

DEBUGGER MANUAL

Specifications Subject to Change.

Convergent Technologies, Convergent, CTOS, CT-BUS,
CT-DBMS, CT-MAIL, CT-Net, AWS, IWS, and NGEN are
trademarks of Convergent Technologies, Inc.

Third Edition (November 1983) A-09-00320-01-A

Copyright © 1983 by Convergent Technologies, Inc.

CONTENTS

GUIDE TO TECHNICAL DOCUMENTATION.....	vii
SUMMARY OF CHANGES.....	xvii
1 OVERVIEW	1-1
INTRODUCTION	1-1
WHO SHOULD USE THIS MANUAL	1-1
HOW TO USE THIS MANUAL	1-2
CONVENTIONS USED IN THIS MANUAL	1-4
2 CONCEPTS	2-1
COMMAND PARAMETERS	2-1
CONSTANTS (NUMBERS, PORTS, AND TEXT) ...	2-3
Numbers.....	2-3
Ports.....	2-3
Text.....	2-4
SYMBOLS	2-4
ADDRESS EXPRESSIONS	2-5
SYMBOLIC INSTRUCTIONS	2-5
CURRENT VALUE	2-5
DEBUGGER MODES	2-6
3 GETTING STARTED WITH THE DEBUGGER	3-1
ENTERING THE DEBUGGER	3-1
Debugger Prompts.....	3-2
Debugger Displays.....	3-2
EXITING FROM THE DEBUGGER	3-2
USING THE DEBUGGER AS A CALCULATOR	3-3
CODE-F : OPENING A SYMBOL FILE	3-3
CODE-R: CHANGING THE BASE OF THE NUMBER SYSTEM	3-5
4 EXAMINING AND CHANGING MEMORY CONTENTS .	4-1
LOOKING AT MEMORY	4-1
Using Pointers to Display Memory Contents.....	4-2
Displaying Several Locations at Once..	4-3
Debugger Prompts.....	4-3
CODE D: DISPLAYING THE CONTENTS OF MEMORY	4-3
CHANGING THE CONTENTS OF A MEMORY LOCATION	4-4
Changing Instructions.....	4-5

	OPENING A NEW LOCATION	4-5
	CODE-O: SEARCHING FOR A BYTE PATTERN IN MEMORY	4-6
5	USING BREAKPOINTS	5-1
	CODE-B: SETTING AND QUERYING BREAKPOINTS	5-1
	CODE-C: CLEARING BREAKPOINTS	5-2
	CODE-A: SETTING CONDITIONAL BREAKPOINTS	5-2
	CODE-P: PROCEEDING FROM A BREAKPOINT ...	5-4
	CODE-G: STARTING A PROCESS AT A SPECIFIED ADDRESS	5-5
	CODE-X: EXECUTING INSTRUCTIONS INDIVIDUALLY	5-5
	CODE-E: BREAKING AFTER THE CURRENT INSTRUCTION	5-6
6	WORKING WITH REGISTERS	6-1
	THE PROCESS REGISTER	6-1
	EXAMINING AND MODIFYING REGISTERS	6-1
	USING REGISTER MNEMONICS	6-2
7	ELEMENTARY DISPLAY COMMANDS	7-1
	CODE-T: DISPLAYING A TRACE OF THE STACK	7-1
	CODE-U: DISPLAYING THE USER SCREEN	7-2
	CODE-Z: DISPLAYING THE CONTENTS OF THE 8087 REGISTERS	7-2
8	DEBUGGER MODES	8-1
	SIMPLE MODE	8-1
	MULTIPROCESS MODE	8-2
	INTERRUPT MODE	8-3
9	USING MULTIPROCESS MODE	9-1
	ENTERING MULTIPROCESS MODE VIA ACTION-B	9-1
	PROCEEDING (CODE-P) AND EXITING	9-1
	KEYBOARD AND VIDEO SWAPPING	9-1
10	SWAPPING, OVERLAYS, AND PORTS.....	10-1
	DEBUGGER SWAPPING	10-1
	EXAMINING CODE IN AN OVERLAY	10-2
	READING AND WRITING TO PORTS	10-2

11	THE HISTOGRAM FACILITY.....	11-1
	CODE-H: INVOKING THE HISTOGRAM FACILITY	11-1
	CODE-Q: QUERYING THE HISTOGRAM FACILITY	11-3
	TURNING OFF THE HISTOGRAM FACILITY	11-4
12	BREAKPOINTS IN INTERRUPT HANDLERS.....	12-1
	CODE-I: SETTING BREAKPOINTS IN INTERRUPT HANDLERS	12-1
	ASSEMBLY LANGUAGE CALLS:	
	THE INT 3 INSTRUCTION	12-2
13	ADVANCED DISPLAY COMMANDS.....	13-1
	CODE-N: DISPLAYING LINKED-LIST DATA STRUCTURES.....	13-1
	CODE-S: DISPLAYING THE STATUS OF PROCESSES AND EXCHANGES.....	13-2
14	OTHER ADVANCED COMMANDS	14-1
	CODE-K: DEACTIVATING THE DEBUGGER (KILLING THE DEBUGGER	14-1
	CODE-L: TURN LINE PRINTER ECHO ON (OFF)	14-1

APPENDIXES

Appendix A: STATUS MESSAGES	A-1
Appendix B: COMMAND SUMMARY	B-1
Appendix C: ALPHABETICAL LIST OF COMMANDS	C-1
Appendix D: THE DEBUG FILE UTILITY	D-1
Appendix E: OPERATING NOTES	E-1

GLOSSARY.....	Glossary-1
----------------------	-------------------

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
1-1. The Debugger in Relation to Other Steps in Executing a Program.....	1-3
1-2. The Debugger's Access to Other Software and Files.....	1-3
13-1. Example of Processes Displayed by CODE-S.....	13-2
13-2. Example of a Run Queue and Exchanges Displayed by CODE-S.....	13-3

LIST OF TABLES

<u>Table</u>	<u>Page</u>
4-1. Examples of Byte Patterns.....	4-7

GUIDE TO TECHNICAL DOCUMENTATION

This manual is one of a set that documents the Convergent™ Family of Information Processing Systems. The set can be grouped as follows:

Introductory

- Installation Guide
- Operator's Guide
- Executive Manual
- Context Manager Manual Status Codes Manual
- Installation Guide (NGEN) Operator's Guide (NGEN)

Hardware

NGEN

- Processor Manual: Model CP-001
- Dual Floppy Disk Manual
- Floppy/Hard Disk Manual
- Diagnostics Manual
- Keyboard Manual
- Power System Manual
- Monochrome Monitor Manual: Model VM-001
- Color Monitor Manual

IWS

- Workstation Hardware Manual
- Peripherals Hardware Manual
- IWS Peripherals Hardware Manual (SMD Version)

AWS

- AWS-210 Hardware Manual
- AWS-220, -230, -240 Hardware Manual
- AWS Color Workstation Hardware Manual

Operating System

- CTOS™ Operating System Manual
- System Programmer's Guide

Guest Operating Systems

- CP/M-86™
- MS™-DOS (and GW™-BASIC)
- XENIX™

Programming Languages

- COBOL Manual
- FORTRAN Manual
- FORTRAN-86 Manual
- BASIC Manual
- BASIC Compiler Manual
- Pascal Manual
- Assembly Language Manual

Program Development Tools

- COBOL Animator
- Editor Manual
- Debugger Manual
- Linker/Librarian Manual

Data Management Facilities

- CT-DBMS™ Manual
- ISAM Manual Forms Manual
- Sort/Merge Manual

Text Management Facilities

- Word Processing User's Guide
- Word Processing Reference Manual
- Word Processing Quick Reference

Applications Facilities

- Project Planner Manual
- CT-MAIL™ User's Reference Manual
- CT-MAIL™ Administrator's Reference Manual
- Multiplan
- Business Graphics User's Guide
- Business Graphics Reference Manual
- Graphics Programmer's Guide
- Font Designer Manual

Communications

- Asynchronous Terminal Emulator Manual
- 3270 Terminal Emulator Manual
- 2780/3780 RJE Terminal Emulator Manual
- SNA Network Gateway Manual
- SNA 3270 Emulator Manual
- X.25 Network Gateway Manual
- Multimode Terminal Emulator User's Guide
- Multimode Terminal Emulator Reference Manual

This section outlines the contents of these manuals.

INTRODUCTORY

The Installation Guide describes the procedure for unpacking, cabling, and powering up a system.

The Operator's Guide addresses the needs of the average user for operating instructions. It describes the workstation switches and controls, keyboard function, and floppy disk handling.

The Executive Manual describes the command interpreter, the program that first interacts with the user when the system is turned on. It describes available commands and discusses command execution, file management, program invocation, and system management. It also addresses status inquiry, volume management, the printer spooler, and execution of batch jobs. This manual now incorporates the System Utilities and Batch Manuals.

The Context Manager Manual describes and teaches the use of the Context Manager, which allows the user to run applications concurrently and interchange them on the screen almost instantly.

The Status Codes Manual contains complete listings of all status codes, bootstrap ROM error codes, and CTOS initialization codes. The codes are listed numerically along with any message and an explanation.

The NGEN Installation Guide describes the procedure for unpacking, assembling, cabling, and powering up an NGEN workstation.

The NGEN Operator's Guide is a link between the operator, the NGEN workstation, and the workstation's documentation. The Operator's Guide describes the operator controls and the use of the floppy disk drives, as well as how to verify that the workstation is operational and how to use software release notices.

HARDWARE

NGEN

The Processor Manual: Model CP-001 describes the Processor Module, which houses the Processor board, Memory board, I/O board, Video/Keyboard board, and Motherboard. It details the architecture and theory of operations of the printed circuit boards, external interfaces, and the Memory Expansion Cartridge, as well as the X-Bus specifications.

The Dual Floppy Disk Manual and the Floppy/Hard Disk Manual describe the architecture and theory of operation for the NGEN modules. They discuss the respective disk drives and controllers, and contain the applicable OEM disk drive manuals.

The Diagnostics Manual describes the diagnostics available for the NGEN workstation. It discusses the Processor Module's bootstrap ROM program and error codes, and individual software diagnostics for modules in the workstation.

The Keyboard Manual describes the theory of operation for the NGEN keyboard.

The Power System Manual describes the operation and connections for the 36-Volt Power Supply and the dc/dc converters used with the NGEN workstation.

The Monochrome Monitor Manual: Model VM-001 describes the operation and connections of the 12-inch Monochrome Monitor used with the NGEN workstation.

The Color Monitor Manual describes the operation and connections of the 15-inch Color Monitor used with the NGEN workstation.

IWS

The Workstation Hardware Manual describes the mainframe, keyboard, and video display for the IWS family of workstations. It specifies system architecture, printed circuit boards (Motherboard, Processor, I/O Memory, Multiline Communications Processor, Video Control, Graphics Control Board, ROM and RAM Expansions), keyboard, video monitor, Multibus interface, communications interfaces, power supply, and environmental characteristics of the workstation.

The Peripherals Hardware Manual describes the non-SMD single-board Mass Storage Subsystem (MSS) and Mass Storage Expansion (MSX) disk subsystems for the IWS family of workstations. It contains descriptions of the disk controller Motherboard, the two controller boards for floppy and Winchester disks, power supplies, disk drives, and environmental characteristics.

The IWS Peripherals Hardware Manual (SMD Version) describes the SMD MSS and MSX disk subsystems having one controller board.

AWS

The AWS-210 Hardware Manual describes the mainframe, keyboard, and video display of the AWS-210 workstation. It specifies architecture, theory of operation of the printed circuit boards (Motherboard, Deflection, and CPU), keyboard, video monitor, expansion interface, cluster communications interface, power supply, and environmental characteristics of the workstation.

The AWS-220, -230, -240 Hardware Manual describes the mainframe, keyboard, disk controllers, and video display of the AWS-220, -230, and -240 workstations. It specifies architecture, theory of operation of the printed circuit boards (Motherboard, Deflection, 8088 CPU, 8086 CPU, Floppy Disk Controller, and Hard Disk Controller), keyboard, video monitor, cluster communications interface, external interfaces, power supply, and environmental characteristics of the workstation.

The AWS Color Workstation Hardware Manual describes the mainframe, keyboard, and color video display of the AWS Color Workstation. This manual reports the architecture and theory of operation of the printed circuit boards (Motherboard, Graphics Control Board, Hard Disk Controller, Color Video, Color Deflection, and CPU), keyboard, color monitor, peripheral interfaces, cluster communications interface, power supply, and environmental characteristics of the workstation. This manual also contains four OEM disk drive manuals and a summary of adjustments for the color monitor.

OPERATING SYSTEM

The CTOS™ Operating System Manual describes the operating system. It specifies services for managing processes, messages, memory, exchanges, tasks, video, disk, keyboard, printer, timer, communications, and files. In particular, it specifies the standard file access methods: SAM, the sequential access method; RSAM, the record sequential access method; and DAM, the direct access method.

The System Programmer's Guide addresses the needs of the system programmer or system manager for

detailed information on operating system structure and system operation. It describes (1) cluster architecture and operation, (2) procedures for building a customized operating system, and (3) diagnostics.

GUEST OPERATING SYSTEMS

The CP/M-86™ and MS™-DOS Manuals describe the single-user operating systems originally designed for the 8086-based personal computer systems.

The GW™-BASIC Manuals describe the version of BASIC that runs on the MS™-DOS Operating System.

The XENIX™ Manuals describe the 16-bit adaptation of the UNIX system, including the XENIX environment for software development and text processing.

PROGRAMMING LANGUAGES

The COBOL, FORTRAN, FORTRAN-86, BASIC (Interpreter), BASIC Compiler, PASCAL, and Assembly Language Manuals describe the system's programming languages. Each manual specifies both the language itself and also operating instructions for that language.

The Pascal Manual is supplemented by a popular text, Pascal User Manual and Report.

The Assembly Language Manual is supplemented by a text, the Central Processing Unit, which describes the main processor, the 8086. It specifies the machine architecture, instruction set, and programming at the symbolic instruction level.

PROGRAM DEVELOPMENT TOOLS

The COBOL Animator describes the COBOL Animator, a debugger that allows the user to interact directly with the COBOL source code during program execution.

The Editor Manual describes the text editor.

The Debugger Manual describes the Debugger, which is designed for use at the symbolic instruction

level. It can be used in debugging FORTRAN, Pascal, and assembly-language programs. (COBOL and BASIC, in contrast, are more conveniently debugged using special facilities described in their respective manuals.)

The Linker/Librarian Manual describes the Linker, which links together separately compiled object files, and the Librarian, which builds and manages libraries of object modules.

DATA MANAGEMENT FACILITIES

The CT-DBMS™ Manual describes Convergent's data base management system (CT-DBMS), which consists of (1) a data manipulation language for accessing and manipulating the data base and (2) utilities for administering the data base activities such as maintenance, backup and recovery, and status reporting.

The ISAM Manual describes both the single- and the multiuser indexed sequential access method. It specifies the procedural interfaces (and how to call them from various languages) and the utilities.

The Forms Manual describes the Forms facility that includes (1) the Forms Editor, which is used to interactively design and edit forms, and (2) the Forms run time, which is called from an application program to display forms and accept user input.

The Sort/Merge Manual describes (1) the Sort and Merge utilities that run as a subsystem invoked at the Executive command level, and (2) the Sort/Merge object modules that can be called from an application program.

TEXT MANAGEMENT FACILITIES

The Word Processing User's Guide introduces the Word Processor to the first-time user. It provides step-by-step lessons that describe basic word processing operations. The lessons show how to execute operations and apply them to sample text.

The Word Processing Reference Manual is a reference tool for users already familiar with the Word Processor. It describes the Word Processor keyboard and screen; basic, advanced, and programmer-specific operations; list processing; printer and print wheel configurations; and hardware considerations.

The Word Processing Quick Reference provides a concise summary of all word processing operations and briefly describes the keyboard and commands.

APPLICATIONS FACILITIES

The Project Planner schedules and analyzes tasks, milestones, and the allocation of resources in a project. By means of diagrams and several kinds of bar charts, Project Planner presents time and resource allocation results and shows the occurrence of project milestones. The Project Planner Manual explains the use of the program and also serves as a reference once the user is familiar with it.

The CT-MAIL™ User's Reference Manual introduces the first-time user to the CT-MAIL electronic mail system. It provides step-by-step instructions for using the basic CT-MAIL operations to create, send, and receive mail.

The CT-MAIL™ Administrator's Reference Manual provides the System Administrator with instructions for installing, configuring, and maintaining the CT-MAIL electronic mail system; setting up communication lines; creating and maintaining mail centers; adding mail users; creating distribution lists; and troubleshooting.

Multiplan is a financial modeling package designed for business planning, analysis, budgeting, and forecasting.

The Business Graphics User's Guide introduces Business Graphics to the first-time user. It provides step-by-step lessons that describe basic Business Graphics operations. The lessons show how to execute operations and apply them to sample charts.

The Business Graphics Reference Manual is a reference tool for users already familiar with

Business Graphics. It describes the Business Graphics keyboard and screen; box and arrow cursor movement; obtaining information from Multiplan; operations; and plotter configurations.

The Graphics Programmer's Guide is a reference for applications and systems programmers. It describes the graphics library procedures that can be called from application systems to generate graphic representations of data, and it includes a section on accessing Business Graphics from an application system.

The Font Designer Manual describes the interactive utility for designing new fonts (character sets) for the video display.

COMMUNICATIONS

The Asynchronous Terminal Emulator Manual describes the asynchronous terminal emulator.

The 3270 Terminal Emulator Manual describes the 3270 emulator package.

The 2780/3780 RJE Terminal Emulator Manual describes the 2780/3780 emulator package.

The SNA Network Gateway Manual describes the SNA Network Gateway, which supports data communications over an SNA network. The SNA Network Gateway comprises the Transport Service and Status Monitor. The Transport Service allows a Convergent workstation to function as cluster controller and forms the foundation for Convergent SNA products.

The SNA 3270 Emulator Manual describes the SNA 3270 emulator package. The SNA 3270 emulator provides CRT and printer subsystems in addition to a Virtual Terminal Interface for use in application programs.

The X.25 Network Gateway Manual describes the X.25 Network Gateway, which supports CCITT Recommendation X.25 communications over a public data network. There are three levels of access to the network: packet, X.25 sequential access method, and the Multimode Terminal Emulator X.25 communications option.

The Multimode Terminal Emulator User's Guide introduces the Multimode Terminal Emulator to the first-time user. It describes the MTE video display, keyboard, display memory, and advanced operations for the X.25 communications option.

The Multimode Terminal Emulator Reference Manual is a reference tool for sophisticated users of the Multimode Terminal Emulator. It describes the MTE escape sequences and field verification program.

CP/M-86 is a trademark of Digital Research.

MS, GW and XENIX are trademarks of Microsoft Corp.

UNIX is a trademark of Bell Laboratories.

SUMMARY OF CHANGES

This third edition (A-09-00320-01-A) of the Debugger Manual documents release 9.0 of the Debugger and replaces the second edition (A-09-00011-01-C). The major changes to the content and organization of the Debugger Manual are described below.

A new command, CODE-O, enables you to search for a byte pattern in memory. This new feature is described in Section 4, "Examining and Changing Memory Contents."

A histogram facility has been added to the Debugger. By using CODE-H and CODE-Q commands, you can record how many times specified sections of code are entered and display this information. Section 11, "The Histogram Facility" describes this new feature in detail.

Besides documenting these new features, this edition of the Debugger Manual is organized to provide easy access to the different Debugger commands. The commands are described functionally in the main sections of the manual. The functions are arranged in order of complexity and frequency of use. Examples of command entries and interpreted Debugger displays are included in the command descriptions.

In addition, Appendix B, "Command Summary", gives a functional summary of each command. Appendix C, "Alphabetical List of Commands", lists the commands alphabetically and provides page number references to the sections of the manual where each command is explained in detail.

A glossary has been included at the end of this edition.

1 OVERVIEW

INTRODUCTION

The Debugger is a software tool that works in real time and lets users perform the following tasks:

- o examine and change data stored in memory or in registers
- o set and clear absolute breakpoints and conditional breakpoints
- o produce formatted displays of memory contents
- o search memory for a pattern of bytes
- o read and write to I/O ports
- o execute program instructions one at a time (single step)
- o assemble symbolic user input into binary machine instructions
- o disassemble the contents of memory into assembly language source instructions

You perform these tasks by entering commands, sometimes accompanied by parameters. This manual describes all of the Debugger commands and explains how to use them.

The Debugger runs concurrently with other user and system processes, and responds to commands as you type them. The parameters of Debugger commands can include numeric literals and fundamental 8086/80186 symbols, and also symbols defined in the program being debugged.

You can use the Debugger with programs written in assembly language, compiled BASIC, FORTRAN, FORTRAN-86, Pascal, and PL/M.

WHO SHOULD USE THIS MANUAL

This manual is intended for programmers who are familiar with assembly language, and who have some experience in debugging software.

Because the Debugger displays instructions of the Intel 8086/80186 microprocessor, users should be familiar with the 8086/80186 instruction set.

HOW TO USE THIS MANUAL

The sections in this manual fall roughly into two groups. Sections 1 through 7 describe elementary debugging procedures, such as working with memory contents, using breakpoints, working with registers, and using fundamental display commands. These are the procedures used most often in debugging.

Sections 8 through 14 describe more advanced features of the Debugger, including its three modes of operation, how it manages memory, working with overlays and ports, the histogramming facility, the use of breakpoints in interrupt handlers, and several infrequently used display commands.

If you are new to debugging, we recommend that you practice using the procedures explained in the first half of this manual, possibly in the presence of someone who is familiar with debugging. (Although reading this manual cover-to-cover before you start debugging a program may be useful, you will probably find that you learn more quickly by using the manual during actual debugging sessions.)

If you are familiar with general debugging procedures, you can use this manual as a reference guide. In particular, the appendixes provide a further distillation of the information in this manual, including a list of error messages, an explanation of the Debug File utility, a summary and an alphabetical list of all commands, and operating notes for using the Debugger with cluster systems and with dual floppy-disk systems. Appendix C also refers you to the page on which each command is described in detail.

Figures 1-1 and 1-2 illustrate the relationship between the Debugger and other elements of your system. Specific information about the Debugger begins in Section 2, "Concepts."

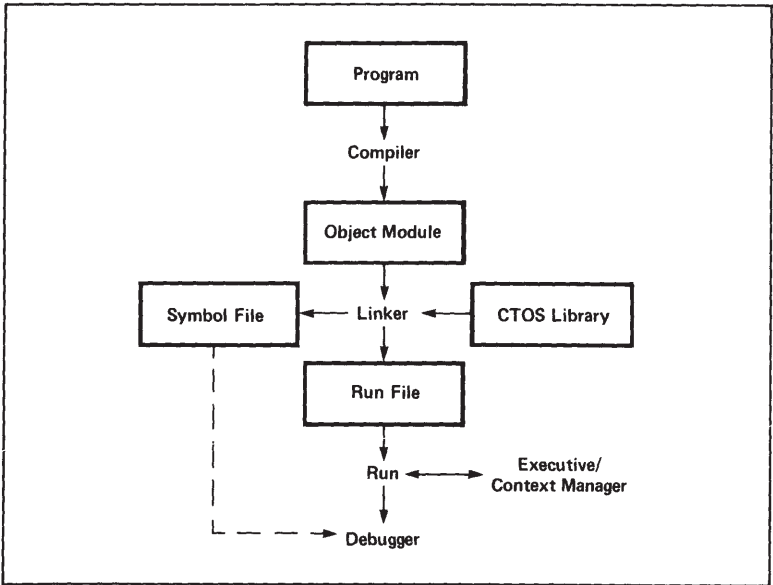


Figure 1-1. The Debugger in Relation to Other Steps in Executing a Program.

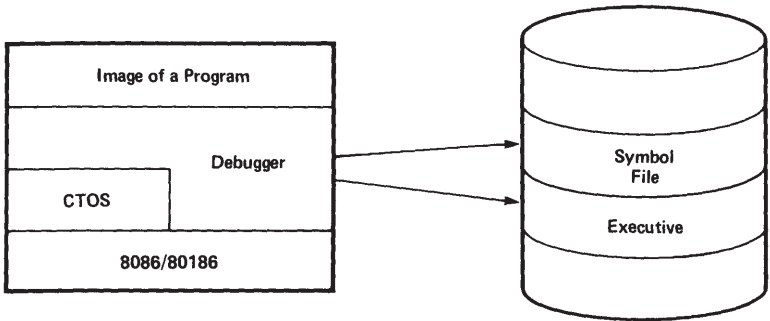


Figure 1-2. The Debugger's Access to Other Software and Files.

CONVENTIONS USED IN THIS MANUAL

- o Commands that consist of arrows (up, down, right, and left) are shown in the text of this manual in the following way:

↑

↓

→

←

- o Commands that use the CODE key together with another key are shown in the following way:

CODE-[the name of the key]

- o Memory addresses are indicated in the following way:

addr

- o Names of files are indicated in the following way:

'filename'

- o The solid right triangle (which appears on the video screen after the user enters a command) is indicated in the following way:

▴

2 CONCEPTS

This section describes the concepts and terminology that are fundamental to understanding how the Debugger functions.

COMMAND PARAMETERS

All Debugger commands have the same format: from zero to three typed-in parameters, followed by one command keystroke. Some commands are entered by holding down the CODE key and pressing another key at the same time. These commands are represented in the format CODE-X, where X is the value of the second key. The following examples illustrate the different command formats:

→	a command with no parameters
CODE-R	a command with no parameters
38DD CODE-B	a command with one parameter
@10, @100 CODE-H	a command with two parameters
112A, 2F20, "'ABC'" CODE-O	a command with three parameters

The command keys are listed below:

- o the up arrow ↑
- o the down arrow ↓
- o the left arrow ← or CODE-←
- o the right arrow → or CODE-→
- o MARK or CODE-MARK
- o RETURN
- o the equals sign (=)

- o CODE-[an alphabetic key]
such as CODE-C

The Debugger accepts parameters similar to the parameters permitted in assembly language. The acceptable Debugger parameters are indicated below:

- o constants (numbers, ports, and text)
- o symbols
- o composite parameters formed using parentheses
- o the minus sign (-)
- o the PTR (pointer) operator, which indicates the type of operand you are dealing with, as shown in the following examples:

```
MOV BYTE PTR [14],2
MOV WORD PTR [14],2
```

In these examples, BYTE PTR points to a byte and WORD PTR points to a word. If the type of the operand is not implied, you must use the PTR operator. Never use PTR alone, but always with BYTE or WORD.

- o the following binary operators:
 - + (addition)
 - (subtraction)
 - * (multiplication)
 - / (division)
- o address expressions, such as those shown below:

```
[BX]
[BP][DI+3]
```
- o symbolic instructions, such as those shown below:

```
PUSH BP
MOV BP,SP
SUB SP,4
```

CONSTANTS (NUMBERS, PORTS, AND TEXT)

NUMBERS

A number, or numeric constant, consists of digits belonging to the set of hexadecimal digits (0 through 9 and A through F) . The number can end with a decimal point (.) or with an "h".

A number that ends with a decimal point is a decimal number. A number that ends with an "h", or that omits both the decimal point and the "h", is a hexadecimal number.

A number that begins with the characters A through F must have a leading zero (0) preceding the character. Examples of numbers appear below:

```
123h    is a hexadecimal number
123     is a hexadecimal number
123.    is a decimal number
0AF     is a hexadecimal number
AF      is not a valid number
```

PORTS

A port constant is a number followed by the character "i" or "o". An "i" indicates that the port is an 8086/80186 hardware input port. An "o" indicates that the port is an output port. Examples of port constants appear below:

```
12i     is the input port that has port
        address 12
0A2o    is the output port that has port
        address 0A2
```

The Debugger can both read and write an input port constant. However, the Debugger never reads an output port constant. The ← and → commands open output ports for modification without reading them.

TEXT

A text constant is a sequence of characters enclosed by single quotation marks. To include a single quotation mark in a text constant, you should type another single quotation mark in front of the mark to be enclosed. For example,

'abed' is the four-character text constant
"abed"

'a' is the two-character text constant
"a" (consisting of a single
quotation mark and the letter "a")

Text constants consisting of one or two characters can be used wherever a number (a numeric constant) can be used. However, text constants consisting of more than two characters can be used only with the CODE-F and CODE-O commands (described in sections 3 and 4, respectively).

SYMBOLS

A symbol is a sequence of alphanumeric and special characters. A symbol must begin with an alphabetical character; it cannot begin with a digit.

The special characters are the underscore (_), the period (.), the dollar sign (\$), the percent sign (%), the pound sign (#), and the exclamation mark (!).

The Debugger recognizes the following four types of symbols:

- o User-defined public symbols, such as those in a symbol file produced by the linker from a source program. (See the Linker/Librarian Manual.)
- o Standard 8086/80186 register mnemonics, such as AX, BL, and SI.
- o Names of Debugger state variables, such as the process register PR. (See Section 6 below.)
- o The period (.) indicates the value of the segment and offset for the most recently opened location.

For examples of symbols, see the "Address Expressions" subsection below.

ADDRESS EXPRESSIONS

Address expressions in the Debugger have the same structure and semantics as address expressions in assembly language. Examples of address expressions appear below:

RgParam	represents the simplest address expression (a symbol)
RgParam+(100/2)	is a more complex address expression, involving a symbol
[BX+5]	is an indexed address expression
ES:[BX+5][SI]	is a doubly indexed address expression having a segment-override prefix

SYMBOLIC INSTRUCTIONS

Symbolic instructions in the Debugger have the same structure and semantics as they do in assembly language, except that symbolic instructions cannot include user-defined public symbols. Examples of symbolic instructions appear below:

```
MOV AX, WORD PTR [BX+5]

LOCK INC [BX]
```

CURRENT VALUE

The Debugger remembers a special value known as the current value. The current value is either the value most recently displayed by the Debugger, or the value that you typed most recently. To display the current value again, type an equals sign (=).

You can also use this instruction to display that value in a different number system, provided you first change the radix by entering a CODE-R

command. (See the example in Section 3, subsection "CODE-R: Changing the Number Base.")

DEBUGGER MODES

The Debugger operates in one of three different modes depending on how you enter it. Each of these modes (simple mode, multiprocess mode, and interrupt mode) is described in detail in Section 8, "Debugger Modes."

3 GETTING STARTED WITH THE DEBUGGER

This section explains how to enter and exit from the Debugger, how to use the Debugger as a calculator, how to open symbol files, and how to change the base of the number system in which the Debugger displays output data.

Some versions of the operating system are configured without the Debugger. If the Debugger is not present in your OS, and if you or a process tries to enter the Debugger, then the operating system either places an error message on the screen, or else sounds the audible alarm.

ENTERING THE DEBUGGER

You can enter the Debugger in any one of the following five ways:

1. When you press the ACTION key and hold it down while you press the A key (a procedure indicated throughout this manual as "ACTION-A").

This is the most commonly used way to invoke the Debugger. In this mode, the Debugger suspends all user processes.

2. When you press the CODE key and the GO key at the same time, after typing an Executive command or when using the Context Manager.
3. When a process reaches a previously placed breakpoint.
4. When you press the ACTION key and the B key at the same time. In this mode, the Debugger suspends only the user processes that reach a breakpoint; all other user processes that are running continue to run.
5. When a process executes a DEBUG instruction (INT 3, advanced debugging).

The Debugger interprets keystrokes in a way that minimizes accidental invocations of the Debugger or termination of the process being executed. For example, pressing the ACTION key has no effect unless you press one of three other keys (A, B, or FINISH) simultaneously.

DEBUGGER PROMPTS

Whenever you are using the Debugger, the screen shows your most recent dialogue with the Debugger, and also shows a Debugger prompt. The Debugger prompt is an asterisk (*), a pound sign (#), a space (), a percent sign (%), or an exclamation mark (!), or a greater than sign (>), depending on the type of debugging you are doing. These prompts are discussed in more detail in subsequent sections of this manual. However, their meanings are listed below:

- * the Debugger has suspended the current process
- # the Debugger has not suspended the current process
- ! the Debugger is at an interrupt level
- % the Debug File utility is in control
- > the system has abnormally terminated

DEBUGGER DISPLAYS

When the Debugger becomes active because a process has reached a breakpoint, the Debugger displays a description of the break. This description consists of the breakpoint address, together with the number of the process that has been suspended. Then the Debugger displays the Debugger prompt and waits for commands. While waiting, the Debugger treats all keyboard input as part of its command input.

EXITING FROM THE DEBUGGER

To leave the Debugger, press the GO key. The Debugger responds by restoring the screen that was present before you began using the Debugger.

After you press GO to leave the Debugger, the operating system directs all keyboard input toward a user process.

You can also press ACTION-FINISH to terminate the current program and invoke the Exit Run File.

USING THE DEBUGGER AS A CALCULATOR

You can use the Debugger as a calculator at any time. To do so, press ACTION-A, enter an expression to be evaluated, and then type an equals sign (=). For example, if you type

```
3*7=
```

the Debugger returns 15 (hexadecimal.)

You can also use the calculator mode to change the number base in which the Debugger expresses values. For example, if you type

```
800=
```

the Debugger returns 800, which simply indicates that the hexadecimal value 800 is equal to itself. To obtain a display of this value in decimal notation, type

```
10. CODE-R
```

and then

```
=
```

In this case, the Debugger returns 2048, which is the decimal equivalent of 800h

For more information about CODE-R, see the "CODE-R: Changing the Number Base" subsection below.

CODE-F: OPENING A SYMBOL FILE

The Linker produces symbol files that contain the addresses of public symbols. (See the Linker/Librarian Manual for more information about the Linker.)

One symbol file usually exists for each run file or for each user task. The Debugger cannot refer to a symbol file unless you open the symbol file first. To open a symbol file, type

```
'filename' CODE-F
```

A file name consists of text constants and must be enclosed in single quotation marks. For example, suppose you want to debug a program called

"Graph.Run". If the corresponding symbol file is "<Jones>Graph.Sym", then you would type

```
'<Jones>Graph.Sym' CODE-F
```

The CODE-F command opens the symbol file for your program, and gives you access to the public symbols in the symbol file.

The Debugger can refer to only one symbol file at one time. When you type 'filename' CODE-F, the Debugger ignores any previously opened symbol files.

A user program usually has only one symbol file. It is good practice to open the symbol file when you begin a debugging session; thereafter, you can use its symbols freely until you terminate the program or end the debugging session.

When a symbol file is open, the Debugger uses it to provide symbolic names for addresses. For example, with a suitable symbol file the instruction

```
CALL 0FFEF:336
```

might appear as

```
CALL ErrorExit
```

To suppress symbolic output, press

```
CODE-F
```

When you do so, the following message appears:

```
Symbols OFF
```

To enable symbolic output again, press

```
CODE-F
```

When you do so, the following message appears:

```
Symbols ON
```

The CODE-F command suppresses only symbolic output. You can use symbolic names as input any time you are using a symbol file.

When debugging a program that was not loaded by the Executive or the Context Manager, you cannot open the symbol file unless you specify the program segment address of the task, as shown below:

```
segment, 'filename' CODE-F
```

For example, to debug the program 'Fred', whose base address is 12340h (paragraph 1234h, for the 8086/80186 microprocessor), type

```
1234, 'Fred.Sym' CODE-F
```

Otherwise, however, you need not specify the program base address, because the Debugger can determine it by making a call to the operating system.

If you are debugging the operating system, you must type zero, the base address of the operating system, before you type the name of the symbol file, as shown in the example below:

```
0, 'OS.Sym' CODE-F
```

CODE-R: CHANGING THE BASE OF THE NUMBER SYSTEM

The CODE-R command changes the output radix. The output radix is the base of the number system in which information is expressed. Decimal, hexadecimal, octal, or any other base from 2 to 16 can be used.

All memory data is displayed using the radix that is in effect at that time. Unless you change it, this radix is hexadecimal.

To set the output radix to another base, type

```
k CODE-R
```

where k is a number from 2 to 16 (decimal.)

To change the output radix back to hexadecimal from any other base, type

```
16. CODE-R
```

or else simply type

```
CODE-R
```

If, however, the output radix is already hexadecimal, and you type CODE-R alone with no parameters, the output radix changes to decimal.

You can also use the CODE-R command to display the current value. To do so, type an equals sign (=) after you specify the base in which you want values to be displayed. (In this case, the equals sign specifies the most recent value.)

For example, if you want the hexadecimal value 100 displayed in decimal notation, you would enter

```
10. CODE-R
```

```
100=
```

The Debugger responds by displaying the decimal value 256.

NOTE

The output radix applies only to numbers that the Debugger displays. Numeric constants that you enter are interpreted or evaluated independently of the output radix, as explained in the "Numbers" subsection in Section 2, "Concepts."

LOOKING AT MEMORY

After entering the Debugger, you can examine the contents of memory by typing either a parameter that designates a machine address, a register, or an internal Debugger register, followed by a command (either ← , → or MARK) . The Debugger displays the contents of the designated address or register, and opens that address or register so that you can change its contents.

- o To display a single byte, type

addr ←

- o To display a single word, type

addr →

- o To display a symbolic instruction, type

addr MARK

For example, to display one byte at address 0A1, type

0A1 ←

The Debugger will return one byte of data, such as

1F

To display one word starting at register location DS:100, type

DS:100 →

The Debugger will return one word of data, such as

1F20

To display the instruction at "CreateISAM+10", type

CreateISAM+10 MARK

The Debugger returns the instruction at that symbolic address. For example, the MARK command might return

```
▲  MOV AX, WORD PTR  [BX]
```

Notice that when you press MARK, a small right triangle (▲) appears on the screen to the right of what you typed.

USING POINTERS TO DISPLAY MEMORY CONTENTS

You can also use indirect addresses to examine bytes, words, and instructions. You do so by specifying the address of a long pointer that addresses the byte, word, or instruction that you want to examine.

To display a single byte that is addressed by a long pointer, type

```
addr of byte pointer  CODE←←
```

To display a single word that is addressed by a long pointer, type

```
addr of word pointer  CODE→→
```

To display a symbolic instruction that is addressed by a long pointer, type

```
addr of instruction pointer  CODE-MARK
```

After you enter the CODE←←, CODE→→, or CODE-MARK command, the open location is the location addressed by the pointer.

For example, suppose you wish to display the byte addressed by a pointer at location 244. You could first fetch the pointer, and then fetch the byte.

```
244  
246  
2199:04AE      00
```

or you could simply use the CODE←← command.

```
244  CODE←←  00
```

DISPLAYING SEVERAL LOCATIONS AT ONCE

You can also use the memory-examination commands to display several locations at once. To do so, when you type the parameter, precede it with a number indicating how many locations you want displayed. In response, the Debugger displays the specified number of parameters, and keeps the last parameter open.

For example, to display three words starting with the word that begins at location DS:100, type

```
3, DS:100→
```

A typical Debugger response appears below:

```
3, DS:100 → 1F20
0AF:102 → 2F30
0AF:104 → 30FA
```

DEBUGGER PROMPTS

The Debugger always prompts you when it is ready for more input. The type of prompt depends on the open location, as explained below:

- o If no location is open (which happens if the Debugger was just entered, or if the previous command closed all of the open locations), then the Debugger issues the prompt that corresponds to the circumstances. (See the "Debugger Prompts" subsection in Section 3, "Getting Started with the Debugger.")
- o If a location is open, then the Debugger prompts you with an empty space (). This prompt appears on the same line as the value of the open location.

CODE-D: DISPLAYING THE CONTENTS OF MEMORY

The CODE-D command lets you display the contents of memory, in both numerical and ASCII form. CODE-D takes two parameters, and can display a number of bytes of memory, along with ASCII equivalents of those bytes, limited only by the amount of memory present in the machine. The Debugger displays the memory content in columns.

For example, to display a specific number of bytes of memory starting at a specific memory location, type

```
k, addr CODE-D
```

where k is the number of bytes, and addr is the memory location

For example, to display 9 bytes of memory starting at memory location 38DD, type

```
9, 38DD CODE-D
```

A typical Debugger response is shown below:

```
38DD    18 83 C3 52 80 3F 00 74 05  =C|${m}n@[
```

The material displayed at the right of this response indicates the ASCII characters that these bytes represent.

The Debugger automatically turns off the symbolic display of addresses while memory contents are being displayed.

CHANGING THE CONTENTS OF A MEMORY LOCATION

When a location is open and the Debugger prompts you with a space, you can change the contents of that location by typing the new contents that you want.

For example, suppose that you want to change the contents of DS:101 from 2F30 to 2F37. Location DS:101 is open, and the Debugger prompts you with a space. You type the right arrow (→) shown below:

```
DS:101→
```

The Debugger responds by displaying

```
DS:101 → 2F30
```

After which you type

```
2F37
```

Location DS:101 now contains the value 2F37.

CHANGING INSTRUCTIONS

Remember that assembly language instructions on the 8086/80186 processors have different lengths. Therefore, an instruction can include bytes of memory that are located beyond the bytes that the Debugger has displayed.

A Debugger command that replaces an instruction can also leave the last few bytes of the original instruction dangling after the end of the new instruction. In such a case, when the new instruction is shorter than the original one, you should replace each dangling byte of the original instruction with a no operation (NOP) instruction. The NOP instruction acts as a place-holder.

For example, suppose a comparison instruction is three bytes long and the jump instruction is two bytes long:

```
24D:120  ─  CMP AX, WORD PTR[BP+4]  JMP +2
```

```
24D:122  ─  ADD AL,75  NOP
```

As shown above, when you write the jump instruction over the comparison instruction, you must insert a NOP instruction after the jump instruction.

OPENING A NEW LOCATION

The memory location that is open changes with each command that modifies the contents of memory. There are three such commands:

- o RETURN
- o ↑
- o ↓

If you press RETURN, the previously open location closes and no new locations are opened.

If you press ↑, the previous location opens.

If you press ↓, the next location opens.

The Debugger interprets the words "next" and "previous" according to the mode in which a location is open. For example, "next" can refer to the next byte, the next word, or the next instruction.

CODE-O: SEARCHING FOR A BYTE PATTERN IN MEMORY

The CODE-O command searches for a byte pattern in memory. A byte pattern is a user-defined group of byte specifiers separated by commas and enclosed in double quotation marks. A byte specifier is either a sequence of two-digit hexadecimal numbers, or a string of characters enclosed in single quotation marks. Examples of byte patterns appear in Table 4-1 below.

To search for a given byte pattern, type

```
lower addr, upper addr, byte pattern CODE-O
```

For example, to search for the byte pattern 30,31 within the range of addresses from 5FE6:0C to 5FE6:100, type

```
5FE6:0C, 5FE6:100, "30,31" CODE-O
```

The Debugger searches for a byte pattern within the range of addresses starting at lower addr and ending at upper addr. If the pattern is found, then the Debugger displays the pattern at the address at which it was found, and changes lower addr to the address of the first byte following the pattern.

To make the Debugger continue the search beginning at the new lower addr, type

```
CODE-O
```

with no parameters.

In either case, if the Debugger does not find a byte pattern, then the Debugger ends the search when it reaches upper addr.

Table 4-1. Examples of Byte Patterns.

<u>Byte Pattern</u>	<u>Pattern Specified</u>
"30,31,32"	123
" 'ABC' "	ABC
"30,31,32, 'ABC',33,34,35"	123ABC456

5 USING BREAKPOINTS

A breakpoint is a user-defined location in code. When a process reaches a breakpoint, the process is suspended, and the Debugger is entered.

If the Debugger is operating in simple mode, all user processes are suspended whenever any breakpoint is taken. If the Debugger is in multi-process mode, only the process that has taken the breakpoint is suspended. (See Section 8 for a detailed description of the Debugger's operating modes, and Section 12 for an explanation of OS debugging using CODE-I to set breakpoints in interrupt handlers.)

CODE-B: SETTING AND QUERYING BREAKPOINTS

To set a breakpoint, type CODE-B, preceded by one parameter. For example, to set a breakpoint at the address addr, type

```
addr CODE-B
```

You can enter CODE-B to set a breakpoint in an overlay, even if the overlay is not present in memory.

A breakpoint stays in effect until you remove it explicitly (by entering the CODE-C command, described below) or until the process terminates.

When a program in a memory partition calls CHAIN or EXIT, or is otherwise terminated, only the breakpoints in that partition are removed.

To obtain a display of a list of all of the breakpoints that are set at any given time, type

```
CODE-B
```

without an address parameter.

This activity is known as querying the breakpoints.

CODE-C: CLEARING BREAKPOINTS

To clear a breakpoint, type a parameter, followed by CODE-C. For example, to clear the breakpoint at address `addr`, type:

```
addr CODE-C To clear all of the breakpoints at once, type
```

CODE-C without an address parameter.

CODE-A: SETTING CONDITIONAL BREAKPOINTS

A conditional breakpoint is a breakpoint that is associated with a relational condition. When a process reaches a conditional breakpoint, the process is suspended only if the relational condition is evaluated as TRUE.

To set a conditional breakpoint, type a parameter and CODE-A. Two examples appear below:

To set a breakpoint at physical address `0E0:3B1`, type

```
0E0:3B1 CODE-A
```

To set a conditional breakpoint at symbolic address `Accept+23`, type

```
Accept+23 CODE-A
```

You define the relational condition by entering code into the Debugger's patch area. The patch area is a 50-byte space addressed by the symbol PatchArea. To enter code, type

```
PatchArea MARK
```

The Debugger then displays the first instruction in the patch area. For example, a typical Debugger response appears below:

```
PatchArea ▶ NOP
```

At the end of this line, the Debugger displays the space () prompt. You can then add your instruction. To display and modify the next instruction in the patch area, type your instruction and then press ↓.

For example, in response to the line displayed above by the Debugger, you could type

```
MOV AX, WORD PTR [0] ↓
```

The Debugger then displays the next instruction in the patch area:

```
PatchArea+3 ↓ NOP
```

You would follow the same procedure, entering the instruction and then pressing ↓, for this instruction and for each of the succeeding ones in a series.

For example, suppose you want to tell the Debugger to take the breakpoint if the value of memory location DS:0 is 200h. After all of your instructions are entered, the Debugger display looks like the example shown below:

```
PatchArea  ▲ NOP  MOV AX, WORD PTR [0]
PatchArea+3 ▲ NOP  CMP AX, 200
PatchArea+6 ▲ NOP  JE  .+5
PatchArea+8 ▲ NOP  MOV AL,0
PatchArea+0A ▲ NOP  DEBUG
PatchArea+0B ▲ NOP  MOV AL, OFF
PatchArea+0D ▲ NOP  DEBUG
```

Instructions that you add to the patch area must set the register AL to 0FFh if the breakpoint is to be taken. If the breakpoint is not to be taken, your instructions must set AL to 0h. The original value of the AX register is restored after the condition is evaluated.

So that control will return to the Debugger, the last instruction in the relational condition must be DEBUG (INT 3).

You can set more than one conditional breakpoint by specifying an additional parameter for the CODE-A command. This parameter specifies the PatchArea offset at which the relational condition begins.

For example, the following parameter and command

```
20, Initialize CODE-A
```

set a conditional breakpoint at the location named "Initialize", whose relational condition begins at PatchArea+20.

You can change an unconditional (CODE-B) breakpoint into a conditional breakpoint at any time, by typing

```
addr CODE-A
```

After entering this command, you must also add the conditional code in the PatchArea, as explained above.

CODE-P: PROCEEDING FROM A BREAKPOINT

To proceed from the most recently found breakpoint in the current process, type

```
CODE-P
```

The breakpoint remains in effect, and the process continues. If the process was not broken by the breakpoint, the Debugger ignores the CODE-P command. (In this case, because the process is still running, you cannot logically command it to resume running.)

To proceed, and to break the kth time the breakpoint is reached (instead of the next time it is reached), type

```
k CODE-P
```

where k is a decimal number.

(CODE-P with no parameters is equivalent to CODE-P with a parameter of 1.)

To remove the breakpoint before proceeding, type

```
0 CODE-P
```

If you entered the Debugger by pressing ACTION-A, you should enter the CODE-P command to exit from the Debugger. (Pressing GO has the same effect as pressing CODE-P; that is, it lets you proceed from the most recent breakpoint.)

CODE-G: STARTING A PROCESS AT A SPECIFIED ADDRESS

The foregoing commands always cause a process to start running from the last breakpoint address. To begin process execution at a different address (for example, at address addr), type

```
addr CODE-G
```

The address addr should be an expression that includes a user-defined public symbol (for example, "RgParam+5"), or else it should have the form indicated below:

```
x:y
```

where x is an appropriate CS parameter, and y is an appropriate IP parameter. (Each process has its own CS and IP registers in the 8086/80186 processors. These registers point to the instructions currently being executed in that process.)

The commands

```
10:0 CODE-G
```

and

```
100 CODE-G
```

both cause execution to begin at absolute address 100. However, the command "10:0 CODE-G" sets CS to 10, and the command "100 CODE-G" sets CS to 0.

CODE-X: EXECUTING INSTRUCTIONS INDIVIDUALLY

To make the system execute the next instruction in the current process, type

```
CODE-X
```

After the system executes this instruction, it opens and displays the next instruction.

Thus, you can type CODE-X again and again to see a series of instructions displayed and executed one by one.

To resume continuous execution of instructions after using CODE-X, either type

CODE-P

or press GO.

Whenever you use the CODE-X command to execute an instruction that loads a segment register, two instructions are actually executed.

For example, in the portion of code listed below, if you use CODE-X to execute the instruction "LES BX, [bp + 6]", the instruction "PUSH ES" is also executed, and the PUSH BX instruction is displayed as the open location, as indicated below:

```
LES BX, [bp + 6]
PUSH ES
PUSH BX
```

CODE-E: BREAKING AFTER THE CURRENT INSTRUCTION

To break after the current instruction, type

CODE-E

In most cases, entering CODE-E has the same effect as entering CODE-X: it lets you step over an instruction. However, if the instruction is a CALL, CODE-X executes the CALL instruction and then breaks at the first instruction in the called procedure.

In contrast, CODE-E executes the entire procedure and then breaks at the first instruction following the RETURN. Therefore, CODE-E provides a convenient way to step over a procedure call.

NOTE

You cannot use CODE-E to break after an instruction that loads a segment register.

6 WORKING WITH REGISTERS

This section describes the Debugger's internal process register, and explains how to examine and modify registers. It also discusses the use of register mnemonics, and tells how to proceed from a new Code Segment and Instruction Pointer (CS:IP) using CODE-G.

THE PROCESS REGISTER

The process register PR is a Debugger internal register. This register always identifies and keeps track of the current process. The PR is set automatically to the process identifier of the process that most recently reached a breakpoint. For example, if Process 2 is the process that most recently reached a breakpoint, then PR is set to 2.

When you invoke the Debugger from the Executive or from the Context Manager (using CODE-GO) just before the execution of an application system begins, PR is set to the identifier of the first user process.

All Debugger commands that involve processes treat the process with which they are concerned as the current process. For example, whenever registers are read or written to, the register of the current process is used.

EXAMINING AND MODIFYING REGISTERS

To debug a single-process program, you normally do not need to refer to PR at all. However, to debug a multiprocess program, you must know which of the processes involved is the current process. You can examine and change PR the same way you would examine and change any other word location: by opening it and then entering a new parameter, if desired. For example, if the current process is number 4 and you want to change it to number 7, type

PR →

In this case, the Debugger responds by displaying

PR → 4

to which you respond by typing

7

The current process is now number 7.

When a multiprocess program is debugged, the 8086/80186 register mnemonics indicate the machine registers that are associated with the current process. Likewise, the IP (instruction pointer) and FL (flags) registers indicate the instruction pointer and flags associated with the current process.

To examine the registers of other processes, you must change PR by following the procedure explained above.

USING REGISTER MNEMONICS

You can read and write to all of the registers listed below.

CS	Code Segment
DS	Data Segment
ES	Extra Segment
SS	Stack Segment

AX, BX, CX, and DX are all general registers. BP, BX, DI, and SI are general registers and index registers. The 16-bit FL register contains flags, and the IP register contains the instruction pointer.

You can use register mnemonic symbols (AX, SI, etc.) only as left-side values. In other words, these symbols must be the first parameters in the commands that you use to examine or to change the contents of memory.

NOTE

If you try to change the register SP, your system will abnormally terminate.

7 ELEMENTARY DISPLAY COMMANDS

This chapter explains how to use three basic Debugger commands to display a trace of the stack, to display the user screen, and to display the contents of the 8087 co-processor.

CODE-T: DISPLAYING A TRACE OF THE STACK

A stack trace consists of several numbered lines. Each line number corresponds to a stack frame, and the first line displayed corresponds to the current stack frame. The CODE-T command displays the procedure-invocation stack for the current process. To display the entire stack, type

```
CODE-T
```

To display the stack for the k most recent active procedure invocations, type

```
k CODE-T
```

For example, to display the stack for the 6 most recent invocations, type

```
6 CODE-T
```

A typical Debugger response is the display shown below:

```
0 3108 12D2:14A (3,3,4F1F,3113)
1 3116 581F:95 (173C,1,4F1F,312A,4F1F,312E)
2 3130 5EC8:0C4 (4F1F,173C,4F1F,314D)
3 313E 571C:255 (26F0,47,4F1F,314D,4F1F,3158)
4 3160 5920:6E (4F1F,2700,4F1F,3178,4F1F,31CD)
5 318E 5FA9:70 (2700,4E1F,0,600,4F1F,31CD)
```

In this display, the first column contains the level number for the current frame. The second column contains the frame pointer (BP) for that frame, and the third column contains the return address (CS:IP) that will be effective when control returns to that level. The remainder of each line contains the parameters that were passed to the procedure corresponding to the stack frame.

The Debugger estimates the number of parameter words displayed (up to a maximum of six). Therefore, this number might not correspond to the number of parameters actually passed to the procedure.

CODE-U: DISPLAYING THE USER SCREEN

When the Debugger is running, it displays only the Debugger screen. The Debugger does not display the screen generated by the user process. To view the user screen without exiting from the Debugger, type

CODE-U

Once the user screen appears, press any key (except CODE-U again) to restore the Debugger screen.

CODE-Z: DISPLAYING THE CONTENTS OF THE 8087 REGISTERS

This command displays the contents of the 8087 registers.

NOTE

You should use CODE-Z only if your system has an 8087 co-processor.

For example, suppose your system has only an 8086/80186 processor. If you enter CODE-Z, then your system will enter a perpetual WAIT state, because the 8087 co-processor is not there.

In such a case, the debugging process (as well as any other processes) will halt, and you will not be able to enter any instructions or commands from the keyboard. The only solution is to re-boot your system.

8 DEBUGGER MODES

The Debugger operates in three modes: simple mode, multiprocess mode, and interrupt mode.

- o The simple (and most often-used) mode applies to the debugging of a single-process user task, such as a compiled BASIC program.
- o The multiprocess mode applies to the debugging of a multiprocess user task whose operation depends on the continuous execution of all processes except the ones that are explicitly stopped at breakpoints.
- o The interrupt mode applies to the debugging of interrupt handlers, or else to debugging that requires that breakpoints be set in the operating system Kernel.

The following paragraphs describe how these three modes are related to the different ways of entering the Debugger.

SIMPLE MODE

You invoke this mode by pressing ACTION-A. This mode also occurs when a CODE-B breakpoint is executed after ACTION-A invokes the Debugger.

You can also enter the Debugger in simple mode by using the Chain or LoadTask operations of standard software. (See Section 7 of the CTOS Operating System Manual for details of this advanced procedure.)

In this mode, all user processes are suspended when you or the program enters the Debugger. This mode does not affect operating system services or interrupt handlers.

The nonresident portion of the Debugger can swap in and out of memory. However, this procedure requires no intervention by the user, and is not evident to the user. Simple mode requires approximately 55K bytes less physical memory than the multiprocess mode or the interrupt mode.

In simple mode, the Debugger prompt is an asterisk (*).

MULTIPROCESS MODE

You invoke this mode by pressing ACTION-B. This mode also occurs when a CODE-B breakpoint is executed after ACTION-B invokes the Debugger.

All user processes continue execution after you enter the Debugger. The Debugger suspends only those user processes that have reached a breakpoint.

This mode does not affect operating system services or interrupt handlers. It is most useful for debugging certain real-time programs, such as the timer process in the Executive.

If you invoke the Debugger by entering the ACTION-B command, then you can still type ACTION-A to invoke the Debugger in its ACTION-A mode, and suspend all user processes. However, once you invoke the Debugger in ACTION-A mode, you cannot change directly to the ACTION-B mode.

The Debugger alone requires about 55K bytes of memory. However, the entire Debugger (including its nonresident portion), together with the program being debugged, must fit into the available memory. If memory is insufficient, a status message is displayed, and the Debugger switches to simple mode.

Provided enough memory is available, you can set a CODE-I breakpoint in multiprocess mode at any time. The Debugger goes into interrupt mode when a CODE-I breakpoint is taken. (See Section 12 for more information about CODE-I breakpoints.)

If the current process as defined by the process register (PR) is not suspended, the Debugger prompt is the pound sign (#).

If the current process has been suspended, the Debugger prompt is an asterisk (*).

For more information about using multiprocess mode, see Section 9.

INTERRUPT MODE

This mode occurs when a user process reaches a CODE-I breakpoint. When the program reaches a CODE-I breakpoint, the Debugger takes control of the interrupt system and freezes the condition of the processor.

As in the multiprocess mode, the entire Debugger must fit into the available memory. If you set a CODE-I breakpoint and enough memory is not available, a status/error message appears.

In the interrupt mode, the Debugger prompt is an exclamation mark (!).

9 USING MULTIPROCESS MODE

This section provides more details about how to use the Debugger's multiprocess mode: how to enter the mode, how to proceed, and how to switch control of the keyboard and the video display between the user process and the Debugger.

ENTERING MULTIPROCESS MODE VIA ACTION-B

As noted in Section 8, you enter multiprocess mode by pressing ACTION-B. If you have invoked the Debugger using ACTION-B, then multiprocess mode is also entered when a CODE-B breakpoint is executed.

PROCEEDING (CODE-P) AND EXITING

In multiprocess mode, if the current process has been suspended, the CODE-P command causes that process to resume.

However, in multiprocess mode, the CODE-P command does not exit from the Debugger. (As noted above, you press CODE-P to proceed with the current process without exiting from the Debugger.) To exit, you must press GO.

Similarly, you press CODE-G to begin process execution at a different address from the one at which the process stopped. (Pressing CODE-G does not exit from the Debugger.)

KEYBOARD AND VIDEO SWAPPING

In multiprocess mode, pressing GO does not cause the currently interrupted process to proceed. Instead, in this mode, you press GO to exit the Debugger and return control of the screen and the keyboard to the user process.

10 SWAPPING, OVERLAYS, AND PORTS

This chapter describes how the Debugger manages memory, explains how to examine code in an overlay, and discusses how to read and write to ports.

DEBUGGER SWAPPING

The Debugger requires approximately 55K bytes of memory. However, under some circumstances (for example, if you are debugging the Word Processor), you can debug a program that occupies all of memory, theoretically leaving no room for the Debugger.

The Debugger manages memory according to the procedures described below:

- o If enough memory is available, the Debugger uses memory that is not used by the other processes.
- o If enough memory is not available, the Debugger "swaps out" part of the user's program, provided swapping out is possible.

When the Debugger swaps out part of the user's program, program execution is not affected.

Swapping can occur only when the Debugger is in simple mode. In simple mode, all user processes are suspended when they reach a breakpoint. (See also Section 8, "Debugger Modes.")

NOTE

The Debugger swapping mechanism is not related or connected in any way to the Virtual Code Segment Management Facility.

EXAMINING CODE IN AN OVERLAY

An overlay is a part of a program that remains on a disk until you call it using the CALL operation. The Debugger can display instructions that are contained in an overlay, even if the overlay is not present in memory. (For more information about overlays, refer to the "Virtual Code Segment Management" section in the CTOS Operating System Manual.)

To display instructions contained in an overlay, type

```
symbolic addr MARK
```

If the overlay is not present in memory, then the Debugger places braces around the symbolic address of the next instruction displayed using the DOWN-arrow command. For example, the message

```
[Initialize+2] MOV BP, SP
```

indicates that the code segment containing the procedure called "Initialize" is in a swapped-out overlay.

You cannot modify instructions that are contained in a swapped-out overlay. Patches that you make to swapped-in overlays stay in effect only while the overlay is present in memory.

READING AND WRITING TO PORTS

To read and write to ports, use the left-arrow and right-arrow commands together with constants.

For example, to read from the byte-input port 17i, type

```
17i ←
```

To read from the word-input port 31i, type

```
31i →
```

In either of these cases, after you type the constant, the port becomes an open location. You can then specify a new parameter to be written to the port. Note that unlike reading from memory, reading from a port can change the state of the system. For example, reading a character from the keyboard removes the character from the keyboard.

11 THE HISTOGRAM FACILITY

The Debugger's histogram facility lets you analyze a section of your program to see how often it executes given instructions. The facility does so by displaying a list of how often the program refers to one or more memory locations within a user-specified range. Thus, you can measure program performance based on this information about where a program spends its time.

To generate a histogram, the Debugger divides memory into small regions known as buckets. Buckets are contiguous; collectively, they span the entire section of code being analyzed. The Debugger maintains a set of counters for these buckets.

The Debugger creates a histogram by sampling the instruction pointer (CP:IP) 60 times per second. If CP:IP refers to an address that is located in the section of code being analyzed, the Debugger determines which bucket is referred to, and adds to the counter for that bucket.

CODE-H: INVOKING THE HISTOGRAM FACILITY

Before creating a histogram, you must specify the lower and upper bounds (addresses) of the section of code to be analyzed. To do so, enter the CODE-H command, preceded by two parameters that specify the lower and upper bounds, respectively, as shown below:

```
parameter, parameter CODE-H
```

For example, to specify line number 10 of a compiled BASIC program as the lower bound and line number 1000 as the upper bound, type

```
@10,@1000 CODE-H
```

(In BASIC programs the symbolic name of a line number is the line number preceded by an @ character.)

By default, the bucket size is 16 (decimal) bytes. To change the bucket size, enter CODE-H with one parameter. Note that the bucket size must be a multiple of 16 (decimal).

For example, to change the bucket size to 64 bytes, type

```
64. CODE-H
```

After setting the bounds and the bucket size, start the histogramming procedure by entering CODE-H with no parameters, as shown below:

```
CODE-H
```

The Debugger responds by displaying the message

```
[Histogram On]
```

before exiting to the program.

Note that CODE-H is equivalent to GO. When you enter CODE-H, the Debugger exits and the user program continues.

NOTE

To support histogramming, the debugger must allocate enough long-lived memory to contain the bucket counters.

If this much memory is not available, the debugger displays the message "Not enough memory." In this case, the histogram facility will not start. To avoid this situation, you must either reorganize your program so the Debugger can acquire the necessary memory or specify a larger bucket size.

You must reenter the Debugger to obtain a display of the histogram report. You can do so either by pressing ACTION-A or by placing a breakpoint in the program before you start the histogramming facility.

NOTE

Do not let your program call Exit (i.e., do not let it terminate) while histogramming is in progress. If you do so, the histogram information will be lost.

CODE-Q: QUERING THE HISTOGRAM FACILITY

Once you have entered the Debugger again, you can obtain a display of the histogram report by entering CODE-Q preceded by the parameter 1, as indicated below:

1 CODE-Q

A typical Debugger response is shown below:

```
*1 CODE-Q
@200+20      1
@300+2       1
@300+12      1
@400+0C      5
@600+6       1D
@600+16      0F
@600+26      9
@700+2       0E
@700+12     23
@700+22     0D
@700+32     11
@700+42     0E
@700+52      4
@800+7       7
```

```
In Range:      0A5
Out of Range:  6DB
Total:         780
```

This report displays the address of each bucket and indicates the value of the associated bucket counter. It also displays a summary indicating the total number of samples taken, the number of times that the instruction pointer was within the specified range, and number of times that the pointer was outside the range.

To obtain a display of the bounds and bucket size in the histogram, enter CODE-Q with no parameters, as shown below:

CODE-Q

Pressing CODE-Q also produces a report of the number of bytes required to contain the bucket counters, as shown in the following display:

```
Lower: PrimeEntry+9  Upper: @1000+1
Bytes/Bucket: 10     Bytes required: 0C0
```

If you enter either CODE-Q or 1 CODE-Q without having exited from the Debugger and run the histogram facility, the Debugger displays the message "No data." In this case, you must again enter the lower and upper bounds and the number of bytes per bucket, and then enter CODE-H or GO again to exit from the Debugger and run the histogram facility.

When you enter the Debugger again after a histogram has been displayed, you can continue histogramming simply by exiting from the Debugger using GO or CODE-P.

TURNING OFF THE HISTOGRAM FACILITY

To turn off the histogram facility, enter CODE-H again with no parameters, as shown below:

```
CODE-H
```

The Debugger responds by displaying the following message:

```
[Histogram Off]
```

After turning off the histogram facility, you can change the bounds and the bucket size and invoke the histogram facility again, as explained above.

12 BREAKPOINTS IN INTERRUPT HANDLERS

The breakpoint commands described in Section 5, "Using Breakpoints," let you place breakpoints in normal user code and in normal operating system code. However, these commands alone cannot place a breakpoint in the operating system kernel or in an interrupt handler.

CODE-I: SETTING BREAKPOINTS IN INTERRUPT HANDLERS

To set a breakpoint at address addr in an interrupt handler or in the OS kernel, type

addr CODE-I

The standard keyboard and video facilities of the operating system support your interaction with the Debugger, except when the OS kernel or an interrupt handler is broken. At these breakpoints, all processes (including OS processes) are suspended, and the Debugger then works by direct access to the physical keyboard and the screen.

CODE-I breakpoints are prohibited if the Debugger has swapped out a part of the user program. (The Debugger does this when the user program and the Debugger together are too large to fit in the partition.)

Before exiting from interrupt mode, you should explicitly remove any CODE-I breakpoints by entering the CODE-C command.

The Debugger uses hardware interrupts 1 and 3. Therefore, user programs should not use hardware interrupts 1 or 3. Other parts of the operating system use other hardware interrupts. Refer to the manual for your operating system to learn which of these other interrupts you should not use.

ASSEMBLY LANGUAGE CALLS: THE INT 3 INSTRUCTION

If you code an INT 3 instruction in an assembly language program, the system enters the Debugger when the INT 3 instruction is executed.

For example, if you code an INT 3 instruction at location 9904:6A, then when the program executes that instruction, the system enters the Debugger. In this case, the system displays the following type of message:

Debugger call at 9904:6B in Process 8

Note that the address displayed in this message (9904:6B) is located one byte after the INT 3 instruction (9904:6A).

The Debugger also disassembles the INT 3 instruction by displaying the DEBUG mnemonic, as shown below:

9904:6A ▀ DEBUG

13 ADVANCED DISPLAY COMMANDS

This section describes two commands that are used occasionally in advanced debugging. CODE-N displays linked-list data structures; CODE-S displays system structures for the status of processes and exchanges.

CODE-N: DISPLAYING LINKED-LIST DATA STRUCTURES

The CODE-N command displays linked-list data structures. You should use CODE-N with the internal Debugger registers CB and DB. The following paragraphs explain how to do so.

To display a block of memory that is k bytes long and that has a link word at the jth byte, you should first set CB equal to k, and then set DB equal to j. For example,

```
CB → 0000 8
```

```
DB → 0000
```

To display the first block, type

```
addr CODE-N
```

To display each subsequent block, again type

```
CODE-N
```

Or, to display n blocks at the same time, type

```
n, addr CODE-N
```

where n is a decimal number. For example,

```
3., 1217:3084 CODE-N
```

specifies that three blocks are to be displayed at once. A typical Debugger response appears below:

```
1217:3084 02 31 C1 01 7C 3E AF 17
1217:3102 DA 31 83 80 68 26 76 5D
1217:31DA 00 00 C1 FF 04 4C 17 12
```

CODE-S: DISPLAYING THE STATUS OF PROCESSES AND EXCHANGES

Processes communicate with one another by sending messages to exchanges. The two primitives used are SEND and WAIT. An exchange can be occupied by a message or by a process that is waiting to receive a message. The CODE-S command displays the status of processes and exchanges. To obtain a display of all of the processes and exchanges, type

CODE-S

When you enter CODE-S, the Debugger displays an entire screen of information. The first item is a listing of processes, as shown in Figure 13-1 below:

Processes												
id	oPcb	cs:ip	link	st	pr	ss:sp	bp	ds	exch	user	oExtPcb	
00	3084	0045:3E7C	31DA	C1	01	17A9:3E4C	0000	0000	0002	0000	3184	
01	3096	1075:08F5	0000	C0	01	1217:5104	2796	1217	000D	0000	3196	
02	30A8	09DC:0091	0000	C0	02	1217:51D4	25DE	1217	0000	0000	31A8	
03	30BA	06FC:0155	0000	C0	03	1217:52FE	1F08	1217	0010	0000	31BA	
04	30CC	06F0:0024	0000	C0	03	1217:54FC	1C60	1217	000B	0000	31CC	
05	30DE	093D:006A	0000	C0	04	1217:55F6	246C	1217	0004	0000	31DE	
06	30F0	0C9C:003E	0000	C0	05	1217:56EE	5710	1217	0006	0000	31FE	
07	3102	12D2:014A	0000	82	80	4F1F:30D0	30EC	4F1F	001C	0000	3202	
08	3114	5257:009E	0000	82	7F	4F1F:1B3C	1B54	4F1F	001D	0000	3214	
13	31DA	082F:000B	0000	C1	FF	1217:4C04	0000	0000	0000	0000	320A	

Figure 13-1. Example of Processes Displayed by CODE-S.

Each line in this display corresponds to a process that is currently active in the system. The number at the beginning of each line is a process identifier (PID).

The column headings in this display are defined below:

- id The process identifier number.
- oPcb The address of the process control block for that process. The address is relative to the operating system's data segment.

cs:ip The address of the next instruction to be executed by the process.

link A link address used by CTOS to keep processes threaded.

st A byte containing status flags.

pr The priority of the process.

ss:sp The address of the top of the stack for the process.

bp The base pointer of the process.

ds The data segment of the process.

exch For that process, the default exchange for responses.

oExtPcb The address of the extended PCB.

The next items the Debugger displays are the Run Queue and the Exchanges. (See Figure 13-2 below.)

Run Queue	00 01 02 13
Exchanges	
	oExchg oMsgHead oMsgTail oPcbHead oPcbTail
01	31F4 3458 3458 0000 3084
03	3204 0000 0000 30DE 30DE
05	3214 0000 0000 30F0 30F0
0A	323C 0000 345E 30CC 30CC
OF	3264 3452 3452 0000 31C8
11	3274 0000 0000 30BA 30BA
1C	32CC 0000 3458 3102 3102
1E	32DC 0000 0000 3114 3114

Figure 13-2. Example of a Run Queue and Exchanges Displayed by CODE-S.

The Run Queue display lists the processes that are ready to run in priority order. These processes are active instead of WAITing.

Each line in the Exchanges display describes the state of an exchange. The first number on each line is an exchange identifier. The addresses that appear as column headings are all relative to the operating system's data segment. These addresses are explained below:

- `oExchg` The address of the exchange.
- `oMsgHead` If messages are queued at the exchange, `oMsgHead` is the address of the first message in the queue.
- `oMsgTail` If messages are queued at the exchange, `oMsgTail` is the address of the last message in the queue.
- `oPcbHead` If processes are queued at the exchange, `oPcbHead` is the address of the first process in the queue.
- `oPcbTail` If processes are queued at the exchange, then `oPcbTail` is the address of the last process in the queue.

If messages are queued at the exchange, then both `oMsgHead` and `oMsgTail` are non-zero.

If processes are waiting at the exchange, then both `oPcbHead` and `oPcbTail` are non-zero.

Either messages or processes, not both, are queued at an exchange.

To obtain a display of processes or of the pointers to messages waiting on exchange k, type

```
k CODE-S
```

For example, to obtain a display of the processes or of the pointers to messages waiting on exchange 7, type

```
7 CODE-S
```

Typical Debugger responses are shown below. The first is for processes, and the second is for messages.

```
Exchange 07 - Processes |03|
```

```
Exchange 07 - Messages |ADB2:1217|
```

14 OTHER ADVANCED COMMANDS

This section describes two additional advanced debugging commands. You use these commands to deactivate the Debugger (as opposed to exiting from it), and to turn the line printer echo on and off.

CODE-K: DEACTIVATING THE DEBUGGER (KILLING THE DEBUGGER)

Use the CODE-K command to deactivate the Debugger. As used in this manual, the term "deactivate" means "to remove from the group of features available to the user." Thus, the CODE-K command prevents you from using the Debugger at all.

When you press CODE-K, the Debugger displays a message asking whether you really want to deactivate it. If so, press CODE-K again.

Once you press CODE-K the second time, you can reenter the Debugger only by pressing the RESET button located on the back of your workstation.

NOTE

You must deactivate the Debugger, using CODE-K, before you can install a different version of the Debugger.

CODE-L: TURN LINE PRINTER ECHO ON (OFF)

The Debugger lets you obtain a duplicate of its screen output on a line printer. The CODE-L command lets you obtain a printed copy of the dialogue that appears on the Debugger screen. To activate this feature, type

CODE-L

If the line printer is properly connected and is online, the following message appears on the screen:

```
Lpt echo ON
```

If the line printer is not properly connected, or if it is not online, the following message appears on the screen:

```
Lpt echo OFF
```

If you type CODE-L again and again, the echo feature alternates between its ON and OFF states.

You cannot use the Debugger's printer-echo feature if the line printer is also being used by the operating system or by the program that is being debugged.

In the file Debugger, CODE-L takes an optional filename parameter as follows:

```
'filename' CODE-L
```

'Filename' is the name of the byte stream file that contains the screen output duplicate.

APPENDIX A: STATUS MESSAGES

The error messages that the Debugger displays are shown below in **boldface** type. The explanation of each message appears in regular type.

Breakpoint already set

A previous CODE-B command already set a breakpoint at the specified address.

Pattern not found

The specified pattern was not found in the range of addresses given as parameters of the CODE-O command

Expected parameter not found

You must use a string parameter with this command.

Segmented address parameter not found

You must use an address parameter with this command.

Address must not be in an overlay

You cannot modify code in an overlay.

Too many parameters

You must enter the command again with the correct number of parameters.

Not allowed when interrupts are disabled

The command in question is not available after a CODE-I breakpoint has been taken, or when interrupts are disabled.

Not enough parameters

You must enter the command again with the correct number of parameters.

System error while opening a symbol file

A file system error occurred when the Debugger tried to open the symbol file. You should verify that the file name is spelled correctly.

Not a symbol file

The filename parameter in the CODE-F command is not the name of a symbol file. Check the spelling.

No such command

That command does not exist. (Refer to Appendix B, "Command Summary," for a list of all of the Debugger commands.)

Not implemented

The specified command is not implemented in the Debugger.

Non-existent memory

No physical memory exists at the specified address .

Too many breakpoints

The Debugger permits only 16 breakpoints at one time.

Radix must be between 2 and 16

The parameter of the CODE-R command must be in the range from 2 to 16 (decimal), inclusive.

Expected numeric parameter not found

You must use a numeric parameter with this command.

No such breakpoint

No breakpoint has been set at the address given in the CODE-C command.

Cannot proceed

You cannot invoke CODE-P to resume a process that is already running.

No bounds set

You must set the upper and lower bounds before you turn on the histogrammer.

Histogrammer ON

This message simply indicates that the histogram function is active.

Bucket size must be a multiple of 16

When using the histogram facility, you must specify a number of bytes that is a multiple of 16.

No data

You must run the histogramming facility to obtain data before you can use CODE-Q to examine the data.

Lower bound is greater than upper

The histogrammer requires that the lower-bound address be numerically less than the upper-bound address.

Expected parameter(s) not found

The parameters of the specified command are not of the correct type or number.

PatchArea offset too large

The offset in the PatchArea must not exceed 50 bytes.

APPENDIX B: COMMAND SUMMARY

1. The CODE-R command and the equals sign (=) neither open nor close any locations.
2. The CODE-X, UP arrow, and DOWN arrow commands close one location and open another.
3. All other commands close all locations.

The Debugger parameters and commands and their effects are summarized in the following table.

Parameter(s)	Command	Effect
	CODE-B	Displays a list of all breakpoints
<u>addr</u>	CODE-B	Sets a breakpoint at <u>addr</u> (when in simple or multi-process mode)
	CODE-C	Clears all breakpoints
<u>addr</u>	CODE-C	Clears the breakpoint at <u>addr</u>
<u>k</u> , <u>addr</u>	CODE-D	Displays <u>k</u> bytes starting at <u>addr</u> CODE-E
	CODE-F	Turns symbolic output ON and OFF
' <u>filename</u> '	CODE-F	Uses the symbol file
<u>segment</u> , ' <u>filename</u> '	CODE-F	Uses the symbol file with load offset
<u>addr</u>	CODE-G	Starts the current process at <u>addr</u>
	CODE-H	Invokes the histogram facility
<u>param</u>	CODE-H	Specifies the bucket size for the histogram facility
<u>param</u> , <u>param</u>	CODE-H	Specifies the range for the histogram facility

	CODE-I	Displays a list of all breakpoints
<u>addr</u>	CODE-I	Sets a breakpoint at <u>addr</u> (in interrupt mode)
	CODE-K	Exits from and deactivates the Debugger
	CODE-L	Turns printer echo ON and OFF
	CODE-N	Displays the next entry in a linked list
<u>addr</u>	CODE-N	Displays the first entry in a linked list
<u>k</u> , <u>addr</u>	CODE-N	Displays <u>k</u> entries in a linked list
<u>lower addr</u> <u>upper addr</u> <u>byte pattern</u>	CODE-O	Searches for byte pattern starting at <u>lower addr</u>
	CODE-O	Searches for byte pattern starting at new <u>lower addr</u>
<u>k</u>	CODE-P	Proceeds from the current breakpoint
0	CODE-P	Proceeds <u>k</u> times from the current breakpoint
	CODE-P	Proceeds after removing the current breakpoint
	CODE-Q	Displays the range and bucket size for the histogram facility
1	CODE-Q	Displays the report created by the histogram facility
	CODE-R	Sets the output radix to hexadecimal (or to decimal, if the current radix is hexadecimal)

<u>k</u>	CODE-R	Sets output radix to <u>k</u>
	CODE-S	Displays status of all processes and exchanges
<u>k</u>	CODE-S	Displays status of exchange <u>k</u>
	CODE-T	Displays a trace of the stack
<u>k</u>	CODE-T	Displays a trace of <u>k</u> levels of the stack
	CODE-U	Causes a video display of either the user process or the Debugger procedures
	CODE-X	Executes an instruction, and opens the next instruction
	CODE-Z	Displays the contents of the 8087 registers
<u>addr</u>	CODE-←	Opens <u>addr</u> as a byte
<u>k</u> , <u>addr</u>	CODE-←	Opens <u>k</u> successive bytes
<u>addr</u>	CODE-→	Opens <u>addr</u> as a word
<u>k</u> , <u>addr</u>	CODE-→	Opens <u>k</u> successive words
<u>addr</u>	MARK	Opens <u>addr</u> as an instruction
<u>k</u> , <u>addr</u>	MARK	Opens <u>k</u> successive instructions
value, expression, or instruction	=	Re-displays the current value

	↑	Opens the previous location
value	↑	Changes the current location, and opens the previous location
	↓	Opens the next location
value	↓	Changes the current location, and opens the next location
	GO	Exits from the Debugger, and (in simple mode) proceeds from the current breakpoint
	RETURN	Closes the open location

APPENDIX C: ALPHABETICAL LIST OF COMMANDS

All of the commands that are presently implemented in the Debugger are listed alphabetically below. The page number at the end of each entry refers to the section in the manual where the command is described.

- CODE-A: Setting Conditional Breakpoints, 5-2
- CODE-B: Setting and Querying Breakpoints, 5-1
- CODE-C: Clearing Breakpoints, 5-2
- CODE-D: Displaying the Contents of Memory, 4-3
- CODE-E: Breaking after the Current Instruction, 5-6
- CODE-F: Opening a Symbol File, 3-3
- CODE-G: Starting a Process at a Specified Address, 5-5
- CODE-H: Invoking the Histogram Facility, 11-1
- CODE-I: Setting Interrupt-Level Breakpoints, 12-1
- CODE-K: Deactivating the Debugger (Killing the Debugger), 14-1
- CODE-L: Turning the Line-Printer Echo ON (OFF), 14-1
- CODE-N: Displaying Linked-List Data Structures, 13-1
- CODE-O: Searching for a Byte Pattern in Memory, 4-6
- CODE-P: Proceeding from a Breakpoint, 5-4
- CODE-Q: Querying the Histogram Facility, 11-3
- CODE-R: Changing the Base of the Number System, 3-5
- CODE-S: Displaying the Status of Processes and Exchanges, 13-2

CODE-T: Displaying a Trace of the Stack, 7-1
CODE-U: Displaying the User Screen, 7-2
CODE-X: Executing Instructions Individually, 5-5
CODE-Z: Displaying the Contents of the 8087
Registers, 7-2

(CODE-J, CODE-M, CODE-V, CODE-W, and CODE-Y are
not implemented in the present Debugger.)

APPENDIX D: THE DEBUG FILE UTILITY

The Debug File utility lets you examine and modify the data in files and devices. It does not let you edit a run file. However, you do use this utility to patch run files and libraries.

To invoke this utility from the Executive, enter the utility name in the command field and press RETURN. The Debug File command form will then appear.

COMMAND: Debug File

DEBUG FILE COMMAND FORM

File Name _____
[Write?] _____
[Image Mode?] _____

where

File Name is the name of the file or device that you want to examine or modify

[Write?] asks you to enter "Yes" or "No" (default is "No") to indicate whether you want to modify any data. (If you enter "Yes", you will be able to modify the data.)

[Image mode?] asks you to enter "Yes" or "No" (default is "No") to indicate how you want the Debug File utility to interpret the data in the run file. If you enter "Yes", the utility interprets the data exactly as the data appears in the run file. Otherwise, the utility interprets the data the way the data appears when loaded in memory.

WARNING

Be very careful not to damage the CTOS file system by writing to a file or to a device such as a floppy disk drive that you have opened via this utility.

PROMPTS AND COMMANDS

When invoked, this utility prompts you with a percent sign (%).

You can use all the Debugger commands except those pertaining to breaking and proceeding, as described in Section 5 above, to examine and modify the data in a file or device.

The CODE-S (process and exchange status) command and the CODE-T (stack trace) command work properly only if the file being debugged is the crash dump file, [Sys]<Sys>CrashDump.Sys.

EXITING

To exit from this utility, press either FINISH or GO. Any modifications you made to the data in the file or device are properly recorded on the disk only if you press FINISH to exit from the utility.

When you modify a run file, the utility automatically corrects the run file checksum word.

APPENDIX E: OPERATING NOTES

DUAL FLOPPY-DISK SYSTEMS

When a system is booted, the Debugger tries to create a file called <Sys>DebuggerSwap.Sys. This file requires 74 free sectors on the disk.

If you want to use the Debugger on a dual floppy disk system, the system disk must contain the file called <Sys>Debugger.Sys. This file must not be write-protected, because if it is, the Debugger cannot create its swap file.

You must enter a CODE-K command to the Debugger before removing the system disk from the drive.

CLUSTER SYSTEMS

Before using the Debugger on a cluster system (either from the master workstation or from one of the cluster workstations), you should follow carefully the cluster software installation procedure described in the Release Notice.

DEBUGGER FILES

The Debugger has different file names, depending on the type of workstation. The Debugger file must be in the system directory [Sys]<Sys>.

The Debugger file names are listed below:

- o IWS Debugger.Sys
- o AWS Debugger.Sys
- o N-GEN/T1 DebuggerT1.Sys

When the Debugger initializes itself, it creates a file called "DebuggerSwp.Sys" in the <\$> directory.

GLOSSARY

Address expression. Description of a location in memory. The description consists of one or more symbols, or an indexed or nonindexed parameter.

Breakpoint. A user-defined point in the code for a process. Execution stops when a process reaches a breakpoint.

Bucket. A small region of memory, measured in multiples of 16 decimal bytes, used by the Debugger's histogram facility.

Byte pattern. User-defined group of byte specifiers. The specifiers are separated by commas and enclosed in double quotation marks.

Byte specifier. Sequence of two-digit hexadecimal numbers, or a string of characters enclosed in single quotation marks.

Clear. Remove a breakpoint from a particular location in memory.

Code listing. English-language display of code generated by a compiler or translator.

Crash dump. Output (memory dump) caused by a system failure.

Current process. The process identified by the PR register in the Debugger. Any registers that are read or written by the Debugger are for the current process.

Current value. The value most recently typed by the user, or the value most recently displayed by the Debugger.

Echo. Repetition on a line printer or on a screen of instructions entered by the user and/or material displayed by the Debugger.

Exchange. Path on which a process waits for or receives messages or communications from another process or processes.

Histogram. Debugger facility that displays the frequency with which a process refers to each memory location in a user-specified range.

Indexed address. Address expression that uses index registers.

Interrupt mode. Debugger operating mode used to debug interrupt handlers or to set breakpoints in the Operating System Kernel.

Link word. Word address (*i.e.*, a 16-bit address) pointing to the next block of data.

Linked-list data structure. Data structure containing elements that are linked by 16-bit addresses (link words) or by 32-bit addresses (link pointers). The CODE-N command uses link words.

Linker. Software system that loads and connects together the object programs output separately by a compiler or assembler (BASIC, FORTRAN, *et al.*), and from them produces a run file.

Multiprocess mode. Debugger operating mode used in debugging a user task that involves more than one process and that depends on continuous execution of all processes except the ones stopped at breakpoints.

Offset. The number of bytes by which a memory location is distant from the beginning of a segment.

Output radix. The base of the notation in which Debugger output is expressed (binary, decimal, hexadecimal, or any other base from 2 to 16, inclusive).

Parameter. A constant (number, port, or text), a symbol, or one or more unary or binary operators, address expressions or symbolic instructions.

Physical address. An address that does not specify a segment base, and is relative to memory location 0.

Pointer. See "segmented address" below.

Port constant. Number followed by an "i" or an "o" (indicating an input port and an output port, respectively).

Public symbol. An ASCII character string associated with a public variable, a public value, or a public procedure.

Public variable. Variable whose address can be referenced by a module other than the module in which the variable is defined.

Public value. Value whose address can be referenced by a module other than the module in which the value is defined.

Public procedure. Procedure whose address can be referenced by a module other than the module in which the procedure is defined.

Register mnemonic. Two-letter symbolic name for a register in the 8086/80186 processor (for example, AX, BL, SI).

Run file. File created by the linker. The run file contains the initial image of code and data for a program.

Run-file checksum word. Number produced by the summation of words in a run file. Used to check the validity of the run file.

Segment. A discrete portion of memory, of a routine, or of a program.

Segment address. Address of a segment base. For an 8086/80186 microprocessor, a segment address refers to a paragraph (16 bytes).

Segmented address (pointer). Address that specifies both a segment base and an offset.

Segment override. Operating code that causes the 8086/80186 to use the segment register specified by the prefix when executing an instruction, instead of the segment register that it would normally use.

Set. Place a breakpoint at a particular location in memory.

Simple mode. Debugger operating mode used in debugging a single-process user task, for example, a compiled BASIC program.

Stack. A region of memory, accessible from one end by means of a stack pointer.

Stack frame. Region of a stack corresponding to the dynamic invocation of a procedure. Consists of procedural parameters, a return address, a saved-frame pointer, and local variables.

Stack pointer. The indicator to the top of a stack. The stack pointer is stored in the registers SS:SP.

Stack trace. Debugger display of a stack, organized by stack frames.

State variable. Symbolic name of a register that contains data indicating the state of the Debugger (for example, PR, IP, or FL).

Symbol. Sequence of alphanumeric and other characters (under-score, period, dollar sign, pound sign, or exclamation mark).

Symbolic instructions. Instructions containing symbols, that is, mnemonic characters corresponding to assembly-language instructions. (These instructions cannot contain user-defined public symbols.)

System process. Any process that is not terminated when the user calls Exit.

Text constant. Sequence of characters enclosed by quotation marks.

User process. Any process that is terminated when the user calls Exit.