakpoint occurs when a DOS interrupt is encountered during program execution

Syntax

SYSINT = [TRUE]
 [FALSE]

Discussion

The pseudovvariable SYSINT determines whether or not a breakpoint occurs when a DOS interrupt is encountered during program execution. When SYSINT is set to the default, true, the debugger monitors the following interrupts:

- INT8 DOS timer tick (18.2/sec) enables the debugger to monitor for a SYSREQ break during program execution
- INT15 SYSREQ handler intercepts depression of the SysReq key, which posts a request for program execution to cease
- INT21 Function 0, Function 4C, DOS Exit command is used by C run-time libraries to terminate program execution. The exit is intercepted by the debugger
- INT21 Function 62 Get PSP Address simulates the program PSP when the user program is running under C run time libraries.

NOTE

Set SYSINT to false only when absolutely necessary; the false setting provides no way to trap the user-program, exit request or to solicit asynchronous break requests with SysReq. Note that trapped interrupt vectors are modified only when the user program is active. The debugger restores interrupt vectors to their default values when execution of the user program is suspended.

Example

In this example, SYSINT changes its value from true to false.

```
*SYSINT  
TRUE  
*SYSINT = FALSE  
*SYSINT  
FALSE
```

Cross-references

Encyclopedia Entries

OBREAK
Pseudovariables

TR

Sets a new tracepoint
or displays a current tracepoint

Syntax

```
TR[n] [= aexpr [expr-list]]
```

Where:

- n* is a number, from 0 to 9 inclusive, assigned to a debugger tracepoint.
- aexpr* is an expression representing the address at which program execution is to be monitored.
- expr-list* is an expression list that is evaluated and displayed at the trace address.

Discussion

The tracepoint command, TR, sets a tracepoint at the specified address, for the purpose of reporting the address at which the program is executing. Program execution resumes after the report. Up to 10 tracepoints are available: TR0, TR1, TR2, TR3, TR4, TR5, TR6, TR7, TR8, and TR9.

There may be a number of comma-separated expressions in *expr-list*. Examples of expressions are variable names, strings enclosed in single quotes, and Boolean expressions enclosed in parentheses.

You can enable tracepoints with the ENABLE command and disable them with the DISABLE command.

Use the ENABLE and DISABLE commands to selectively activate tracepoints (e.g., ENABLE/DISABLE TR*n*) or to activate all tracepoints (ENABLE/DISABLE TR).

To display a list of expressions with the tracepoint, specify the list of expressions.

To show the current definition of a tracepoint, enter TR*n*.

NOTE

Expressions in the Watch Window are updated when breakpoints are encountered in the executing user program.

Examples

1. In this example, TR0 displays its current definition, which is to report when program execution reaches `mymod.myproc`.

```
*TR0 = :mymod.myproc
```

2. In this example, a breakpoint is defined to report the values of `x` and `y` when program execution reaches line number 50.

```
*TR1 = #50, 'The value of x is:',x,'The value of y is:',&
**y,'x equals y:', (x == y)
```

Cross-references

Encyclopedia Entries

Breakpoint Commands	FB
CB	GO
DIR	TR
ENABLE/DISABLE	WA

Menu/Item

Set/Tracepoint

Keyboard Control

sF9 Set tracepoint

User-program Symbols

Are accessible for display

Discussion

User-program symbols and their addresses are accessible for display.

To display the value of a user-program, symbol, type *symbol-name*.
If the symbol name conflicts with a debugger keyword, use the double quote (") prefix operator.

To display the address of a user-program symbol, type *.symbol-name* (prefixing the dot operator [.]) Depending on the current scope, you may need to give more information.

To change a user-program symbol, type *symbol-name = value*.
Assigned values are automatically coerced to the user symbol type.

Cross-references

Encyclopedia Entries

Debug Variables
Lexical Elements
SCOPE

WA

Sets a new watch expression
or displays a current watch expression

Syntax

WA[*n*] [= *expr*]

Where:

n is a number, from 0 to 5 inclusive, assigned to a debugger watch expression.

expr is an expression.

Discussion

The watch expression command, **WA**, allows you to view program variables or memory activity in the Watch Window at program execution points. Watch expressions can be memory template commands, references to symbolic variables, or, when enclosed in parentheses, a complex series of references that evaluate to a unique expression.

Watch expressions are updated as follows:

- At every program step when used with **CSTEP**, **ISTEP**, **LSTEP**, **PSTEP**, or **RSTEP**
- At program breakpoints
- At program tracepoints
- At each execution step when **DB86** is running in Auto Step mode

An advantage to using program tracepoints for updating is that the user program can continue running with periodic tracepoints providing the watch update function.

Up to six watch expressions are available: **WA0**, **WA1**, **WA2**, **WA3**, **WA4**, and **WA5**.

You can enable watch expressions with the **ENABLE** command and disable them with the **DISABLE** command.

WA (continued)

Use the ENABLE and DISABLE commands to selectively activate watch expressions (e.g., ENABLE/DISABLE *WAn*) or to activate all watch expressions (ENABLE/DISABLE WA).

To show the current definition of a watch expression, enter *WAn*.

Examples

1. In this example, WA0 sets up a watch of `count_var`.

```
*WA0 = count_var
```

2. In this example, WA1 sets up a watch of the value of a Boolean expression. The expression evaluates to true when `xy` equals 5 or when `mtop` is less than 25 hexadecimal. Otherwise, it is false.

```
*WA1 = (xy == 5) OR (mtop < 25H)
```

3. In this example, WA2 sets up a watch on a pointer at location 0000:0040.

```
*WA2 = pointer 0000:0040
```

Cross-references

Encyclopedia Entries

Breakpoint Commands	ENABLE/DISABLE	LSTEP
CB	FB	PSTEP
CSTEP	GO	TR
DIR	ISTEP	

Menu/Item

Window/Watch Window

Keyboard Control

sF2 Toggle Watch Window

WPORT

Is a pseudovvariable that displays or changes the contents of word-wide ports

Syntax

WPORT(*port-number*) [= *data*]

Where:

port-number displays the contents of the specified word-wide, I/O port in the current base. *port-number* is a number or expression that specifies one of the I/O ports in the range 0000H to 0FFFFH.

data writes a word of data to the specified port.

Discussion

The pseudovvariable WPORT displays or changes the contents of a specified word-wide, I/O port. There is no protection against writing a read-only control register. The debugger displays the output word in hexadecimal.

When used as a pseudovvariable in a debugger expression, WPORT represents the current value of the word port.

Examples

1. In this example, WPORT reads a word-wide port.

```
*WPORT(023H)
0A12
```

2. In this example, WPORT writes a word-wide port.

```
*WPORT(0123H) = 5556
*WPORT(0123H)
5556
```

WPORT (continued)

Cross-references

Encyclopedia Entries

PORT
Pseudovariables

Contents

Appendixes

Appendix A DB86 Installation

A.1	Installing the DB86 Debugger Software.....	A-1
A.2	Modifying the System Configuration.....	A-3
A.3	Setting Environment Variables.....	A-3
A.4	Keyboard Templates	A-4

Appendix B DB86 Invocation

B.1	Syntax.....	B-1
B.2	Controls.....	B-1
B.3	Examples.....	B-4

Appendix C Shortcuts and Tips

C.1	Setting up a Second Monitor.....	C-1
C.2	Using RAM Disk for the Virtual Symbol Table Buffer.....	C-2
C.3	Shortcut for Setting SPATH.....	C-3
C.4	Hints on Naming Modules.....	C-3
C.5	Using the Debug Control in ASM86 Programs.....	C-4
C.6	Debugging Programs with Overlays.....	C-4
C.7	Using the SWAP Control.....	C-4
C.8	Tips on Variable Names and Reserved Words.....	C-5
C.9	Tips on Clearing Breakpoints.....	C-5

Appendix D Language Support

D.1	iC-86.....	D-1
D.2	PL/M-86.....	D-1
D.3	ASM86.....	D-2
D.4	Fortran-86.....	D-2
D.5	Pascal-86.....	D-2

Appendix E Error Messages

Appendix F Reserved Words

Appendix G ASCII Codes



This appendix is a guide to installing the DB86 debugger on DOS systems.

A.1 Installing the DB86 Debugger Software

The following hardware and software is required to run the DB86 debugger on DOS systems:

- Hardware: IBM PC XT, IBM PC AT, and IBM PS/2 or fully equivalent system
- Operating system: DOS V3.0 or later or OS/2 running in compatibility mode
- System memory requirements: 210K bytes RAM plus memory required by the DOS, user-application program

An 8087 numeric coprocessor is not required in the user system. Floating-point math can be accomplished via the 8087 emulator that accompanies the user program being debugged. If an 8087 coprocessor is present, the debugger can use it. The debugger uses the same floating-point support that the program being debugged uses.

Make a backup copy of the product disk(s) before installation, using the DOS DISKCOPY program.

To avoid name conflicts, create a directory for the DB86 debugger.

If your version of DB86 comes on 5.25-inch disks, DB86 and help and error files are on the disk labeled Disk 1 of 2, and the EXAMPLE program is on the disk labeled Disk 2 of 2. If your version of DB86 comes on a 3.5-inch disk, all files are on that disk.

To install the software, perform the following steps:

1. Change to the directory you have created for the DB86 debugger.
2. Copy all files from your single 3.5-inch disk to the current directory with the COPY command. Or, copy all files from each 5.25-inch product disk. Assuming that the product disk is in drive A and the current drive is drive C, enter:

```
C>copy a:\*.*
```

There are three files on the product disk (Disk 1 of 2 if you have a 5.25-inch disk) that you must copy to run DB86:

```
DB86.EXE  
DB86.OVE  
DB86.OVH
```

Also, you can copy the following EXAMPLE program files (on Disk 2 of 2 if you have a 5.25-inch disk):

```
EXAMPLE.EXE  
EXAMPLE.86  
EXAMPIO.C  
MENU.C  
INFO.C  
BLIMP.C  
TRYIT.BAT  
EXAMPLE.MAC  
README  
TOOLPACK.EXE
```

See Section 1.4 and Chapter 2 for further instructions on trying the debugger with the EXAMPLE programs.

Be sure to read the README text file and Release Notes for further information on the particular release of DB86 that you have.

A.2 Modifying the System Configuration

Before you use the DB86 debugger, you must create or modify the system configuration file CONFIG.SYS so it includes the FILES and BUFFERS commands.

The FILES command specifies the maximum number of files that can be opened at the same time. The BUFFERS command specifies the number of disk buffers allocated in memory.

Set the value of FILES to 12 (or greater). Set the value of BUFFERS to 10 (or greater).

If there is no CONFIG.SYS file on your fixed disk, follow these steps to create the necessary commands:

1. Type:

```
C>copy con \config.sys <Enter>
```

2. Enter the commands:

```
FILES = 12 <Enter>  
BUFFERS = 10 <Enter>
```

3. Save the file: press the F6 key or <CTRL>-Z and then press <Enter>.
4. Reboot the system.

If the CONFIG.SYS file already exists, use an editor to add or modify the needed commands in the existing file. Include the commands FILES = 12 and BUFFERS = 10.

A.3 Setting Environment Variables

The debugger uses the value of the :WORK: environment variable as a default pathname for its virtual symbol table buffer. If you do not specify a value for :WORK:, the debugger uses the root directory on drive C: for the virtual symbol table buffer. To specify a different path, use the DOS SET command to assign a value to :WORK:.

For example, the following DOS command assigns :WORK: to a RAM disk, drive E.

```
C>set :WORK:=E:
```

You can place the SET command in the file AUTOEXEC.BAT if you wish. Then, :WORK: is automatically defined each time you reboot. See your DOS user's guide for further information.

A.4 Keyboard Templates

Two keyboard templates come with the debugger. These templates provide quick reference information on the debugger keyboard controls and other quick reference information. (See Chapter 4 for more information on the keyboard controls.)

The templates come with an adhesive backing and can be permanently attached to the keyboard by removing the protective paper from the back of the template before applying it to the keyboard surface. If you do not wish to attach them with the adhesive, do not remove the protective paper backing.

One template is designed for keyboards with function keys on the left side. The other is designed for keyboards with function keys across the top; the top template can also be used with keyboards that have function keys on the side.

This appendix describes the DOS command line used to invoke DB86.

B.1 Syntax

DB86 is invoked with a DOS command line in the following form:

```
db86 [controls] [load-file [load-file-tail]]
```

Where:

- | | |
|-----------------------|---|
| <i>controls</i> | can be any of the valid DB86 controls: MACRO, NOMACRO, VSTBUFFER, SWAP, NOSWAP, NOMB, CL2, MON2, or L43. |
| <i>load-file</i> | is the name of the file containing the user program to be loaded for debugging. It can include a drive and pathname as part of the specification. |
| <i>load-file-tail</i> | is the command tail used by the load file. For example, it can consist of filenames or controls used by program to be debugged. |

B.2 Controls

DB86 *controls* must appear on the command line before any *load-file* is specified. The following *controls* can be typed on the command line:

- | | |
|--------------------------|--|
| MACRO(<i>filename</i>) | specifies a file containing DB86 commands to be executed during initialization. |
| | If a macro <i>filename</i> is specified, it can include a drive and path as part of the specification; enclose the <i>filename</i> in parentheses. |

If neither MACRO nor NOMACRO is specified, DB86 looks for DB86.MAC, but still runs even if DB86.MAC is not present.

If MACRO is specified, DB86 lists the commands in the macro file as it executes them.

The macro file can be used to automatically set up the debugging environment at the start of the session.

Abbreviation: MR

NOMACRO

prevents DB86 from trying to find a macro file to be executed during initialization. This control is useful to prevent execution of DB86.MAC even if it is present.

Abbreviation: NOMR

VSTBUFFER(*size*)

specifies the size of the virtual symbol table buffer in memory. Performance can be improved by increasing the size of the symbol table buffer. *size* is the buffer size in K bytes, expressed as a positive integer. The minimum size is 6K bytes; the maximum is 61K bytes. If a *size* less than 6 is specified, 6K bytes are used; if a *size* greater than 61 is specified, 61K bytes are used. If this control is not specified, DB86 uses a default size of 6K bytes.

Abbreviation: VSTB

SWAP

uses screen image swapping, which requires an additional 8K bytes of memory. SWAP is the default for monochrome monitors and need not be specified for the MDA video adapter. For color video adapters, DB86 normally flips between the user-program screen and the DB86 screen using alternate video pages. In this case, DB86 assigns video page 0 for the user-program screen and video page 3 for the DB86 screen. Video page flipping is used only if the DOS video mode is 2 or 3 on a CGA, EGA, or VGA video adapter. Use SWAP to override this page flip default and force the use of screen image

swapping on color adapters. The SWAP control also assigns video page 0 for both the user-program screen and the debugger control screen.

NOSWAP

disables screen image swapping. With NOSWAP, the user program overwrites the DB86 display screen. If DB86 is running with a monochrome video adapter (MDA), screen flipping is accomplished by image swapping instead of writing to alternate video pages. Use NOSWAP if you want to disable swapping on monochrome adapters. Note that, if you do so, you effectively disable screen flipping and all your user-program output will overwrite the DB86 display screen.

NOMB

runs DB86 with no Menu Bar present on the screen. You can still access the menu system and temporarily display the Menu Bar while you are in the menu system. Once DB86 is running, you can toggle the Menu Bar on and off with the Menu Bar On/Off item in the Window Menu.

CL2

uses monochrome video attributes regardless of the video adapter present. CL2 is intended for monochrome displays that map color attributes into grey scale. If the grey scale colors do not show up very well on the display screen, use CL2 to force DB86 to use monochrome attributes, regardless of the type of video adapter present. CL2 attributes are automatically enabled if DOS video mode 2 or 7 is already in effect during DB86 initialization.

MON2

runs DB86 using two video monitors. For example, you can use a monochrome adapter (MDA) for the DB86 display screen and show the user-program output at the same time on a color adapter (CGA, EGA, or VGA). Both video adapters must be present in the host DOS computer for MON2 to work. If DOS is using the color adapter, MON2 directs DB86 to use the monochrome adapter for its display screen. If

DOS is using the monochrome adapter, MON2 directs DB86 to use the color adapter for its display screen. The user program uses the same monitor that DOS uses. See Appendix C for tips on setting up a second monitor.

L43

runs DB86 in 43-line mode if an EGA or VGA adapter is present. If DOS is already set to use the 43-line mode, DB86 automatically uses 43-line mode. In 43-line mode, DB86 uses 43 lines for both its display screen and for the user-program output screen.

NOTE

An unrecognized control is interpreted as the name of the user program to be debugged; any text following the program name is treated as the user-program, command tail.

B.3 Examples

1. In this example, DB86 is run with a virtual symbol table buffer of 40K bytes. During initialization, DB86 executes the macro file TEST.MAC. DB86 uses two video monitors and loads the user program MYTEST.86 with the command tail /A /B preserved for the user program:

```
C>db86 macro(test.mac) vstb(40) mon2 mytest.86 /A /B
```

2. In this example, DB86 is run with the default virtual symbol table buffer of 6K bytes; uses 43-line mode on the EGA video adapter; runs with no Menu Bar present; and does not look for a macro file to include. The program MYPROG.86 is loaded during initialization.

```
C>db86 143 nomb nomr myprog.86
```

This appendix describes techniques you can employ to more effectively use the debugger. Many of the tips mentioned assume a familiarity with the debugger and contain advanced techniques to make the debugger run faster and better.

C.1 Setting Up a Second Monitor

A host, DOS-compatible PC can have two video adapter boards, each with its own monitor attached. In this configuration, the host PC provides the memory used by both video adapters.

The debugger can use both monitors, if the host PC has them. By using two monitors, you can see the debugger display screen on one monitor at the same time that you see the user-program screen on the other monitor. This configuration is particularly valuable when you are debugging a program that does a lot of screen output and keyboard input. Without two monitors, you would have to flip the screen between the debugger display and the user-program display.

To setup a host PC with two monitors, follow these steps:

1. Install the second video adapter with its monitor on the host PC. Consult the appropriate system manuals for installing a second monitor under DOS.
2. Once installed, attach the monitor to the video adapter board. Consult your system documentation for more information.
3. Run the debugger with the MON2 control:

```
C:\DB86>db86 mon2
```

The MON2 control requires that both monitors are present. Both video adapter boards must be installed on the bus; the host PC memory for each video adapter board must be present; and a monitor must be attached to each video adapter. If the debugger detects only one video adapter present on the system, it ignores the MON2 control as if you had not typed it in.

Screen flipping is automatically disabled by the debugger when the MON2 control is used, so the F4 and sF4 keys are irrelevant. Swapping is not used by the debugger when the MON2 control is specified, so the SWAP and NOSWAP controls are ignored.

When you install the two video adapters in your system, one is designated as the primary video adapter while the other one is the secondary adapter. The primary adapter is the default adapter that the operating system uses when it first initializes and the one you enter DOS commands on.

The secondary adapter is used for the debugger screen while the primary adapter is for the user-program screen.

Typically, one of the two monitors is set up as VGA, EGA, or CGA color video, while the other one is set up as MDA monochrome video. In this configuration, the VGA, EGA, or CGA adapter is usually set up as the primary adapter and thus serves as the user-program screen, while the MDA adapter is used for the debugger screen. This setup is typical because it provides a richer set of graphics modes for the user program. However, if the primary adapter is the MDA, the user program appears on it while the debugger appears on the secondary (VGA, EGA, or CGA) monitor.

On a two-monitor system, be sure that the user program does not change its video mode to the adapter being used by the debugger. Changing to the mode being used by the debugger causes a direct conflict; the user-program screen display overwrites the debugger screen display and vice versa.

C.2 Using RAM Disk for the Virtual Symbol Table Buffer

By setting the DOS environment variable named :WORK:, to a RAM disk, you can speed the performance of the debugger.

See Appendix A for more information on setting this environment variable.

You can also use a disk cache program to speed hard disk performance when the debugger is accessing source files.

C.3 Shortcut for Setting SPATH

When the debugger is invoked, it implicitly sets the SPATH variable to the subdirectory specified for the load file on the command line. Thus, if your command line looks like the following, SPATH is set to the directory CSRC. The debugger looks in that directory to find the source files and you do not have to explicitly issue the SPATH command.

```
C:\DB86>db86 c:\csrc\myprog.86
```

C.4 Hints on Naming Modules

The debugger uses the module name from the OMF file to generate candidate source filenames by adding the extensions from SPATH.

With C programs, the source filename is used for the module name. Thus, there is no possibility that the module name and source filename can ever be different.

With other compilers, however, you can specify a module name within the source code. This module name is passed on to the OMF file. Thus, the source filename may differ from the module name.

For instance, with a PL/M program, the module name must be specified as a label on the outermost DO...END block. This module name is preserved in the OMF file and is used by the debugger. If the module name is BLIMP\$PROGRAM while the source file is named BLIMP.P86, the debugger first truncates the module name to the first eight characters that are legal, DOS filename characters and then tries, unsuccessfully, to find a source file with that name. In this example, the \$ character is not a valid filename character, so it is dropped leaving BLIMPPRO as the filename that cannot be found.

As another example, assume that the module is named BLIMP and the listing file is called BL.LST. In this case, the debugger will not find the source since the listing filename is not the same as the module name.

Be sure that you use the same name for the module name, the source filename, and the list filename if you specify any of these names during compilation. Or, use the SETMOD command to explicitly correlate the module name with its associated source or list file.

C.5 Using the Debug Control in ASM86 Programs

The DEBUG control in ASM86 puts symbol names and types into the OMF file. However, it does not include line numbers. The View Window displays disassembled code for ASM86 programs instead of using the source code with symbols. However, the symbols and types are known to the debugger and you can include symbol names and types in debugging commands.

C.6 Debugging Programs with Overlays

The OBREAK flag is normally set to true causing the user program to automatically break any time an overlay is loaded. If you do not wish this break to occur, set the OBREAK flag to false. You may wish to incorporate the command to set OBREAK into a macro file so it is done before you start debugging:

```
*obreak=false
```

C.7 Using the SWAP Control

The debugger implements flipping between the user-program screen and the debugger screen in different ways depending on the video adapters available. Swapping is the default for MDA monochrome adapters; page switching is the default for CGA, EGA, or VGA color adapters.

In some cases, color adapters may require swapping instead of page switching; namely, if the user program makes use of page switching itself, it will conflict with the debugger's use of page switching. In these cases, the SWAP control must be specified to force the debugger to use swapping instead of page switching.

The SWAP control does not resolve the conflict if the user program switches video modes on the same adapter used for the debugger control screen. In this case, setting up a second video adapter and using the MON2 control is recommended. See Section C.1 for further tips on setting up a second monitor.

C.8 Tips on Variable Names and Reserved Words

If a program symbol is the same as a reserved word, you can still access the program symbol by prefixing it with a double quote character when you type it.

For example, suppose your program has the variable name `ch` which is also a reserved word for the CH register. You can still access your program variable name and display its value instead of the value of the CH register by typing a double quote first:

```
*"ch  
B
```

C.9 Tips on Clearing Breakpoints

To clear a temporary breakpoint, scroll to the source line where the temporary breakpoint is set and press the F9 key. The F9 key toggles the temporary breakpoint on and off at the current source line shown in the View Window.

To clear a conditional breakpoint, fixed breakpoint, or tracepoint is set, scroll to the source line where the breakpoint or tracepoint is set and press the F9 key two times. The first time replaces any temporary breakpoint, conditional breakpoint, fixed breakpoint, or tracepoint at that location. The second time toggles off the temporary breakpoint.

The Clear at Cursor item on the Set Menu clears any breakpoint or tracepoint at the current location.

The Remove All item on the Set Menu clears all breakpoints and tracepoints that are currently defined.

Watch expressions can be enabled or disabled with the **ENABLE** and **DISABLE** commands; and the Watch Window can be removed from the screen with the **sF2** key or with the **Watch Window** item on the **Window Menu**. Watch expressions, however, cannot be removed or cleared.

The debugger supports the following languages:

iC-86
PL/M-86
ASM86
Fortran-86
Pascal-86

This appendix outlines the level of support provided for each of these languages.

D.1 iC-86

All C-language features provided by iC-86 Version 4.0 are supported, with the following exceptions:

- Complex number types or functions that return complex number types are not supported. For example, the following functions in `math.h` are not supported: Bessel functions of the first kind, `double hypot(r,i)`, and `double cabs(z)`. Thus, complex numbers cannot be accessed symbolically by the debugger.
- The union type is not provided. Thus, members of unions cannot be accessed symbolically by the debugger.
- Bit fields and enumeration are not supported.

D.2 PL/M-86

All PL/M language features provided by PL/M-86 Version 3.1 are supported.

D.3 ASM86

All ASM86 language features provided by ASM86 Version 3.1 are supported, with the following exceptions:

- The use of the `GROUP` directive in ASM86 causes grouped segments to be declared as a single segment with one base address. DB86 keeps track of these segments individually with separate base addresses. Thus, references to grouped segments will be inaccurate.
- No source code display is performed for any module or procedure written in assembly language.

D.4 Fortran-86

All Fortran-86 language features provided by Fortran-86 Version 3.0 are supported, with the following exceptions:

- Implicit `GO TO` conditionals are not supported by the `LSTEP`, `ISTEP`, or `PSTEP` commands in the debugger. For example, the debugger does not execute the following code fragment in the proper sequence, nor does it break at the appropriate breakpoint:

```
X = LINES - CONSTANT
IF (X) 200, 330, 450
```

D.5 Pascal-86

All Pascal language features provided by Pascal-86 Version 3.1 are supported, with the following exceptions:

- Variant record types are not supported. Thus, variant records and their fields cannot be accessed symbolically by the debugger.
- Enumerations, sets, and files are not accessed symbolically by the debugger.

Appendix E Error Messages

- 00 Type definition record with unrecognizable format.
- 01 Array's lower bound is unknown - zero is assumed.
- 02 Symbol is not an array or has fewer dimensions than
 specified.
- 03 Array index is out of bounds.

 The array index specified in the symbolic request was
 larger than the maximum dimension defined for the
 array in the original source program.
- 04 Referenced array expects a single character array
 index.
- 05 Address of module is not known.

 The module referenced contains no debug information
 and the address of the module can not be determined.
 DB86 can not provide symbolic support without
 symbolic information in the module being loaded. Be
 sure that the DEBUG option is being used when
 compiling and assembling or symbolic information will
 not be generated.
- 06 Unknown module specified.

 A module name was specified that could not be found.
 Be sure the name is spelled correctly or check the list
 of program modules with the DIR MODULE
 command.
- 07 No line information was loaded for module.

 There was no line information in the module. DB86
 can not provide source support without line
 information in the module being loaded.

 Be sure that the DEBUG option is being used when
 compiling and assembling or line information will not
 be generated.

- 08 No symbol information was loaded for module.
There was no symbolic information in the module. DB86 can not provide source or symbolic support without symbolic information in the module being loaded. Be sure that the DEBUG option is being used when compiling and assembling or symbolic information will not be generated.
- 09 Cannot determine module for specified location.
Could not find the specified location in any known module. Either the specified location is outside of the program or in a module for which there is no symbol information.
- 10 Cannot determine current default module.
Could not find current location in any known module. Either the current execution point is outside of the program or in a module for which there is no symbol information.
- 11 Symbol currently not active.
The symbol is stack resident and is only available when the current execution point is in the procedure in which the symbol is defined.
- 12 Symbol not known in current context.
The debugger cannot find and identify the symbol as either a keyword or program symbol. This message can occur if a debugger keyword or program symbol does not exist or is misspelled.
- 13 No symbol information was loaded for program.
There is no symbolic information in the loaded program. DB86 can not provide source or symbolic support without symbol information in the loaded program. Be sure that the DEBUG option is being used when compiling and assembling or symbol information will not be generated.

- 14 Symbol reference of unsupported type, displayed as a word.
DB86 does not know the type of the symbol being referenced. This may be because no type information exists in the program or DB86 does not support the symbolic type being referenced. By default The contents of the symbol is display as a WORD.
- 15 Symbol is not known to be a structure.
The symbol was specified as a structure and was found not to be a structure.
- 16 Symbol is not a known structure field name.
The symbol was specified as part of a structure and was not found to be part of the structure.
- 17 Cannot determine offset of field from start of structure.
The debugger was unable to determine the size of one of the preceding structure fields, hence the requested field cannot be referenced.
- 18 Nested symbolic references not permitted.
- 19 Symbol isn't a pointer variable or its dereference type is unknown.
Only symbols that are defined as pointers and that have an additionally defined type may be dereferenced. Only C and PASCAL allow this, PL/M does not and hence PL/M pointer may not be dereferenced.
- 20 Specified line is not an executable statement.
The specified line does not exist in the loaded program. Lines that exist may always be found by using the DIR LINE command. Disappearing line numbers may occur if compiling with any optimize level higher than 0 because some lines may be eliminated during the optimization process.

- 21 Specified line does not exist in module.
The specified line does not exist in the referenced module. Lines that exist may always be found by using the DIR LINE command. Disappearing line numbers may occur if compiling with any optimize level higher than 0 because some lines may be eliminated during the optimization process.
- 22 Cannot evaluate line reference.
The segment part of line reference pointer not known. It may be that no symbol information was 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.
- 35 Breakpoint is already defined at this address.
In order to define another breakpoint address to this address the existing break must be removed. Position the address into the view window where you may use the F9 key to toggle the breakpoint off. Setting SCOPE to this address will position the break into the view window.
- 36 Maximum number of temp breaks already defined.
DB86 limits the number of temp breaks that may be defined to 10. Either some of the existing temp breaks must be removed or the breakpoint must be assigned to a break of another type.
- 37 Execution cannot proceed.
Either you have not loaded a program or else your program has Exited. In either instance execution is not allowed to occur.
- 41 Workspace exceeded.
There is no DOS memory remaining for the debugger to utilize.

- 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.
- 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.
- 53 Overlay name does not exist.
- 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.
Strings of length >1 may not be used in numeric conversion.
- 78 Invalid floating point value for output.
- 79 Invalid expression for MTYPE.
- 80 Invalid boolean operation.
- 81 Invalid string operation.
- 82 Invalid pointer operation.
- 84 Attempt to assign value to code instead of variable.
An attempt was made to assign an expression to a location associated with user code (eg, :main.procl = 5, where procl is a procedure in module main). Direct assignments may only be made to variables or with mtype operators (eg, BYTE :main.procl = 5).
- 85 Attempt to assign illegal value to BASE variable.

- 87 Not in a procedure or in a procedure with no debug information.
In order for the calling procedure to be identified (and the CALLS 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 sequence of operations performed by the debugger caused DB86 to overflow its run-time stack.
- 89 UDI Exception.
The operation performed generated a DB86 run-time exception. Operations such as divide-by-zero are known to cause this exception.
- 91 Illegal extended integer.
- 94 Error occurred during shell escape.
Either there is insufficient DOS memory available or COMMAND.COM is not visible in the current directory.
- 96 Illegal File Extension.
The specified file extension is not valid for DOS. Extensions are restricted to no more than three characters in length.
- 97 Maximum number of file extensions exceeded.
The maximum number of list file extensions that may be specified is 10.
- 98 Illegal file name in SETMOD.
The filename used is not a valid DOS filename.
- 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. Start address needs fixing up by linker.
Program start address needs fixup by linker.
- 113 The 8087 Emulator was not found in the load module.
- 115 Bad object record in load file.
The loader encountered a record while loading the program that was not recognized. This can be caused by loading a file that is not an object file (for example a source file), an object file that is not in OMF86 form, or an object file that has been corrupted.
- 116 Load file contains absolute load addresses.
The load file contains a module which has been fixed in memory. Possibly the file has been run through LOC86 which creates an absolute program. The debugger will only load relocatable (LTL) programs linked with LINK86 using the BIND control.
- 117 Load file contains unresolved externals.
The program being loaded was found to have unresolved referenced to symbols and the program load has been aborted. Be sure all program modules have been properly linked together and that no unresolved symbol warnings are issued during the final program link.
- 118 Accessed Symbol not in resident overlay.
The symbol being accessed is defined in an overlay that is not currently resident in the user program overlay area. The value returned will not be correct as it represents a value that exists in another overlay. Correct reverences may only be made when the overlay in which the symbol is defined is in memory.

- 119 Memory segment request failure during load.
More memory is needed to load program than is available. More memory may be obtained by eliminating programs that are resident in memory before the debugger is used.
- 120 Load module contained no starting address information.
The load module did not provide any information about the starting address. The load has been aborted and execution of the program is not possible. Be sure that a main module has been created and linked into the program.
- 136 Divide by zero (operation yields 0 result).
- 137 Invalid type for arithmetic.
- 138 Invalid integer operation.
- 139 Real math is not available.
In order to use real math (including any operations or reference to real numbers), you must have an 8087 math coprocessor or have the 8087 emulator linked into the program under debug. This error may be detected if references to 8087 libraries were encountered in the loaded program but neither the 87 emulator or math chip was found by DB86.
- 140 Invalid real number.
- 141 Attempted real comparison with NAN, +infinity or -infinity.
- 142 Invalid real operation.
- 143 Invalid extended integer operation.
- 144 Illegal numeric constant.
- 160 Attempt to INCLUDE :CI.
- 161 I/O error on INCLUDE file.
- 162 I/O error on LIST file.
- 163 I/O error while loading object file.
- 164 Could not open load file.

- 165 Error while attempting to open virtual symbol table.
The virtual symbol table uses :WORK: for the disk-resident portion of the virtual symbol table. Ensure that the device for :WORK: is ready and that DB86 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.
The disk device used for the virtual symbol table cannot be written to. The most common cause of this problem is a full device.
- 168 Error while attempting to close virtual symbol table.
- 169 Error while attempting to read virtual symbol table.
- 177 First address is greater than second address.
- 180 Illegal mnemonic.
- 181 Illegal number.
- 182 Unrecognized 8086/8087 mnemonic.
- 183 Illegal use of indirect addressing.
The correct forms of indirect addressing are:
 <symbolic ref> [BX] + offset
 <symbolic ref> [BP] + offset
 <symbolic ref> [DI] + offset
 <symbolic ref> [SI] + offset
 <symbolic ref> [BP] [DI] + offset
 <symbolic ref> [BP] [SI] + offset
 <symbolic ref> [BX] [DI] + offset
 <symbolic ref> [BX] [SI] + offset
The symbolic reference (of the form :MODULE.SYMBOL.SYMBOL.etc) and the '+ offset' are optional.
- 184 Illegal single line assembler operand.
- 185 Single line assembler syntax error. See HELP ASM.
- 186 Memory pointer (e.g., BYTE, WORD, etc) without memory operand (e.g., number or symbolic reference).

- 187 Too few operands for this instruction.
- 188 Illegal operands, both operands appear to reference memory.
- 189 The types of the operand(s) do not match the mnemonic or each other.
- 190 One byte relative jump is out of range. Range is -128 to +127.
- 198 BASE must be 2T, 10T or 16T.
The value was detected during the use of the BASE variable.
- 226 Unsupported variable assignment.
The attempted assignment operation is invalid.
Potential causes:
Assigning a value to a variable whose type is unknown;
(Use a memory template command to perform the assignment)
- 249 Insufficient memory for Virtual Symbol Table buffers.
- 512 The cause of execution break is unknown to DB86.
Execution was broken in a manner that DB86 cannot determine. It was not via a known breakpoint or a control-c. Most likely caused by placing an interrupt at the given address.
- 513 This breakpoint is already active.
- 532 No program was loaded.

Appendix F Reserved Words

This appendix contains the keywords that DB86 recognizes and uses. To avoid ambiguity, do not use these keywords as user-defined names for debug objects. The debugger always tries to interpret these names as keywords first, which can have confusing results if you were attempting to reference the user-defined name instead of the keyword. To force the debugger to reference the user-defined name instead of the corresponding keyword, you must prefix the symbol with a quotation mark (").

!	AX	DH	FCW
"	BASE	DI	FDA
\$	BH	DIR	FIA
(BINARY	DISABLE	FILE
)	BL	DL	FIO
*	BOOLEAN	DO	FLAGS
+	BP	DS	FOREVER
,	BX	DWORD	FSW
-	BYTE	DX	FTW
.	CALLS	ELSE	GLOBAL
/	CALLSTACK	ENABLE	GO
:	CB	END	HELP
;	CB0	ENDCOUNT	HEX
<	CB1	ENDIF	IF
<=	CB2	ENDREPEAT	IFL
<>	CB3	ENUMERATION	INCLUDE
=	CFL	ES	INTEGER
==	CH	EVAL	IP
>	CHAR	EXIT	ISTEP
>=	CL	EXP	LABEL
ADDRESS	COUNT	EXTINT	LENGTH
AFL	CS	FALSE	LINE
AH	CSTEP	FB	LIST
AL	CX	FB0	LOAD
AND	DEBUG	FB1	LOCALS
ARRAY	DECIMAL	FB2	LONGINT
ASM	DEFINE	FB3	LONGREAL
	DFL	FB4	LSTEP
		FB5	

MOD	REGS87	ST5	TR8
MODULE	REPEAT	ST6	TR9
NAMESCOPE	RSTEP	ST7	TRUE
NOLIST	SASM	STACK	UNTIL
NOT	SCOPE	SYMBOL	WA
OBREAK	SELECTOR	SYSINT	WA0
OFL	SET	TEMPREAL	WA1
OR	SETMOD	THEN	WA2
ORIF	SFL	TIL	WA3
OV	SI	TO	WA4
PFL	SMALLINT	TR	WA5
POINTER	SP	TRO	WHILE
PORT	SPATH	TR1	WORD
PROCEDURE	SS	TR2	WPORT
PSTEP	ST0	TR3	XOR
PUBLIC	ST1	TR4	ZFL
REAL	ST2	TR5	[
RECORD	ST3	TR6]
REGS	ST4	TR7	^

Appendix G ASCII Codes

This appendix lists ASCII codes. Table G-1 is a list of codes. Table G-2 is a list of code functions.

Table G-1 ASCII Code List

Dec	Hex	Character	Dec	Hex	Character
0	00	NULL	29	1D	GS
1	01	SOH	30	1E	RS
2	02	STX	31	1F	US
3	03	ETX	32	20	SP
4	04	EOT	33	21	!
5	05	ENQ	34	22	"
6	06	ACK	35	23	#
7	07	BEL	36	24	\$
8	08	BS	37	25	%
9	09	HT	38	26	&
10	0A	LF	39	27	'
11	0B	VT	40	28	(
12	0C	FF	41	29)
13	0D	CR	42	2A	*
14	0E	SO	43	2B	+
15	0F	SI	44	2C	,
16	10	DLE	45	2D	-
17	11	DC1	46	2E	.
18	12	DC2	47	2F	/
19	13	DC3	48	30	0
20	14	DC4	49	31	1
21	15	NAK	50	32	2
22	16	SYN	51	33	3
23	17	ETB	52	34	4
24	18	CAN	53	35	5
25	19	EM	54	36	6
26	1A	SUB	55	37	7
27	1B	ESC	56	38	8
28	1C	FS	57	39	9

Table G-1 ASCII Code List (continued)

Dec	Hex	Character	Dec	Hex	Character
58	3A	:	93	5D]
59	3B	;	94	5E	^
60	3C	<	95	5F	-
61	3D	=	96	60	'
62	3E	>	97	61	a
63	3F	?	98	62	b
64	40	@	99	63	c
65	41	A	100	64	d
66	42	B	101	65	e
67	43	C	102	66	f
68	44	D	103	67	g
69	45	E	104	68	h
70	46	F	105	69	i
71	47	G	106	6A	j
72	48	H	107	6B	k
73	49	I	108	6C	l
74	4A	J	109	6D	m
75	4B	K	110	6E	n
76	4C	L	111	6F	o
77	4D	M	112	70	p
78	4E	N	113	71	q
79	4F	O	114	72	r
80	50	P	115	73	s
81	51	Q	116	74	t
82	52	R	117	75	u
83	53	S	118	76	v
84	54	T	119	77	w
85	55	U	120	78	x
86	56	V	121	79	y
87	57	W	122	7A	z
88	58	X	123	7B	{
89	59	Y	124	7C	
90	5A	Z	125	7D	}
91	5B	[126	7E	~
92	5C	\	127	7F	DEL

Table G-2 ASCII Control Code Definition

Abbreviation	Meaning	Dec	Hex
NUL	NULL Character	0	0
SOH	Start of Heading	1	1
STX	Start of Text	2	2
ETX	End of Text	3	3
EOT	End of Transmission	4	4
ENQ	Enquiry	5	5
ACK	Acknowledge	6	6
BEL	Bell	7	7
BS	Backspace	8	8
HT	Horizontal Tabulation	9	9
LF	Linefeed	10	0A
VT	Vertical Tabulation	11	0B
FF	Form Feed	12	0C
CR	Carriage Return	13	0D
SO	Shift Out	14	0E
SI	Shift In	15	0F
DLE	Data Link Escape	16	10
DC1	Device Control 1	17	11
DC2	Device Control 2	18	12
DC3	Device Control 3	19	13
DC4	Device Control 4	20	14
NAK	Negative Acknowledge	21	15
SYN	Synchronous Idle	22	16
ETB	End of Transmission Block	23	17
CAN	Cancel	24	18
EM	End of Medium	25	19
SUB	Substitute	26	1A
ESC	Escape	27	1B
FS	File Separator	28	1C
GS	Group Separator	29	1D
RS	Record Separator	30	1E
US	Unit Separator	31	1F
SP	Space	32	20
DEL	Delete	127	7F



active window	the window in which the cursor is located. You can enter commands at the command-line prompt if the Command Window is active. You can scroll through the user program source if the View Window is active.
address	a memory location expressed as a <i>segment:offset</i> pair. Both the <i>segment</i> and the <i>offset</i> are unsigned integers in the range 0 to 65535. An address corresponds to a location in the 8086, target-system memory.
alias	another name for a character string.
base	see number base.
Break Status Column	a single column to the left of the View Window. This column contains single letter codes indicating where breakpoints and tracepoints are set.
browsing	the ability to move the current scope from the current module into other modules in the user program.
CGA	Color Graphics Adapter.
call ancestry	a list of procedure names that constitute the history of procedure calls that were executed prior to the current execution point.
Command Window	the window that appears at the bottom of the screen for entry of DB86 commands.
control construct	a grouping of commands to be executed in a block.

current execution point	the location of the instruction pointer in the user program (CS:IP).
debug environment	the configuration of debugger control options and user-defined debug objects.
debug session log	a list file that records all interaction between the user and the debugger during a debug session.
debug variable	a variable defined at debug time as a specific data type, optionally given an initial value, and stored in host memory.
disassembled code	code displayed in assembly language.
EGA	Enhanced Graphics Adapter.
exception breakpoint	one of four built-in breakpoints recognized by DB86.
flipping	the ability to switch between the DB86 control screen and the user program output screen.
hidden breakpoint	a breakpoint that is set within the range of a program line but is not exactly on the line boundary. For example, a breakpoint that is set on an assembly-language instruction that is not the first instruction in a disassembled line. A hidden breakpoint is designated by a plus sign (+) in the Break Status Column.
highlight bar	the horizontal, screen wide marker that highlights the current execution point in the View Window.
history buffer	memory storing a record of command entries.

home scope	the scope associated with the current execution point. When directed with the Grey * command, DB86 returns directly to the home scope after browsing or scrolling.
human interface	the part of the debugger software that assists you in entering commands, using the menu system and keyboard, and performing other tasks.
keyword	a character string that has a reserved definition in DB86.
least-significant bit (LSB)	lowest value bit. Not the same as low order, which indicates a bit position.
list file	a host-system file in which all interaction on the terminal is saved. Do not confuse list file with listing files which are compiler-generated files containing the source code for the user program.
load-file tail	a string made up of options and controls required by the user program. Used when the user program is specified on the DB86 invocation line. Can also be set in the LOAD command.
MDA	Monochrome Display Adapter.
make file	a text file containing the necessary commands to compile and link a user program. Make files can be interpreted by a separate MAKE utility, or they can be automated batch files (.BAT) which are executed by the command interpreter of the operating system.
mapped memory	the memory ranges that are accessed by the user program during debugging.
mtype	memory type.

Menu Bar	the menu selections that appear at the top of the screen for selection of DB86 menus. The Menu Bar can be turned off.
Menu System	provides easy visual access to many of the same debugging features and commands as the Command Window.
most-significant bit (MSB)	highest value bit. Not the same as high order, which indicates a bit position.
number base	setting to which all input and output is interpreted (e.g., binary, decimal, or hexadecimal numbers).
OMF86	object module format for the 8086 microprocessor. OMF is the Intel standard for the structure of object modules.
partition	an address with an associated length in the target system.
patch	a user modification to the program being debugged. See the ASM command for further information on assembling patches into memory.
point-and-shoot	a method of debugging in which the user locates bugs by scrolling through the program source, setting breakpoints, and executing the program to those breakpoints.
port	circuitry on a microprocessor for input to or output from an external device.
pseudovvariable	a system defined variable that cannot be removed, with a predefined value range. Like a command, a pseudovvariable initiates operations and affects system operation. Like a variable, a pseudovvariable has a name and value and can be assigned, displayed, and used in expressions.

radix	see number base.
Register Window	the window that appears as an option at the right side of the screen. Target register contents are displayed and, during program execution, updated.
scope	an address in the user program that determines symbol visibility, starting with the innermost program block and moving outward to the current module.
scrolling	makes it possible to move the cursor or window within the current module.
search string	string specified by user for the Find item in the Browse Menu.
shell escape	passes host operating system commands to the host operating system.
source display file	either the listing file produced by the compiler or the source program file used as input to the compiler. The debugger uses these files for source display in the View Window.
Status Line	the line that appears at the bottom of the screen for indicating the current status of DB86 and providing messages about miscellaneous program activity.
stepping	executing the user program one specified step at a time.
stype	a user program variable of type ARRAY, RECORD, PROCEDURE, or LABEL.
swapping	one of two internal techniques used by DB86 to switch between the DB86 control screen and the user program output screen. (See Appendix B).

symbolic information	information about user program elements such as modules, procedures, and variables, which is available when the program has been translated with the DEBUG option.
thumbmark	the marker appearing in the left margin of the View Window and indicating the cursor location. The thumbmark is displayed even when the View Window is not the active window.
toggle	turn off or on.
tracepoint	a specified program location like a breakpoint, but which does not stop execution. When execution passes through a tracepoint, an announcement appears in the Command Window.
View Window	the window in which the user program source is displayed during debugging.
VGA	Video Graphics Array.
watch expression	memory template commands, references to symbolic variables, or references to a unique expression. The program variables or memory activity is displayed in the Watch Window at program execution points.
watchpoint	a construct created by using the conditional breakpoint and auto stepping features of the debugger. A user-specified expression is evaluated at every step during auto stepping. It causes a breakpoint to occur when the expression evaluates to true.
Watch Window	the window that appears as an option at the top of the screen for viewing user program variables during program execution.

! Shell Escape, 5-10
+, 5-17
43-line control, 2-7
8086/8088 registers and flags, 5-95, 5-96
8087 coprocessor, 1-3
8087 registers and flags, 5-99, 5-100

A

Aborting a command, 4-8
Accessing the menu system, 5-87
Accessing user-program memory, 5-84
Active window, Glossary-1
Address, Glossary-1
Alias, Glossary-1
Application development process, 1-4
ASM Display, 3-7, 3-11
ASM examples, 5-13
ASM, 5-11
Assembler data types, 5-34
Assembling code into memory, 5-12
Assembly, 1-2
Assignment command, 2-26
Auto Step mode, 3-13, 5-22
Automatic type conversions, 5-60, 61

B

Base 10, 5-15
Base 16, 5-15
Base 2, 5-15
BASE examples, 5-16
BASE, 5-15, 5-77
Base, Glossary-1
Binary, 5-15
Blinking screen, 3-9
Boolean data types, 5-34
Boolean operators, 5-54
Break status, 5-17
Break Status Column, Glossary-1
Breaking on access to data, 2-13

Breakpoints, 1-2, 1-3, 1-18, 2-10, 2-12, 3-5, 3-8, 3-12, 3-13, 3-15, 3-16, 3-18, 4-5, 4-7, 5-17, 5-22, 5-46, 5-47, 5-62

Browse Menu, 3-19

Browsing, 2-19, 3-19, 3-20, 4-2, 4-8, Glossary-1

C

Call ancestry, 3-17, Glossary-1
Call instruction, 3-14, 4-6
Call Step, 3-13
Called procedure, 3-13, 3-14, 3-17, 4-6, 4-7, 5-32
Calls, 3-17, 3-21
CALLS, 5-19
CALLS example, 5-20
Callstack, 4-2, 5-19
Callstack browsing, 3-21
CB, 5-22
CGA, Glossary-1
Changing a byte port, 5-89
Changing a word port, 5-121
Changing debug variables, 5-35
Changing flags, 5-64
Changing user-program symbols, 5-118
Changing variables, 5-84
Character data types, 5-34
Character delimiters, 5-76, 77, 78
Clear at Cursor, 3-16
Clearing breakpoints, 2-14
Clearing tracepoints, 2-14
Command entry, 5-24
Command line, 1-3
Command overview, 5-3
Command Window, 3-10, 3-15, 4-2, 4-3, 4-5, 4-6, 5-24, Glossary-1
Command-line buffer, 4-3
Command-line comments, 5-25
Command-line controls, 2-27

- Command-line editing, 4-3, 4-4, 5-24, 25, 26
- Command-line entry, 5-24
- Command-line interface, 4-3, 4-4, 5-24, 25, 26
- Comments, 5-25
- Composite data types, 5-33
- Conditional breakpoint examples, 5-23
- Conditional breakpoints, 2-12, 5-17, 5-22
- Configuring the debug environment, 2-27
- Continuation character, 5-24
- Continuing command-lines, 5-24
- Control construct, Glossary-1
- Control constructs, 5-102, 5-4, 5-28, 5-30, 5-44, 5-71,
- Control keys, 4-1
- Controlling the display of DB86 information, 5-27
- Controls, 2-27
- Correlating a module with a source file, 5-108
- COUNT, 5-28, 5-30
- COUNT example, 5-31
- CSTEP, 5-32
- Ctrl-Break key, 2-14
- Current execution point, Glossary-2
- Cursor, 4-1, 4-4
- Cycle Window, 3-10

D

- Data access, 1-2
- Data types, 5-33, 5-55
- DB86 commands, 5-36
- DB86 debugger features, 1-2
- DB86 help, 1-12
- DB86 invocation, 1-8
- DB86 prompt, 5-24
- DB86 screen display, 1-9, 3-9
- DB86, learning, 1-3, 3-1
- Debug environment, Glossary-2
- Debug environment commands, 5-4
- Debug Menu, 3-4

- Debug session log, 5-79, Glossary-2
- Debug Status, 3-18
- Debug variable, Glossary-2
- Debug variables, 3-18, 5-15, 5-35, 5-53
- Debug variables, types, 5-37
- Debugger commands, 5-36
- Debugger screen, 3-8
- Debugger source display files, 2-4
- Debugging session, log of, 1-3
- Debugging with DB86, 1-6
- Decimal, 5-15
- Default extension, 3-5
- DEFINE, 5-37
- DEFINE example, 5-38
- Defining debug variables, 5-35, 5-37
- Deleting characters, 4-4
- Delimiters, 5-76, 5-77, 5-78
- Dereferencing pointers, 2-24
- DIR, 5-39, 5-40
- DIR example, 5-42
- DISABLE, 5-46, 5-47
- DISABLE example, 5-47
- Disabling breakpoints, 2-14, 5-46, 5-47
- Disassembled code, Glossary-2
- Disassembly, 1-2, 5-11
- Disassembly display, 2-22, 2-26, 3-7, 3-11, 4-5
- Display control, 5-27
- Displaying a byte port, 5-89
- Displaying a flag, 5-94
- Displaying a register, 5-94
- Displaying a watch expression, 5-119
- Displaying a word port, 5-121
- Displaying an 8086/8088 flag, 5-95
- Displaying an 8086/8088 register, 5-95
- Displaying an 8087 flag, 5-99
- Displaying an 8087 register, 5-99
- Displaying data, 2-22
- Displaying debug variables, 5-35
- Displaying flags, 5-64
- Displaying memory, 5-84
- Displaying stack values, 5-112
- Displaying symbols, 5-39, 40

- Displaying the base, 5-15
- Displaying the path, 5-110
- Displaying user output, 1-22
- Displaying user-program symbols, 5-118
- Displaying variables, 5-84
- DO, 5-28, 5-37, 5-44
- DO example, 5-44
- DOS interrupts, user program, 5-114
- DOS Shell, 3-6, 5-10
- DOS version number, 1-7
- Dumping memory, 2-25, 5-84

E

- Editing command lines, 4-3, 4-4, 5-24, 25, 26
- Editing user responses, 4-4
- EGA, Glossary-2
- ENABLE, 5-46, 5-47
- ENABLE example, 5-47
- Enabling breakpoints, 2-14, 5-46, 5-47
- Entering commands, 1-12
- Error message help, 4-6
- EVAL, 2-25, 5-49, 5-50
- EVAL examples, 5-50
- Evaluating expressions, 5-49
- Evaluating line numbers, 5-49
- Evaluating procedures, 5-49
- Evaluating symbols in overlays, 5-49
- Evaluating symbols, 5-49
- Examining data, 2-22
- Example debugging, 2-28
- Exception breakpoint, Glossary-2
- Exception breakpoints, 5-114, 5-17
- EXE files during linking, 2-5
- Executing a command, 4-4
- Executing a user program, 1-20, 2-15
- Execution and watch commands, 5-7
- EXIT, 5-52
- EXIT example, 5-52
- Exiting from DB86, 1-24, 3-6, 5-52
- Exiting from the Help Menu, 3-3
- Exiting from the menu system, 3-3
- Expanded Calls, 3-18

- Expression typing, 5-58, 59, 60, 61
- Expressions, 5-53
- Extension list, 3-5
- Extensions, 3-5

F

- Fast mode, 5-27
- FB, 5-62
- Find ..., 3-21
- Finding a bug, 2-28
- Fixed breakpoint examples, 5-62
- Fixed breakpoints, 2-12, 5-17, 5-62
- FLAGS examples, 5-65
- Flags, 3-7, 3-8
- FLAGS, 5-64
- Flags, 5-94
- Flip Screen, 3-8, 3-9
- Flipping, 2-19, Glossary-2
- Floating-point data types, 5-33
- Function keys, 1-3, 4-5

G

- Global debug variables, 5-37
- GO, 2-15, 5-66
- GO examples, 5-67
- Go Forever, 3-12
- GO FOREVER, 5-66
- Go Keep Fixed, 3-12
- Go Menu, 2-15, 3-11
- Go Til Breakpoint, 3-12
- Go Til Cursor Line, 3-12
- GO TIL, 5-66
- Grouping commands, 5-28

H

- HELP example, 5-69
- Help Menu, 3-23
- Help screens, 3-23
- Help, 1-3, 3-23, 4-5, 5-68
- Hexadecimal, 5-15
- Hidden breakpoint, Glossary-2
- Hidden breakpoints, 5-17
- Highlight bar, Glossary-2

History buffer, Glossary-2
Home scope, Glossary-3
Human interface, Glossary-3

I

Identifiers, 5-76
IF, 5-28, 5-71
IF example, 5-72
INCLUDE, 5-73
INCLUDE example, 5-73
Include files, 2-27
Insert editing mode, 4-4
Invoking the debugger, 2-6
ISTEP example, 5-74
ISTEP, 5-74

K

Keyboard controls, 1-3, 4-1
Keyword, Glossary-3
Keywords, 5-76

L

L43 control, 2-7
Language instruction stepping, 5-82
Last error help, 5-68
Least significant bit (LSB), Glossary-3
Lexical elements, 5-76
Line mode, 5-27
Line-editing keys, 4-3
Link Response File, 2-4
Linking, 2-2
LIST example, 5-79
List file, Glossary-3
List files, 2-27
List files during compile, 2-4
LIST, 5-79
LOAD example, 5-80
Load Program, 3-4
LOAD, 5-80
Load-file tail, 2-6, Glossary-3
Loading a user program, 2-6, 2-8, 3-4,
5-80
Local debug variables, 5-37

Index-4

Local variables, 3-18, 3-21
LSTEP, 5-82
LSTEP example, 5-82

M

Machine instruction stepping, 5-74
Macro files, 2-7, 2-27
Macros, 5-73
Make file for linking, 2-2
Make file, Glossary-3
Mapped memory, Glossary-3
MDA, Glossary-3
Memory access commands, 5-6
Memory access examples, 5-85
Memory access, 1-2
Memory access, 5-84
Menu bar, 3-1, 3-9
Menu Bar, Glossary-4
Menu prompting, 3-3, 3-20
Menu system, 1-3, 3-1, 3-2, 3-9, 5-24,
5-87, Glossary-4
Modifying data, 2-26
Modules, 3-18
Modules in EXAMPLE program, 2-2
MON2 control, 2-7
Monitors, two, 2-7
Most significant bit (MSB), Glossary-4
Moving around in the user program,
5-106
Mtype variables, 5-39
Mtype, Glossary-3
Mtypes, 5-41, 5-42
Multiple command lines, 5-24

N

Names, 5-76
NAMESCOPE, 5-106
Navigational key controls, 4-1
Nested procedures, 3-17, 4-7, 5-19
Nesting DO blocks, 5-44
Nesting INCLUDE commands, 5-73
Next Find, 3-22
No Flipping, 3-8, 3-9

NOLIST, 5-79
NOLIST example, 5-79
Number base, Glossary-4
Numbers, 5-76

O

OBREAK, 5-88, 5-91
OBREAK example, 5-88
OBREAK flag, 2-14
Observing program output, 2-19
OMF86, Glossary-4
Opening a debug session log, 5-79
Operands, 5-53
Operators, 5-53, 5-54, 5-55
Output screen, 3-8, 3-9, 4-5
Overlay, 1-3
Overlays, user program, 5-88
Overstrike editing mode, 4-4

P

Page mode, 5-27
Paired commands, 5-28
Partition, Glossary-4
Patch, Glossary-4
Patching a user program, 2-22
Patching code, 5-12
Pathname, 3-5
Path, 5-110
PC-DOS version number, 1-7
Point-and-shoot, Glossary-4
Pointer data types, 5-34
Point-and-shoot debugging, 2-12
PORT examples, 5-89
PORT, 5-89, 91
Port, Glossary-4
Preparing a program for debugging, 2-2
Previous Find, 3-22
Procedure calls, 5-19
Procedure Step, 3-14
Procedure stepping, 5-92
Processor status commands, 5-5
Program development, 1-4
Program symbols, 1-2

Program variables, 3-18, 3-20
Prompts, 3-3, 3-20
Pseudovvariable, Glossary-4
Pseudovvariables, 5-88, 5-89, 5-91, 5-114,
5-121

PSTEP, 5-92

PSTEP example, 5-92

Q

Quit Help, 3-23

R

Radix, 5-15, 5-77, Glossary-5
Register access, 1-2
Register Window, 3-8, 4-5, Glossary-5
Registers and flags, 3-7, 5-94
REGS, 5-95
REGS example, 5-98
REGS87, 5-99
Relational operators, 5-54
Reload, 3-5
Reloading a program, 3-5
Remove All, 3-16
REPEAT, 5-28, 5-102
REPEAT example, 5-102
Repeating a block of commands, 5-102
Return instruction stepping, 5-104
Return instruction, 3-14, 4-7
Return Step, 3-14
Returning to DOS shell, 3-6, 5-10
RSTEP, 5-104
RSTEP example, 5-104
Running with two monitors, 2-7

S

Sample command entry, 5-2
SASM, 5-12
Scope ..., 3-20
Scope, 3-19, 3-20, 4-2, 4-8, Glossary-5
SCOPE, 5-106
Screen flipping, 1-3, 3-6, 3-8, 3-9, 4-7
Screen, blinking, 3-9

Scrolling, 1-16, 2-19, 3-19, 3-20, 4-1, 4-2,
 Glossary-5
 Search string, Glossary-5
 Searching for a user program symbol,
 5-106
 Searching, 3-21, 3-22
 Selecting a menu, 3-2
 Selecting a menu item, 3-2
 Selecting the menu bar, 3-2
 Separate source files, 2-2
 Sequencing through a user program,
 2-15
 Set Menu, 3-15
 SETMOD, 5-108
 SETMOD example, 5-109
 Setting 8086/8088 flags, 5-95
 Setting 8086/8088 registers, 5-95
 Setting 8087 flags, 5-99
 Setting a new execution point, 5-106
 Setting breakpoints, 2-10
 Setting flags, 5-64, 5-94
 Setting registers, 5-94
 Setting the base, 5-15
 Setting the path, 5-110
 Setting watch expressions, 5-119
 Shell escape, 5-10, Glossary-5
 Shotgun approach, 2-15
 Showing user output, 1-22
 Signed data types, 5-33
 Single-step, 1-2
 Source display, 1-2, 3-7, 4-5
 Source display commands, 5-6
 Source display file, Glossary-5
 Source path, 2-7
 Source Path, 3-4, 3-5
 SPATH, 2-7, 5-110
 Specifying drive and path names, 3-4
 STACK, 5-112
 STACK example, 5-112
 Starting program execution, 5-66
 Status Line, 3-17, Glossary-5
 Step, 3-13
 Stepping, 3-8, 3-11, 3-13, 3-14, 4-7, 5-32,
 5-74, 5-82, 5-92, 5-104, 5-113,
 Glossary-5
 Stepping commands, 2-16, 5-113
 Strings, 5-76, 5-77
 Stype, Glossary-5
 Stype variables, 5-39
 Swapping, Glossary-5
 Symbol, user program, searching for,
 5-106
 Symbolic display, 5-84
 Symbolic information, Glossary-6
 Symbols, 5-76
 SYSINT, 5-91, 5-114
 SYSINT example, 5-115
 SYSINT flag, 2-14
 SysReq key, 2-14

T
 Temporary breakpoints, 2-12, 5-17
 Thumbmark, 3-21, Glossary-6
 Toggle, Glossary-6
 TR, 5-116
 TR example, 5-117
 Tracepoint example, 2-10, 5-117
 Tracepoint, Glossary-6
 Tracepoints, 3-12, 3-15, 3-16, 3-18, 4-5,
 4-7, 5-17, 5-46, 5-47, 5-116
 Transparent overlay, 1-3
 Two monitors, 2-19
 Type conversions, 5-58, 5-59, 5-60, 5-61
 Type operators, 5-55, 5-57
 Types, 5-33

U
 Unary and binary operators, 5-54
 Unsigned data types, 5-33
 User response, 3-3, 3-20
 User response editing, 4-3
 User-program symbols, 5-118

V

VGA, Glossary-6

View Window, 3-7, 3-9, 3-10, 3-11, 3-16,
3-17, 3-19, 4-1, 4-2, 4-5, 4-6, 4-8,
5-24, Glossary-6

W

WA, 5-119

WA examples, 5-120

Watch expression, Glossary-6

Watch expression examples, 5-120

Watch expressions, 1-2, 2-24, 2-38, 3-13,
3-18, 5-17, 5-46, 5-47, 5-119

Watch Window, 1-2, 2-38, 3-8, 4-7,
Glossary-6

Watchpoint, Glossary-6

Watchpoints, 2-13

Window divider, 4-8

Window Menu, 3-7

WPORT, 5-91, 5-121

WPORT examples, 5-121

(

(

(



READER RESPONSE CARD

We'd Like Your Opinion

Please use this form to help us evaluate the effectiveness of this manual and improve the quality of future versions.

To order publications, contact the Intel Literature Department (see page ii of this manual).

Fill in the squares below with a rating of 1 through 10:

POOR			AVERAGE				EXCELLENT		
1	2	3	4	5	6	7	8	9	10
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Readability									
<input type="checkbox"/> Technical depth									
<input type="checkbox"/> Technical accuracy									
<input type="checkbox"/> Usefulness of material for your needs									
<input type="checkbox"/> Comprehensibility of material									
<input type="checkbox"/> OVERALL QUALITY OF THIS MANUAL									

If you gave a 4 or less (in any category), please explain here:

What suggestions would you have for improving this manual:

If you would like us to call you for more specific suggestions about this book, please additionally fill in your phone number below.

Name _____

Phone Number (_____) _____

Address _____

Thanks for taking the time to fill out this form.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 HILLSBORO, OR

POSTAGE WILL BE PAID BY ADDRESSEE

**INTEL CORPORATION
DTO TECHNICAL PUBLICATIONS HF2-38
5200 NE ELAM YOUNG PARKWAY
HILLSBORO OR 97124-9978**



Please fold here and close the card with tape. Do not staple.

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the other side of this form will help us produce better manuals. Each reply will be reviewed. All comments and suggestions become the property of Intel Corporation.

If you are in the United States and are sending only this card, postage is prepaid.

If you are sending additional material or if you are outside the United States, please insert this card and any enclosures in an envelope. Send the envelope to the above address, adding "United States of America" if you are outside the United States.

Thanks for your comments.



International Sales Offices

BELGIUM

Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK

Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND

Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND

Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE

Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL

Intel Semiconductors LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY

Intel Corporation S.P.A.
Milanfiori, Palazzo E/4
20090 Assago (Milano)

JAPAN

Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

NETHERLANDS

Intel Semiconductor (Nederland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

NORWAY

Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN

Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN

Intel Sweden A.B.
Dalvaegen 24
S-171 36 Solna

SWITZERLAND

Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY

Intel Semiconductor GmbH
Seidlestrasse 27
D-8000 Muenchen 2

1

2

3