intel

Development Solutions

ICE™-186/188 In-Circuit Emulator User's Guide

Order Number: 452320-003

Notational Conventions

This supplement uses the following notational conventions:

`Computer output`	In the manual examples, computer output is distinguished with monospace type.
[...]	means the preceding item may be repeated. If a delimiter is required between entries, the delimiter appears before the ellipsis. For example, [,...] indicates a comma delimiter.
italics	indicate variable expressions. Substitute a value or a symbol.
item1::=item2	denotes that item1 is replaced by the entries designated as item2.
{items}	between braces indicate you must select one and only one of the enclosed items. Alternatives are stacked or separated by a vertical bar (e.g., item1 | item2).
[items]	between brackets indicate you may optionally select one and only one of the enclosed items. Alternatives are stacked or separated by a vertical bar (eg., item1 | item2).
<Key>	represents a named key. For example, <Enter> denotes the enter key (referred to in the syntax menu as <eol>).
punctuation	when punctuation such as periods, parentheses, braces ({}), or brackets ([]) must be entered as part of the command syntax, it will be noted in the description of the syntax in the command dictionary.
UPPERCASE	Upper-case letters represent command or parameter keywords. You can enter keywords in either upper- or lower-case letters.
`USER INPUT`	In the manual examples, user input is distinguished with monospace type in color.

ICE™-186/188 IN-CIRCUIT EMULATOR USER'S GUIDE

Order Number: 452320-003

REV.	REVISION HISTORY	DATE
-001	Original Issue.	11/87
-002	Revised Issue. Mechanical changes to the user probe cable and enhanced human interface.	07/88
-003	Change Notice. Order number 482108-001. New commands added. Numeric processor extension unit now supported.	01/89

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

CAUTION

This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instruction manual, may cause interference to radio communications. As temporarily permitted by regulation, it has not been tested for compliance with the limits for Class A Computing Devices pursuant to Subpart J of Part 15 of FCC rules, which are designed to provide reasonable protection against such interference. Operation of this equipment in a residential area is likely to cause interference in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

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

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

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

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

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

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

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

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

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

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

Above	iLBX	Intellink	MICROMAINFRAME	Ripplemode
BITBUS	i_m®	iOSP	MULTIBUS®	RMX/80
COMMputer	iMDDX	iPAT	MULTICHANNEL	RUPI
CREDIT	iMMX	iPDS	MULTIMODULE	Seamless
Data Pipeline	Inboard	iPSC	ONCE	SLD
ETOX	Insite	iRMK	OpenNET	SugarCube
FASTPATH	Intel®	iRMX®	OTP	UPI
Genius	int_el®	iSBC®	PC BUBBLE	VLSiCE
i^{Δ}_{1}	Intel376	iSBX	Plug-A-Bubble	376
$i_{®}$	Intel386	iSDM	PROMPT	386
I^2ICE	int_elBOS	iSXM	Promware	386SX
ICE	Intel Certified	KEPROM	QueX	387
iCEL	Intelevision	Library Manager	QUEST	387SX
iCS	$Int_eligent$ Identifier	MAPNET	Quick-Erase	4-SITE
iDBP	$Int_eligent$ Programming	MCS®	Quick-Pulse Programming	
iDIS	Intellec®	Megachassis		

MULTIBUS® is a patented Intel bus.

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

MDS is an ordering code only and is not used as a product name or trademark, MDS® is a registered trademark of Mohawk Data Sciences Corporation.

GPIB-PC2A is a trademark of National Instruments Corporation.

Contents

Chapter 4 Emulation Features

To get started with your ICE™-186/188 In-Circuit Emulator, do the following:

- Set up your host computer as directed in the *ICE™-186 In-Circuit Emulator Installation Supplement.*
- Install the ICE-186/188 emulator hardware and software as directed in the *ICE™-186/188 In-Circuit Emulator Installation Supplement.*
- Run the on-line ICE-186/188 tutorial as described in Chapter 2 of this manual.

This manual has the following structure:

Chapter 1	Introduces you to the ICE-186/188 features and use.
Chapter 2	Provides a sample debug session and a guide to the ICE-186/188 tutorial.
Chapter 3	Provides a detailed description of the emulator command language.
Chapter 4	Provides a detailed description of the ICE-186/188 emulation features.
Chapter 5	Provides useful information for using the ICE-186/188 emulator with external equipment.
Chapter 6	Provides a command dictionary for the ICE-186/188 command language.
Appendix A	Lists limitations associated with the ICE-186 emulator.
Glossary	Defines terms used in this manual.

Index	Provides an alphabetical index to the topics covered in this manual.
Service Information	The inside-back cover of this supplement provides the information you need if service is required for the emulator.

Contents
Chapter 1 Overview

Chapter 1
Overview

1.1 Introduction

The Intel ICE™-186/188 In-Circuit Emulator is a versatile and efficient tool for developing, debugging, and testing products based on the Intel 80C186 or 80C188 microprocessor. It operates in conjunction with an IBM® Personal Computer AT and features hardware and software emulation support.

Key features of the emulator include:

- **Deep Trace Memory** -- A deep trace memory allows you to capture up to 3072 CYCLES-format frames. Each frame contains either an execution or a bus event. You can view large blocks of real-time program execution.
- **Two-Level Breakpoints with Occurrence Counters** -- You can define IF-THEN breakpoint statements to stop emulation at specific execution points. The breakpoints can be defined as execution addresses, bus addresses, or bus access types such as memory and I/O reads or writes.
- **Single-Step Capability** -- Stepping commands allow you to view program execution one step at a time. A step can be defined as an assembly language instruction or a high-level language statement.
- **128K Bytes of Zero Wait-State Mapped Memory** -- During the debugging session, you have access to 128K bytes of mapped emulator memory. You can create a "virtual" user environment before your target system reaches prototype form. You can map memory and I/O resources in 4K-byte increments.
- **RS232C and GPIB Communication Links** -- Two communication links are available for interfacing with the IBM PC AT. You can use either a serial (RS232C) or a parallel (GPIB) link.
- **Numeric Processor Extension Unit Support** -- The emulator supports the 80C187 numeric processor extension unit. You can execute 80C187 instructions and view the associated memory and register contents.

- **Crystal Power Accessory (CPA)** -- A CPA board is available for emulator self-test. You can also use the CPA as a target system for debugging code when prototype hardware is not available.
- **PLCC Package Support** -- The emulator supports prototype hardware that incorporated surface mounted 80C186 or 80C188 microprocessors. The ability to emulate with surface mounted components works well for the testing of production boards or systems.

The rest of this chapter describes the emulator and its hardware and software components.

1.2 Hardware Overview

The following paragraphs briefly describe the hardware components, options, and host computer. Refer to Figure 1-1 for an overview of how these hardware pieces fit together.

NOTE

For instructions on how to install the hardware pieces, refer to the *ICE™-186/188 In-Circuit Emulator Installation Supplement.*

1.2.1 Host Computer

The host computer is user-supplied. You can use an IBM PC AT or a fully equivalent computer. For a complete list of PC requirements, refer to Section 1.4 in this chapter.

1.2.2 Host-to-Emulator Communication Link

You can use either an RS232C or a GPIB communication link between the emulator and the IBM PC AT.

The RS232C communication link supports serial transfers on PC ports COM1 and COM2. It provides standard baud rates up to 57.6K baud.

The GPIB (IEEE-488) communication link supports parallel transfers at rates up to 300K bytes/second. To use this communication link, you must have a National Instruments GPIB-PC2A^{R} board and GPIB.COM driver software (Rev C4) installed in your IBM PC AT. You must also supply an appropriate GPIB cable.

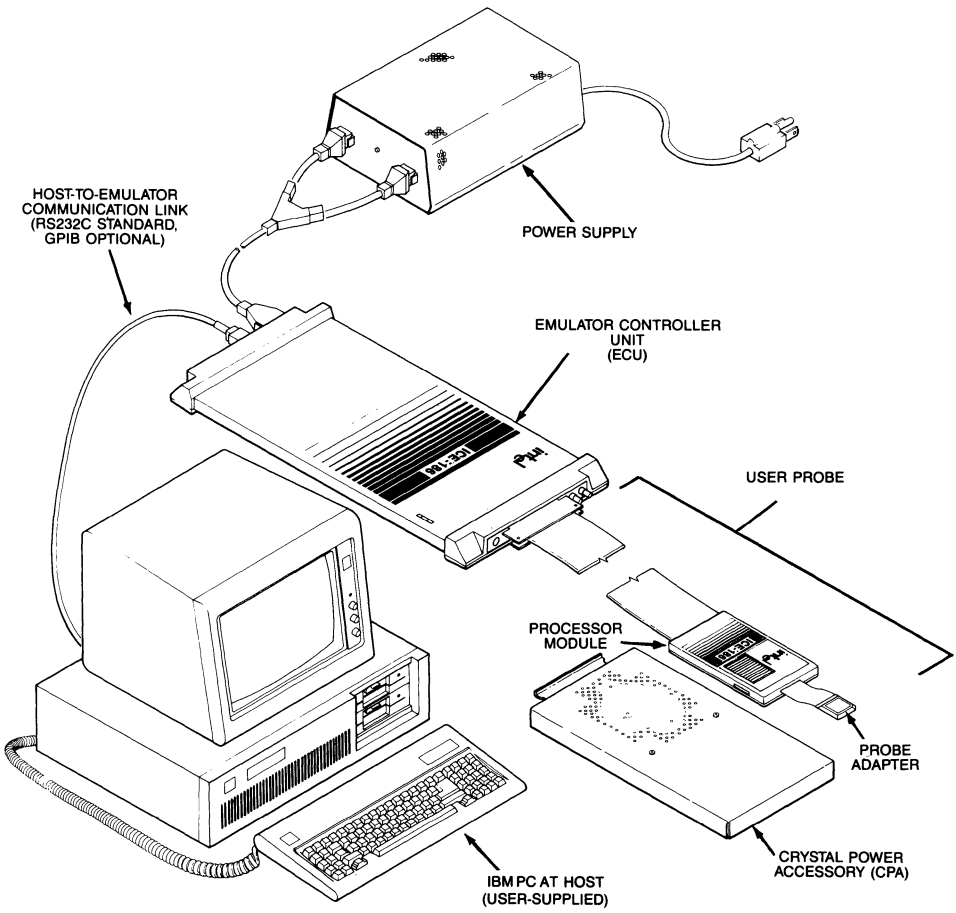


Figure 1-1 ICE™-186/188 Hardware Overview

1.2.3 Emulator Controller Unit

The emulator controller unit (ECU) maintains the communication link between the emulator and the IBM PC AT and any optional external equipment. It supports the following emulator functions:

- Activity Monitor and word recognizers
- Real-time trace
- Memory mapping
- I/O mapping

For a description of the above emulator functions, refer to Chapter 4 in this manual.

1.2.4 User Probe

The user probe provides the physical connection between the emulator controller unit and the user's target system. It consists of a processor module, cable, and probe adapter. The processor module contains a special bondout version of the 80C186 microprocessor or the 80C188 microprocessor, depending on the in-circuit emulator you purchased.

NOTE

For applications that support a surface-mounted PLCC package of the 80C186 contact your local Intel sales office to order a customized socket adapter.

NOTE

The emulator draws power from your target system (1 amp max.). This ensures that cycling the power in the target system will not damage the in-circuit emulator bondout processor. This also prevents the user from having to exit the emulator software to remove or insert the emulator probe into the target system. See section 1.5 for electrical specifications.

1.2.5 Crystal Power Accessory

The crystal power accessory (CPA) provides the signals required by the emulator for stand-alone operations. You use the CPA for running the emulator SELFTEST command and for debugging code when prototype hardware is not available.

1.2.6 Power Supply

The power supply unit provides the voltages needed by the emulator. It meets UL and CSA standards for Information Processing Equipment and Data Processing Equipment, respectively. You can configure the power supply for operation in most countries.

1.2.7 Optional External Equipment

The emulator provides connectors for the following external equipment:

- iPAT™ Analyst -- A 60-pin connector is available for the Intel performance analysis tool.
- Logic Analyzer -- A 60-pin connector is available for a general-purpose logic analyzer.

The above equipment can be used with the emulator, but it is not essential to emulator operation.

1.3 Software Overview

The emulator software installs and runs under PC-DOS (Version 3.1 or later). Once the software is invoked, you can access the emulator command language and emulator functions. You can also access DOS at any time without exiting the emulator software.

NOTE

For instructions on how to install the emulator software, refer to your *ICE™-186/188 In-Circuit Emulator Installation Supplement.*

The following paragraphs briefly describe the human interface and emulation features of the emulator. More detailed information is provided later in this manual.

1.3.1 Human Interface Features

The emulator command language provides many features that assist you in entering commands, handling files, and managing the debug session. These features include:

- Command construction
  - Command line entry and editing
  - Command history
  - Syntax menu
  - On-line help
  - On-line error message query
- Control constructs (block commands)
- I/O redirection commands
- Pipe commands
- Debug object maintenance
- Program memory maintenance using program symbolics or CPU addresses and data types
- Evaluation of expressions
- Access to DOS from the user interface (DOS escape)

For more information about the human interface features, refer to Chapter 3 of this manual.

1.3.2 Emulation Features

The command language also provides powerful emulation features for debugging 80C186-based (or 80C188-based) systems. These features include:

- Invocation commands
- Utility commands
- Memory management commands

The following paragraphs briefly describe the human interface and emulation features of the emulator. More detailed information is provided later in this manual.

1.3.1 Human Interface Features

The emulator command language provides many features that assist you in entering commands, handling files, and managing the debug session. These features include:

- Command construction
  - Command line entry and editing
  - Command history
  - Syntax menu
  - On-line help
  - On-line error message query
- Control constructs (block commands)
- I/O redirection commands
- Pipe commands
- Debug object maintenance
- Program memory maintenance using program symbolics or CPU addresses and data types
- Evaluation of expressions
- Access to DOS from the user interface (DOS escape)

For more information about the human interface features, refer to Chapter 3 of this manual.

1.3.2 Emulation Features

The command language also provides powerful emulation features for debugging 80C186-based (or 80C188-based) systems. These features include:

- Invocation commands
- Utility commands
- Memory management commands

- Program support commands
- Emulation support commands
- I/O control commands
- Processor control commands

For more information about the emulation features, refer to Chapter 4 of this manual.

1.3.3 Language Support

The emulator command language supports emulation for programs assembled with ASM86 or compiled with any of the following:

- C-86
- PL/M-86
- Pascal-86
- Fortran-86

To utilize the full symbolics capabilities of the emulator, modules must be compiled without optimization and with the compiler switches for debug information and symbols on.

1.3.4 Unsupported Features

This release of the emulator software does not include support for the following features:

- The ESC mnemonic in single line assembly. (However, the ESC mnemonic is disassembled and followed by the appropriate operands in the trace display.)
- Halting emulation, including FASTBREAK actions, while the 80C186 or 80C188 component is in a state of "power save," is not supported.
- "ONCE" Test Mode. If your application takes advantage of this feature and you try to emulate the 80C186 (or 80C188), the output pins of the emulator processor will not be placed into the high-impedance state.

When support for any of these features becomes available, you will receive a software update. To be eligible for this software update, you must return the Software Registration card included in your emulator software package.

1.4 PC Requirements

You can use an IBM PC AT or a fully equivalent computer as the host computer for the emulator. The PC must meet the following minimum requirements:

- 640K bytes of RAM in conventional memory
- Intel Above™Board with at least 1M bytes of expansion memory
- One 360K bytes or one 1.2M-byte diskette drive
- One 20M bytes fixed-disk drive with at least 1.4M bytes of free disk space
- PC-DOS 3.1 or later
- A serial port (COM1 or COM2) supporting a 9600 baud minimum data transfer rate, or a National Instruments GPIB-PC2A board and GPIB.COM driver software (Rev C4).
- An 80287 numeric coprocessor in the host computer. (Required only if the target has an 80C187 numeric processor extension unit and you plan to display the 80C187 stack registers.)

1.5 ICE™-186 Electrical Specifications

The following paragraphs describe the electrical specifications for the ICE-186 in-circuit emulator. Figure 1-2 illustrates the dimensions of the user probe adapter and the clearance required in your target system.

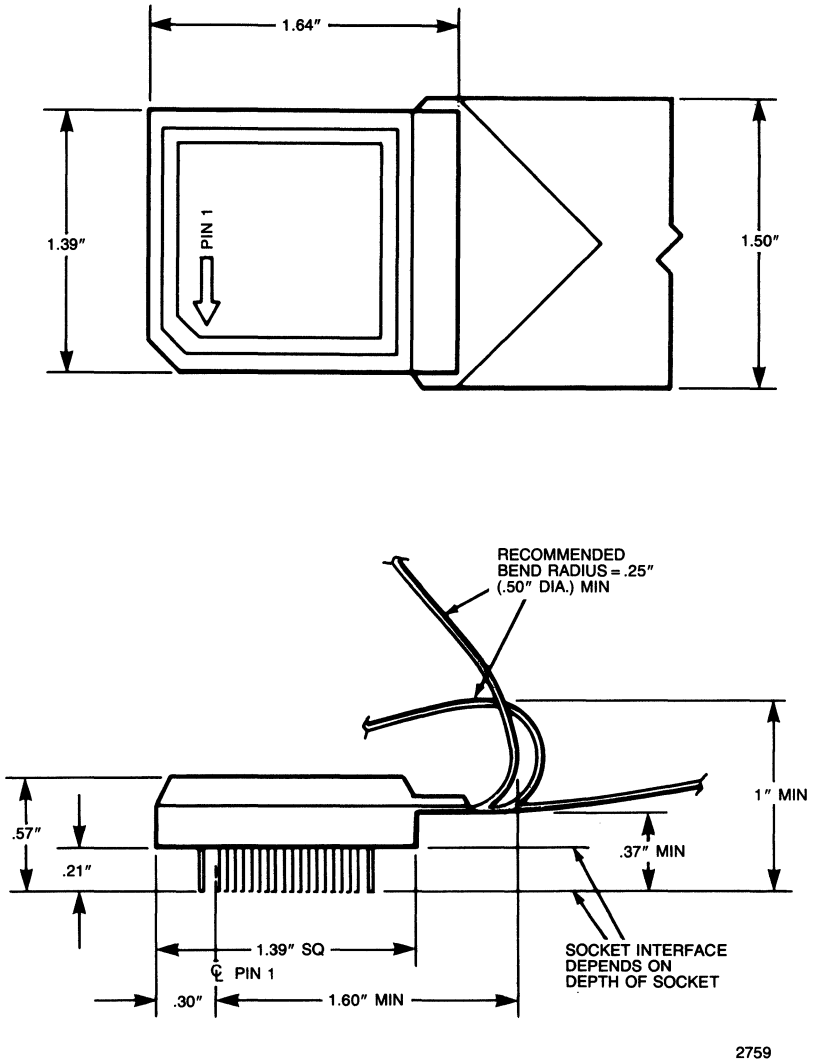


Figure 1-2 ICE^{TM}-186 User Probe Dimensions

Table 1-1 AC Specifications

SYMBOL	PARAMETER	80C186 CPU		ICE-186	
		MIN	MAX	MIN	MAX
TIMING REQUIREMENTS					
TDVCL	Data in Setup (A/D)	15		24	
TCLDX	Data in Hold (A/D)	5		5	
TARYCH	Asynchronous Ready (ARDY) Resolution Transition Setup Time	15		23	
TARYLCL	ARDY Inactive Setup Time	25		25	
TCLARX	ARDY Active Hold Time	15		15	
TARYCHL	ARDY Inactive Hold Time	15		15	
TSRYCL	Synchronous Ready (SRDY) Transition Setup Time	15		25	
TCLSRY	SRDY Transition Hold Time	15		15	
THVCL	HOLD Setup	15		32	
TINVCH	NMI /TEST INTR, TIMERIN Setup Time	15 15 15		32 31 17	
TINVCL	DRQ0, DRQ1, Setup Time	15		19	

Table 1-1 AC Specifications (continued)

SYMBOL	PARAMETER	80C186 CPU		ICE-186	
		MIN	MAX	MIN	MAX
MASTER INTERFACE TIMING RESPONSES					
TCLAV	Address Valid Delay	5	36	5	33
TCLAX	Address Hold	0		0	
TCLAZ	Address Float Delay				
	READ Cycles	TCLAX	25	5	36
	INTA Cycles	TCLAX	25	0	25
	HLDA	TCLAX	25	10	50
TCHCZ	Command Lines Float Delay		33		33
TCHCV	Command Lines Valid Delay (after Float)		37		37
TLHLL	ALE Width (min)	TCLCL-30		TCLCL-32	
TCHLH*	ALE Active Delay		25		42
None**	CLKOUT Low to ALE Active				19
TCHLL	ALE Inactive Delay		25		40
TLLAX	Address Hold to ALE Inactive (min)	TCHCL-15		TCHCL-28	
TCLDV	Data Valid Delay	5	36	3	36
TCLDOX	Data Hold Time	5		7	
TWHDX	Data Hold after /WR (min)	TCLCL-20		TCLCL-20	

*Applies only when the ALEMODE variable is set to START.
**Applies only when the ALEMODE variable is set to END.

Table 1-1 AC Specifications (continued)

SYMBOL	PARAMETER	80C186 CPU		ICE-186	
		MIN	MAX	MIN	MAX
MASTER INTERFACE TIMING RESPONSES					
TCVCTV	Control Active Delay 1				
	/DEN	5	47	3	48
	/INTA	5	47	1	47
	/WR	5	47	2	47
TCHCTV	Control Active Delay 2	5	37	3	37
TCVCTX	Control Inactive Delay				
	/DEN	5	37	3	40
	/INTA	5	37	1	37
	/WR	5	37	2	37
TCVDEX	/DEN Inactive Delay (Non-Write Cycle)	5	47	5	47
TAZRL	Address Float to /RD Active	0		-30	
TCLRL	/RD Active Delay	5	37	2	37
TCLRH	/RD Inactive Delay	5	37	2	37
TRHAV	/RD Inactive to Address Active (min)	TCLCL-20		TCLCL-20	
TCLHAV	HLDA Valid Delay	5	33	2	33
TRLRH	/RD Pulse Width (min)	2TCLCL-40		2TCLCL-40	
TWLWH	/WR Pulse Width (min)	2TCLCL-30		2TCLCL-30	
TAVLL	Address Valid to ALE Low (min)	TCLCH-15		TCLCH-19	

Table 1-1 AC Specifications (continued)

SYMBOL	PARAMETER	80C186 CPU		ICE-186	
		MIN	MAX	MIN	MAX
MASTER INTERFACE TIMING RESPONSES					
TCHSV	Status Active Delay	5	35	5	35
TCLSH	Status Inactive Delay	5	35	5	35
TCLTMV	Timer Output Delay		40		40
TCLRO	Reset Delay		40		40
TCHQSV	Queue Status Delay		28		35
TCHDX	Status Hold Time	5		5	
TAVCH	Address Valid to Clock High	0		0	
TCLLV	/LOCK Valid-Invalid Delay	5	40	1	40
TDXDL	/DEN Inactive to DT|/R Low	0		-7	
CHIP-SELECT TIMING RESPONSES					
TCLCSV	Chip-Select Active Delay		33		33
TCXCSX	Chip-Select Hold from Command Inactive	TCLCH-10		29	
TCHCSX	Chip-Select Inactive Delay	5	28	1	28

Table 1-1 AC Specifications (continued)

SYMBOL	PARAMETER	80C186 CPU		ICE-186	
		MIN	MAX	MIN	MAX
CLKIN REQUIREMENTS					
TCKIN	CLKIN Period	40	1000	40	1000
TCKHL	CLKIN Fall Time		5		5
TCKLH	CLKIN Rise Time		5		5
TCLCK	CLKIN Low Time	15		15	
TCHCK	CLKIN High Time	15		15	
CLKOUT TIMING					
TCICO	CLKIN to CLKOUT Skew		21		37
TCLCL	CLKOUT Period	80	2000	80	2000
TCLCH	CLKOUT Low Time (min)	.5TCLCL-5		.5TCLCL-5	
TCHCL	CLKOUT High Time (min)	.5TCLCL-5		.5TCLCL-5	
TCH1CH2	CLKOUT Rise Time		10		10
TCH2CL1	CLKOUT Fall Time		10		10

TABLE 1-2 ICE™-186 DC Input Specifications

SIGNAL	V_{IL} MAX	V_{IH} MIN	I_{IL} MAX	I_{IH} MIN
X1	1.65 V	3.85 V	-.001 mA	0.001 mA
/RES	0.7 V	2.0 V	-0.6 mA	0.02 mA
/TEST	0.8 V	2.0 V	-1.5 mA	0.02 mA
TMR IN 0, 1	.2 Vcc-.3V	.2 Vcc+.9 V	-0.01 mA	0.01 mA
DRQ0, 1	.2 Vcc-.3 V	.2 Vcc+.9V	-0.01 mA	0.01 mA
NMI	0.8 V	2.0 V	-0.25 mA	0.025 mA
INT0, INT1	.2 Vcc-.3V	.2 Vcc+.9 V	-0.35 mA	0.045 mA
INT2|/INTA0	.2 Vcc-.3 V	.2 Vcc+.9 V	-0.35 mA	0.045 mA
INT3|/INTA1	.2 Vcc-.3 V	.2 Vcc+.9 V	-0.35 mA	0.045 mA
AD0-AD15	0.8 V	2.0 V	-0.07 mA	0.07 mA
ARDY	0.8 V	2.0 V	-0.5 mA	0.02 mA
SRDY	0.8 V	2.0 V	-0.5 mA	0.02 mA
HOLD	0.8 V	2.0 V	-0.6 mA	0.02 mA

Table 1-3 summarizes the differences in DC loading imposed by the ICE-186 emulator.

TABLE 1-3 ICE™-186/188 DC Output Specifications

SIGNAL	V_{OL} MAX	V_{OH} MIN	I_{OL} MAX	I_{OH} MAX
RESET	0.5 V	2.5 V	20 mA	-1 mA
CLKOUT	0.5 V	4.0 V	5 mA	-1 mA
TMR OUT 0, 1	0.45 V	2.4 V	2.5 mA	-2.4 mA
INT2|/INTA0	0.45 V	2.4 V	2.5 mA	-2.4 mA
INT3|/INTA1	0.45 V	2.4 V	2.5 mA	-2.4 mA
A16|S3-A19|S6	0.5 V	2.4 V	20 mA	-1 mA
AD0-AD15	0.5 V	2.4 V	48 mA	-3 mA
/BHE|S7	0.4 V	2.4 V	2.5 mA	-2.4 mA
ALE|QS0	0.5 V	2.4 V	24 mA	-3.2 mA
/WR|QS1	0.5 V	2.4 V	24 mA	-3.2 mA
/RD|/QSMD	0.5 V	2.4 V	24 mA	-3.2 mA
/LOCK	0.45 V	2.4 V	2.5 mA	-2.4 mA
/S0, /S1, /S2	0.5 V	2.4 V	20 mA	-3 mA

TABLE 1-3 ICE™-186/188 DC Output Specifications (continued)

SIGNAL	V_{OL} MAX	V_{OH} MIN	I_{OL} MAX	I_{OH} MAX
HLDA	0.5 V	2.5 V	20 mA	-1 mA
xCS	0.45 V	2.4 V	2.5 mA	-2.4 mA
DT|/R	0.45 V	2.4 V	2.0 mA	-2.4 mA
/DEN	0.5 V	2.4 V	48 mA	-3 mA

Table 1-4 lists the DC specifications for the ISYNCH and OSYNCH lines.

TABLE 1-4 DC Specifications for ISYNCH and OSYNCH

SYMBOL	PARAMETER	MIN	MAX	NOTES
ISYNCH I_{il}	Input Low current		2.6 mA	V_i = 0.5 V
ISYNCH I_{ih}	Input High current		-100 uA	V_i = 2.7 V
OSYNC V_{ol}	Output Low voltage		0.5 V	I_{ol} = 15 mA
OSYNCH V_{oh}	Output High voltage	2.7 V	I_{oh} = -0.25 mA	

1.6 Related Publications

This manual contains the information required to use the ICE-186/188 emulator. The following related publications provide installation information and reference information on topics related to the use of the emulator. Related publications and additional copies of this manual may be ordered from the Intel Literature Department, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051.

- *ICE™-186/188 In-Circuit Emulator Installation Supplement*, order number 452321. This supplement provides installation, software invocation, and confidence testing procedures for the ICE-186/188 In-Circuit Emulator.
- *Development Tools Handbook*, order number 210940. This handbook contains data sheets, application notes, and article reprints for all Intel hardware development tools and software development tools.

- *Microprocessor and Peripheral Handbook*, order number 230843. This handbook contains data sheets, application notes, and article reprints for all Intel microprocessors and related peripheral components.
- *86/88/186/188 User's Manual--Programmer's Reference*, order number 210911.
- *86/88/186/188 User's Manual Hardware Reference*, order number 210912.
- *Software Handbook*, order number 230786. This handbook contains data sheets, application notes, and article reprints for all software offered by Intel.
- *Embedded Controller Handbook*, order number 210918. This book contains data sheets, application notes, and article reprints for various embedded controllers, and for the 80186 component family.
- *iPAT™ Analyst User's Guide*, order number 450583. This manual provides complete operating information for the Intel performance analysis tool.
- *80C186 CHMOS High Integration 16-Bit Microprocessor* data sheet, order number 270354. This data sheet contains the specifications for the 80C186 microprocessor.
- *80C188 CHMOS High Integration 16-Bit Microprocessor* data sheet, order number 270432. This data sheet contains the specifications for the 80C188 microprocessor.
- *C: A Reference Manual*, Second Edition, Harbison and Steele

Contents
Chapter 2 Using the Emulator

Chapter 2
Using the Emulator

2.1 Introduction

This chapter contains a sample debugging session, along with invocation procedures for the tutorial.

The sample debugging session is intended as a quick reference guide for experienced users. Refer to it when you need to refresh your memory about how to start, run, or stop the emulator.

The tutorial is a hands-on learning guide for first-time users. All first-time users should use the tutorial to become familiar with the emulator and its operation before attempting actual applications.

2.2 Sample Debugging Session

This section outlines the steps involved in starting, running, and stopping the emulator. It assumes that you have already installed the emulator hardware and software, and have connected the emulator user probe to your target system. If not, refer to the *ICE™-186/188 In-Circuit Emulator Installation Supplement* for detailed installation information.

1. **Turn on the power to the IBM PC AT.**
2. **Prepare your application program.** Create, compile, and link your program. The resulting load files must be in Intel 8086 absolute Object Module Format (OMF).

   To utilize the full symbolics capabilities of the emulator, the modules must be compiled without optimization and with the compiler switches for debug information and symbols on.

3. **Modify the emulator configuration files to support the appropriate I/O device and communication link.** By default, the configuration files specify a serial communication link at 38400 baud, via PC COM1. If you are using a different communication link or baud rate, modify the configuration files accordingly. Procedures for modifying the configuration files are provided in your *ICE™-186/188 In-Circuit Emulator Installation Supplement.*

4. **Turn on the power to the emulator and the prototype.** The power switch for the emulator is located on the back panel of the emulator power supply.

5. **Invoke the emulator software and activate the emulator hardware.** You can invoke the software and activate the emulator hardware by using the RUN186.BAT batch file. To use this file, enter "RUN186" at the DOS prompt:

   ```
   C> RUN186
   ```

   The RUN186.BAT file automatically invokes the software and activates the hardware, using the I/O device information provided in the configuration files. During execution, the file prints the status of the invocation process to the screen. When the process is complete, the command-line prompt changes to "`hlt>`". This prompt indicates that the emulator is active and in Halt mode.

   Instead of using the batch file, you can invoke the emulator software and activate the emulator hardware explicitly. This allows you to override the values of the configuration files. Procedures for explicit invocation are provided under the Invocation entry in Chapter 6 of this manual.

6. **Set the number base for data entry and display.** If you invoked the emulator software by using the RUN186.BAT file, the default base is hexadecimal. If you invoked the emulator software explicitly, the default base is decimal.

   You can change the default base by using the BASE command. Number bases include hexadecimal (16T), decimal (10T), octal (8T), or binary (2T).

For example, enter the following command to set the base to hexadecimal:

```
hlt> BASE = 16T
hlt>
```

7. **Create a record of your debug session.** You can keep a record of your debug session by using the LIST command. This command opens a list file on a disk. All the emulator commands entered and all the data displayed on the screen during the session are saved in this file.

   For example, to open a list file on disk named "sample1", enter the following command (where *path* is the path to the filename):

```
hlt> LIST path\sample1
hlt>
```

   Command parameters are also available for APPENDing or OVERWRITEing an existing log file. (See the LIST entry in Chapter 6 for more information.)

8. **Map program memory.** Use the MAP command to provide space for your application program. With this command, you can map program memory to the prototype (USER) or the emulator (ICE), and specify read/write or read-only access.

   By default, all memory is mapped to USER with read/write access. You can map memory to ICE in 4K-byte blocks, up to 128K bytes maximum. The starting addresses of the mapped 4K-byte blocks fall on 4K-byte boundaries (e.g., 0000HP, 1000HP, 2000HP, etc.).

   For example, to map 128K bytes of memory to ICE, starting at physical address 0HP, enter the following command:

```
hlt> MAP 0HP LENGTH 128K ICE
00000HP  length    20000H   ICE    READ/WRITE
20000HP  length    e0000H   USER   READ/WRITE
hlt>
```

Unless otherwise specified, all memory defaults to read/write access. If you want to specify read-only memory access, use the READ parameter with the MAP command. Note that if you set ICE-mapped memory to read-only access, you must program any chip-select registers associated with the address range to recognize external READY. (See the MAP entry in Chapter 6 for more information.)

You can view the current memory map by entering the MAP command without parameters. You can also reset the memory map to the default values by entering the RESET MAP command.

9. **Map I/O resources.** Use the MAPIO command to map your I/O port addresses to the emulator (ICE) when prototype hardware is not available.

   By default, all I/O port addresses are mapped to the prototype (USER). You can map I/O port addresses to ICE in 4K-byte blocks, up to 64K bytes maximum. The starting addresses of the mapped 4K-byte blocks fall on 4K-byte boundaries (e.g., 0000H, 1000H, 2000H, etc.).

   For example, to map the first 4096 I/O ports to ICE, enter the following command:

```
hlt> MAPIO 0H LENGTH 4K ICE

0000H   length   1000H   ICE
1000H   length   f000H   USER
hlt>
```

   Note that if you map I/O addresses to ICE, you must program any chip-select registers associated with the address range to recognize external READY. (See the MAPIO entry in Chapter 6 for more information.)

   You can view the current I/O map by entering the MAPIO command without parameters. You can also reset the I/O map to the default values by entering the RESET MAPIO command.

10. **Load your program.** Use the LOAD command to load your program into mapped memory.

    For example, to load a program file named "messg" located at directory path "c:\ice186\tutorial", enter the following command:

```
hlt> LOAD c:\ice186\tutorial\messg
```

The emulator loads the program file into mapped memory, creates a symbol table file named "messg.sym", and attaches the symbol table file to the database. (The database manager maintains centralized storage of all the objects in the debug environment.) If the program file contains program-start information, the emulator displays this information when the file is loaded.

11. **Execute the program.** Use the GO command to start, stop, and control the execution of your program. For example, to trace the execution of your program from address 1100:4H until the occurrence of symbol ":messg.rotate_message_", enter the following:

```
hlt> GO FROM 1100:4H TRACE TIL EXECUTION \
hlt>> :messg.rotate_message_ HALT

  Use S (status) command to determine emulation status.
emu+> S
  Emulator Status: halted (not servicing interrupts).
    Execpointer is :MESSG.ROTATE_MESSAGE_#184 + 1H (1000:0067H)
  Activity Monitor Status: Stopped.
hlt>
```

Notice that the command-line prompt changes to "emu+>" while the emulator is running. The S (STATUS) command displays the current emulator status, which is "halted". The "hlt>" prompt then reappears.

12. **Halt emulation.** You can halt emulation at any time by using the HALT command. For example, suppose you enter a GO FOREVER command and want to explicitly halt emulation:

```
hlt> GO FROM 1100:4H FOREVER

  Use S (status) command to determine emulation status.
emu+> HALT
  Emulator Status: halted (not servicing interrupts).
   Execpointer is :MESSG.DELAY_A_LITTLE_BIT_#163 (1000:004eH)
  Activity Monitor Status: Stopped.
hlt>
```

Notice that the command-line prompt changes to "hlt>" immediately after the HALT command is entered.

13. **Print the trace buffer.** Use the PRINT command to format and display the contents of the trace buffer. The following example illustrates the default format (INSTRUCTIONS) of the print display:

```
hlt> PRINT INSTRUCTIONS NEWEST 5
FRAME   ADDRESS       DATA      MNEMONICS
 1983   1000:005cH    0bc2      OR    AX,DX
 1985   1000:005eH    75ee      JNE   $-10H ;A=0001004eH
:MESSG.DELAY_A_LITTLE_BIT_#163
 1989   1000:004eH    8b46fc    MOV   AX,WORD PTR[BP]-04H
 1992   1617eHP-R-b619
 1993   1000:0051H    8b56fe    MOV   DX,WORD PTR [BP]-02H
 1995   16180HP-R-0100
 1996   1000:0054H    836efc01  SUB   WORD PTR [BP]-04H,01H
hlt>
```

14. **Close the LIST file.** If a LIST file was created in Step 7, close the file by entering:

```
hlt> NOLIST
hlt>
```

15. **Deactivate the emulator.** You can deactivate the emulator hardware without exiting the emulator software by using the DEACTIVATE ICE186 command. For example:

```
hlt> DEACTIVATE ICE186
Symbol file(s) removed from database.
->
```

Notice that the command-line prompt changes to "->" after you enter the DEACTIVATE ICE186 command. This prompt indicates that the emulator hardware has been deactivated, but the emulator software is still resident. You can re-activate the emulator hardware by entering the ACTIVATE ICE186 command.

16. **Exit the emulator software.** Once the analysis of the program is complete, exit from the software to DOS by using the EXIT command.

```
-> EXIT
C>
```

17. **Turn off the emulator and the prototype system.**

For additional information about the commands and topics discussed in the preceding steps, refer to the appropriate entries in Chapter 6 of this manual. For a hands-on demonstration of the emulator features, proceed to the tutorial.

2.3 Tutorial Guide

This section shows you how to invoke and use the tutorial. It describes the tutorial organization and provides listings of the tutorial screens and sample program.

This section is divided as follows:

- **Emulator Setup** (Section 2.3.1) -- describes the hardware and software setup required for running the tutorial.
- **Tutorial Invocation** (Section 2.3.2) -- explains how to invoke the tutorial software.
- **Tutorial Organization** (Section 2.3.3) -- describes the tutorial paths and shows you how to use them.
- **Tutorial Screens** (Section 2.3.4) -- provides a listing of all the tutorial screens.
- **Sample Program** (Section 2.3.5) -- contains a list file of the MESSG program used in the tutorial. The list file includes a C-86 listing and an equivalent ASM86 listing.

2.3.1 Emulator Setup

The tutorial assumes that you have already installed the emulator hardware and software, and have connected the user probe to the crystal power accessory (CPA) as shown in Figure 2-1. The CPA allows the emulator to operate in stand-alone mode, where the CPA serves as the target system.

Note that the setting of jumper J1 on the CPA circuit board does not affect the operation of the tutorial. The tutorial operates the same, regardless of whether the jumper is set to COMP (Compatible) or ENH (Enhanced) mode operation.

If you have not installed the hardware and software as described, do so before proceeding with this section. Installation procedures are provided in the *ICE™-186/188 In-Circuit Emulator Installation Supplement.*

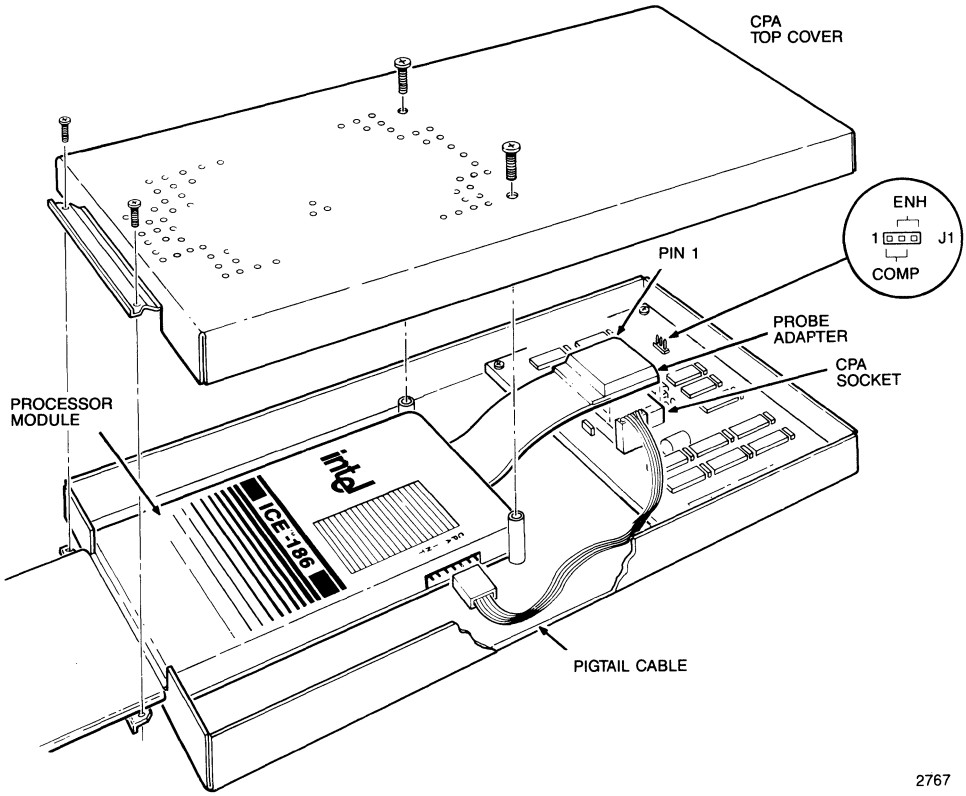


Figure 2-1 ICE™-186/188 Emulator in Stand-Alone Mode

2.3.2 Tutorial Invocation

To invoke the tutorial, perform the following steps:

1. Turn on the IBM PC AT and the emulator power supply.
2. After the DOS prompt appears, change to the tutorial directory:

```
C> CD\ICE186\TUTORIAL
```

3. Run the tutorial batch file:

```
C> TUTORIAL
```

The batch file automatically activates the emulator and invokes the tutorial software. When the invocation process is complete, the first tutorial screen appears on the display, as shown in Figure 2-2.

```
**********************************************************************
*                                                                    *
*     WELCOME TO THE ICE-186/188 IN-CIRCUIT EMULATOR TUTORIAL        *
*                                                                    *
**********************************************************************

                          PLEASE NOTE:

This tutorial assumes the following:

   1.   The tutorial software is in the current directory.
   2.   An ICE-186/188 emulator is connected to the host computer.
   3.   The ICE-186 or ICE-188 emulator user probe is connected to
        the crystal power accessory (CPA).

Refer to Chapter 2 of the ICE-186/188 In-Circuit Emulator User's Guide
for more information about the tutorial.

          Press the <Enter> key to continue, <Ctrl><Break> to abort
```

Figure 2-2 Tutorial Invocation Display

2.3.3 Tutorial Organization

Once the tutorial is invoked, use the on-screen instructions to operate the tutorial. The following paragraphs describe the tutorial organization and operation in general terms.

The tutorial is a sequence of text screens divided into three paths:

- **Human Interface** (HI) -- This path introduces the human interface features of the emulator. The topics include an introduction to debug objects (variables, aliases, etc.), using built-in functions, editing command lines, and creating debug procedures.
- **Load and GO** (LG) -- This path demonstrates the commands used to load a program into emulator memory and execute it. It also introduces using debug procedures and executing a program in fastbreak emulation mode.
- **Debug Session** (DB) -- This path demonstrates using symbolic information to debug a program, including single-stepping through code, disassembling code, and displaying trace information.

You can execute the tutorial paths in any order; however, we recommend that you start with the HI path. The LG and DB paths assume that you have already covered the topics discussed in the HI path.

The screen commands within a given tutorial path should be executed sequentially. The commands on one screen in the path are frequently prerequisites for the commands on the next screen. Executing commands out of sequence may cause unexpected results.

After you have completed a given tutorial path, you can go back to individual screens and experiment with the commands. Each tutorial screen has a name (e.g., HI5, LG13, DB17), as listed in Section 2.3.4. You can display any screen by entering its name at the "hlt>" prompt.

The Main Menu screen is shown in Figure 2-3. The Main Menu screen gives an overview of the tutorial organization. To display this menu on your screen, enter MM (or M) followed by <Enter>.

When you are ready for the next screen, call it by entering N followed by <Enter>. You can move back to the previous screen by entering PR followed by <Enter>. If a screen scrolls out of view before you are finished, redisplay that screen by entering R followed by <Enter>.

```
                                                    --------------------
This tutorial shows you how to use the emulator     |MM: MAIN MENU      |
command language and takes you through a debugging  | M=Go to main menu |
session.                                            | N=Next screen     |
                                                    |PR=Previous screen |
The tutorial is organized into paths.  Paths are    | Q=Quit tutorial   |
sequences of screens with information and examples  | R=Rewrite MM      |
on a group of related topics.                       |xx#=Screen desired |
                                                    --------------------

To start, enter a path name followed by <eol>.  Follow the sequence of
screens within a path because executing the commands on the screens out of
sequence could produce unexpected results!

     HI     Human Interface: An introduction to operating the emulator.

     LG     Load and Go: A sample program load and execute sequence.

     DB     Debug Session: A debug session on the sample program.

                 Enter N <eol> to continue with the tutorial

hlt>
```

Figure 2-3 Tutorial Main Menu (MM)

2.3.4 Tutorial Screens

Table 2-1 lists all of the tutorial screens. Each path is a major division of the tutorial. Begin a path by entering the name of the path (e.g., HI, LG, or DB).

Table 2-1 Tutorial Screens

IS0	Welcome	Introductory Screen
IS1	Welcome (cont)	Introductory Screen (cont)
MM	Main Menu	Tutorial Main Menu

HUMAN INTERFACE (HI) PATH

This path discusses topics that help customize debugging sessions.

HI0	HI Path Menu	HI15	NAMEPATH
HI1	HI Introduction	HI16	Change Paths
HI2	Syntax Menu	HI17	Restore Paths
HI3	HELP Screens	HI18	Functions
HI4	HELP Screens	HI19	Math Functions
HI5	Edit Keys	HI20	PROC
HI6	Debug Objects	HI21	EDITOR
HI7	DIR PROC	HI22	Execute PROC
HI8	Scroll Control	HI23	Edit PROC
HI9	SHOW	HI24	Control Blocks
HI10	INCLUDE	HI25	Control Blocks
HI11	PUT and REMOVE	HI26	Create PROC
HI12	History Buffer	HI27	Create PROC
HI13	DEFINE	HI28	Screen Control
HI14	Search Path	HI29	End HI Path

Table 2-1 Tutorial Screens (continued)

LOAD AND GO (LG) PATH

This path demonstrates the commands used to load and execute a program.

LG0	LG Path Menu	LG12	DOMESSG PROC
LG1	LG Introduction	LG13	DOMESSG PROC
LG2	LIST/NOLIST	LG14	Fastbreak Mode
LG3	Mapping	LG15	Load MESSG2
LG4	MAP	LG16	GO FOREVER
LG5	Load a Program	LG17	REGS
LG6	DIR	LG18	PCB
LG7	Symbol Access	LG19	HALT
LG8	GO	LG20	STATUS/CAUSE
LG9	QUERY	LG21	Fastbreak Mode
LG10	NSTRING memory	LG22	GO TIL FETCH
LG11	DOMESSG PROC	LG23	End LG Path

DEBUG SESSION (DB) PATH

This path demonstrates using symbolic information to debug a program, including single- stepping through code, disassembling code, and displaying trace information.

DB0	DB Path Menu	DB13	Download Code
DB1	DB Introduction	DB14	CALLSTACK
DB2	DOMESSG PROC	DB15	Stepping
DB3	Verify TEMP_	DB16	ISTEP
DB4	Verify TEMP_	DB17	STEP
DB5	SHOWVAR PROC	DB18	LSTEP/PSTEP
DB6	Execute SHOWVAR	DB19	Trace Buffer
DB7	Disassembly	DB20	PRINT
DB8	Assemble Code	DB21	Trace Display
DB9	Verify Patch	DB22	PRINT CYCLES
DB10	Verify Program	DB23	PRINT CYCLES
DB11	SAVE Code	DB24	End DB Path
DB12	Download Code		

2.3.5 Sample Program

The sample program, MESSG, is used in the Load and Go (LG) and Debug Session (DB) tutorial paths. The MESSG program is written in C-86. The program displays and rotates the following message:

```
ICE Tutorial Program
```

The program rotates the message by shifting the leftmost character to the rightmost location. A program delay loop triggers the rotation. The program loops endlessly until it is aborted.

The program contains a bug on line #208 that is found and corrected in the Debug Session (DB) path of the tutorial.

There are four files on disk associated with the tutorial program:

MESSG	compiled, linked, and located absolute code (with bug)
MESSGC.LST	MESSG program listing in C-86
MESSGA.LST	MESSG program listing in ASM86
MESSG2	compiled, linked, and located absolute code (no bugs)

The tutorial paths reference the C-86 program listing.

2.3.5.1 C-86 Program Listing

```
1:  /*****************************************************************************
2:   * PROJECT:          ICE-186, IBM PC AT hosted In-Circuit Emulator          *
3:   *                                                                          *
4:   * ENVIRONMENT:      ICE-186 Emulator connected to Crystal Power Accessory  *
5:   *                                                                          *
6:   * FILENAME:         messg.c  (with bug)                                    *
7:   *                                                                          *
```

```
 8:    * SYNOPSIS:          this file contains the following functions          *
 9:    * int                                                                   *
10:    * main()             - entry point                                      *
11:    *                                                                       *
12:    * void                                                                  *
13:    * delay_a_little_bit()       - delays for < 1/3 second at 12.5 MHz      *
14:    *                                                                       *
15:    * void                                                                  *
16:    * rotate_message()           - rotates message                          *
17:    *                                                                       *
18:    * void                                                                  *
19:    * initialize_buffer()        - sets up work buffer                      *
20:    *                                                                       *
21:    * DESCRIPTION:    This file contains a program designed for use         *
22:    *   with the ICE-186/188 tutorial.  The program is written in the       *
23:    *   C programming  language.  It takes the string "ICE Tutorial Program"*
24:    *   and performs a "marquee" rotation on it;  that is, it saves the first*
25:    *   character in the string, shifts all other characters to the left one*
26:    *   character position, then puts the saved character at the end.       *
27:    *   Repeated reads of memory where the string resides should result in: *
28:    *       ICE Tutorial Program                                            *
29:    *       CE Tutorial Program I                                           *
30:    *       E Tutorial Program IC   etc...                                  *
31:    *                                                                       *
32:    *   The program structure is described in the entry point function      *
33:    *   "main()"  described below.                                          *
34:    *************************************************************************/
35:   /* ---------------------------------------------------------------- *
36:    * NOTE:                                                            *
37:    * This program has a bug in it that will prevent the string characters *
38:    * from rotating as indicated above.  The bug is located in the function *
39:    * "rotate_message()", and is a very common programming error involving *
40:    * boundary conditions.                                             *
41:    * ---------------------------------------------------------------- */
42:
```

```
43:  /*****************************************************************************
44:   *                              DEFINES                                      *
45:   *****************************************************************************/
46:  #define FIRST_CHAR 0    /* index of first character in string to be rotated*/
47:  #define LAST_CHAR  21   /* index of last character in string to be rotated */
48:  #define TRUE   1
49:
50:  /*****************************************************************************
51:   *                              FUNCTIONS                                    *
52:   *****************************************************************************/
53:  /* ------------------------------------------------------------ *
54:   * The following C functions (procedures) must be declared      *
55:   * before they are used.  The term "void" means the function    *
56:   * does not return any data to the function that called it.     *
57:   * ------------------------------------------------------------ */
58:  void delay_a_little_bit(); /* the 1/3 second delay loop    */
59:  void initialize_buffer();  /* copies string to work buffer */
60:  void rotate_message();     /* performs marquee translation */
61:
62:  /*****************************************************************************
63:   *                              DATA                                         *
64:   *****************************************************************************/
65:  /* ------------------------------------------------------------ *
66:   * The text string, or array of ASCII characters, that will     *
67:   * undergo marquee transformation.                              *
68:   * ------------------------------------------------------------ */
69:  static char *message = {" ICE Tutorial Program "};
70:
71:  /* ------------------------------------------------------------ *
72:   * work buffer - string is rotated in here                      *
73:   * ------------------------------------------------------------ */
74:  char display_buffer[30];
75:
```

```
76:  /*****************************************************************************
77:   * FUNCTION:   main()                                                        *
78:   *                                                                           *
79:   * DESCRIPTION:                                                              *
80:   *    Entry point;                                                           *
81:   *       Copies string into work buffer so that program can be rerun;        *
82:   *       Loops forever:                                                      *
83:   *          Rotates message                                                  *
84:   *          Delays approximately 1/3 second at 12.5 MHz                      *
85:   *                                                                           *
86:   * PARAMETERS PASSED:  None                                                  *
87:   *                                                                           *
88:   * RETURNS:  VOID                                                            *
89:   *****************************************************************************/
90:  int
91:  main()
92:  {
93:     /* ------------------------------------------------ *
94:      * Copy message string to work buffer               *
95:      * ------------------------------------------------ */
96:     initialize_buffer(message,display_buffer);
97:
98:     /* ------------------------------------------------ *
99:      * and stay in this loop forever.                   *
100:     * ------------------------------------------------ */
101:    while (TRUE)
102:    {
103:       delay_a_little_bit();
104:       rotate_message();
105:    }
106: } /* end main() */
107:
```

```
108:  /*****************************************************************************
109:   * FUNCTION:   initialize_buffer()                                           *
110:   *                                                                           *
111:   * DESCRIPTION:  copies "ICE Tutorial Program" into work buffer              *
112:   *                                                                           *
113:   * PARAMETERS PASSED:                                                        *
114:   *   char *src;          * pointer to the start of the string to be copied   *
115:   *   char *destination;  * pointer to the start of the work buffer          *
116:   *                                                                           *
117:   * RETURNS: void                                                             *
118:   *****************************************************************************/
 119:  void
120:  initialize_buffer(src,destination)
121:  char *src;              /* where to get data from   */
122:  char *destination;      /* where to put the data    */
123:  {
124:     /* ------------------------------------------------------------- *
125:      * Copy until the end of the "ICE Tutorial Program" string is    *
126:      * encountered, determined by encountering a NULL character.     *
127:      *                                                               *
128:      * The following C statement translates:                         *
129:      * "while the character being copied is not equivalent to 0 (NULL),  *
130:      * read the character at the src address (*src), then increment the  *
131:      * src address (++), write the character to the character at the     *
132:      * destination address (*destination), and increment the         *
133:      * destination address (++)".                                    *
134:      * ------------------------------------------------------------- */
135:     while ( *destination++ = *src++ );
136:
137:  } /* end initialize_buffer() */
138:
```

```
139:  /*****************************************************************************
140:   * FUNCTION:   delay_a_little_bit()                                          *
141:   *                                                                           *
142:   * DESCRIPTION:                                                              *
143:   *   delays about 1/3 second  at 12.5 MHz                                    *
144:   *                                                                           *
145:   * PARAMETERS PASSED:  None                                                  *
146:   *                                                                           *
147:   * RETURNS: void                                                             *
148:   *****************************************************************************/
149:  void
150:  delay_a_little_bit()
151:  {
152:  unsigned long timer;   /* holds number of iterations to delay */
153:
154:      /* ------------------------------------------------- *
155:       * Initialize the loop timer, whose value has        *
156:       * been arrived at by trial and error                *
157:       * ------------------------------------------------- */
158:      timer = 0x15709L;
159:
160:      /* ------------------------------------------------- *
161:       * Stay here and decrement timer until the timer is 0.   *
162:       * ------------------------------------------------- */
163:      while (timer--);
164:
165:  } /* end delay_a_little_bit() */
166:
```

```
167:  /*****************************************************************************
168:   * FUNCTION:   rotate_message()                                              *
169:   *                                                                           *
170:   * DESCRIPTION:                                                              *
171:   *     Performs a "marquee" rotation on the string (as in theatre marquee),  *
172:   *     taking the current first character of the string, shifting the        *
173:   *     others left one position, and placing the saved character at the end. *
174:   *     E.g.,  ICE Tutorial Program                                           *
175:   *            CE Tutorial Program                                            *
176:   *            E Tutorial Program    etc...                                   *
177:   *                                                                           *
178:   * PARAMETERS PASSED:  None                                                  *
179:   *                                                                           *
180:   * RETURNS: void                                                             *
181:   *****************************************************************************/
182:  void
183:  rotate_message()
184:  {
185:  int index_ptr;      /* represents current index into string array    */
186:  char temp;          /* local storage for first character of string   */
187:
188:     /* ------------------------------------------------------------------ *
189:      * Save the first character in the display buffer.                    *
190:      * ------------------------------------------------------------------ */
191:     temp = display_buffer[FIRST_CHAR];
192:
193:     /* ------------------------------------------------------------------ *
194:      * Point to the second character in the array.                        *
195:      * Start moving characters from this point.                           *
196:      * ------------------------------------------------------------------ */
197:     index_ptr = 1;
198:
199:     /* ------------------------------------------------------------------ *
200:      * Shift the rest of the string one position to the left.             *
201:      * ------------------------------------------------------------------ */
```

```
202:     /* ----------------- BUG BUG BUG BUG BUG ------------------------------ *
203:      *  The while statement should be:                                     *
204:      *     "while (index_ptr <= LAST_CHAR)".                               *
205:      *  This bug prevents the last character in the string                 *
206:      *  from being moved.                                                  *
207:      * ----------------- BUG BUG BUG BUG BUG ------------------------------ */
208:     while (index_ptr < LAST_CHAR)
209:     {
210:        /* ------------------------------------------------------------ *
211:         * Since the display_buffer is an array, take the character      *
212:         * to the right of the current array position (index_ptr)        *
213:         * and move it into the next lower position...                   *
214:         * ------------------------------------------------------------ */
215:        display_buffer[index_ptr - 1] = display_buffer[index_ptr];
216:
217:        /* ------------------------------------------------------------ *
218:         * then increment the array index to the next character.         *
219:         * ------------------------------------------------------------ */
220:        index_ptr++;
221:     }
222:     /* ------------------------------------------------------------ *
223:      * Moved them all so put the first character into the last       *
224:      * character  position.                                          *
225:      * ------------------------------------------------------------ */
226:     display_buffer[LAST_CHAR] = temp;
227:
228:  } /* end rotate_message() */
```

2.3.5.2 ASM86 Program Listing

```
  name MESSG
; line 1: source file 'MESSG.C'

  impure

message_:

  strings

L1:
  db      020h
  db      049h
  db      043h
  db      045h
  db      020h
  db      054h
  db      075h
  db      074h
  db      06Fh
  db      072h
  db      069h
  db      061h
  db      06Ch
  db      020h
  db      050h
  db      072h
  db      06Fh
  db      067h
  db      072h
  db      061h
  db      06Dh
  db      020h
  db      00h

  impure

  dw      L1
; line 69: static 'message_' ptr:2 char:1

  bss
```

```
  public display_buffer_
display_buffer_:
  db     01Eh dup(0)
; line 74: 'display_buffer_' array:30 char:1

  code

  public main_
main_:
; line 92: 'main_' func int:2
  push     si
  push     di
  push     bp
  mov      bp, sp
; line 92: '{'
; line 96: ';'
  mov      ax, offset display_buffer_
  push     ax
  push     word ptr message_
  call     initialize_buffer_
  add      sp, 04h
L4:
; line 101: ';'
; line 102: '{'
; line 103: ';'
  call     delay_a_little_bit_
; line 104: ';'
  call     rotate_message_
; line 106: '}'
  jmp      word ptr L4
; line 106: '}'
  pop      bp
  pop      di
  pop      si
  ret
```

```
  public initialize_buffer_
initialize_buffer_:
; line 121: 'initialize_buffer_' func void:0
; line 121: 8 'src' ptr:2 char:1
; line 122: 10 'destination' ptr:2 char:1
  push      si
  push      di
  push      bp
  mov       bp, sp
L7:
; line 123: '{'
; line 135: ';'
  mov       di, word ptr [bp+0Ah]
  inc       word ptr [bp+0Ah]
  mov       si, word ptr [bp+08h]
  inc       word ptr [bp+08h]
  movb      al, byte ptr [si]
  movb      byte ptr [di], al
  orb       al, al
; line 135: ';'
  jne       word ptr L7
; line 137: '}'
  pop       bp
  pop       di
  pop       si
  ret
  public delay_a_little_bit_
delay_a_little_bit_:
; line 151: 'delay_a_little_bit_' func void:0
  push      si
  push      di
  push      bp
  mov       bp, sp
  sub       sp, 04h
; line 151: '{'
; line 152: auto -4 'timer' ulong:4
; line 158: ';'
  mov       word ptr [bp-04h], 05709h
  mov       word ptr [bp-02h], 01h
```

```
L10:
; line 163: ';'
  mov       ax, word ptr [bp-04h]
  mov       dx, word ptr [bp-02h]
  sub       word ptr [bp-04h], 01h
  sbb       word ptr [bp-02h], 00h
  or        ax, dx
; line 163: ';'
  jne       word ptr L10
; line 165: '}'
  mov       sp, bp
  pop       bp
  pop       di
  pop       si
  ret
  public rotate_message_
rotate_message_:
; line 184: 'rotate_message_' func void:0
  push      si
  push      di
  push      bp
  mov       bp, sp
  sub       sp, 03h
; line 184: '{'
; line 185: auto -2 'index_ptr' int:2
; line 186: auto -3 'temp' char:1
; line 191: ';'
  movb      al, byte ptr display_buffer_
  movb      byte ptr [bp-03h], al
; line 197: ';'
  mov       word ptr [bp-02h], 01h
L13:
; line 208: ';'
  cmp       word ptr [bp-02h], 015h
  jge       word ptr L12
; line 209: '{'
; line 215: ';'
  mov       si, word ptr [bp-02h]
  mov       di, word ptr [bp-02h]
  movb      al, byte ptr [si+display_buffer_]
  movb      byte ptr [di+display_buffer_-01h], al
```

```
; line 220: ';'
  inc       word ptr [bp-02h]
; line 226: '}'
  jmp       word ptr L13
L12:
; line 226: ';'
  movb      al, byte ptr [bp-03h]
  movb      byte ptr display_buffer_+015h, al
; line 228: '}'
  mov       sp, bp
  pop       bp
  pop       di
  pop       si
  ret
```

Contents
Chapter 3 Command Language

Chapter 3
Command Language

3.1 Introduction

The emulator command language provides many features to assist you in entering commands, handling files, and managing a customized debug environment. The following sections describe how to use these features.

3.2 Entering Commands

The command entry features include the syntax menu, command line editing, the command history buffer, help mode, extending a command to another line, aborting a command, multiple commands on a line, comments, redirection commands, the pipe command, the escape operator, and block commands.

3.2.1 Syntax Menu

The syntax menu displays a row of options that you can enter. If you follow its choices, you cannot construct a syntactically incorrect command, although it may be semantically incorrect.

When you invoke the emulator software, the first line of the menu appears on the bottom of the screen. If all of the options do not fit on the screen at one time, the flag "----more----" appears on the line above the syntax menu. Call up subsequent lines by pressing the <Tab> key. The menu is circular in one direction. Pressing <Tab> will eventually cause the first line of options to reappear. An option need not be displayed for you to enter it.

Each line contains a list of choices. The command language keywords are shown in upper-case letters. The menu also contains lower-case entries enclosed in angle brackets. These represent user-defined names, data, or a set of emulator keywords. For example, <wild-identifier> can represent the name of a debug variable that has been

defined with the DEFINE command. The entry <expr> represents an expression. Also, <data type> represents one of the keywords identifying an data (or memory) type, such as BYTE (or ORD1), or WORD (or ORD2). Every keyword in the syntax menu is listed as a separate entry in Chapter 6.

If you enter a space after you enter an option, the menu displays the next level of options. As shown in the following example (where the underscore represents the cursor), when you enter the keyword HELP and follow it with a space, the new menu is displayed.

```
hlt> HELP _
----more----
<eol>   <help_item>   ERR   ;
```

In the syntax menu, the option <eol> indicates end-of-line, or the <Enter> key, and the semicolon (;) indicates command delimiter, which means you can enter another command after the semicolon. When these options appear, the command is complete and can be executed.

While you have an incomplete command entered on the command line, you can return to higher menu levels. If, after you enter the space, you press the <Backspace> or left cursor key, the menu returns to the previous level, the one from which you chose HELP in the preceding example.

If you enter the keyword DEFINE and follow it with a space, the new menu displayed is as follows. (The underscore represents the cursor position.)

```
hlt> DEFINE _
----more----
LITERALLY  STATIC  <DATA TYPE>  POINTER  TOOL  SYMBOLGROUP
```

Press the <Tab> key to see the rest of the menu.

```
----more----
KEYWORDGROUP  SYMBOLGROUPLINK  SYMBOLFILE  PROC
```

Pressing the <Tab> key again returns you to the first part of the menu.

If an incorrect character is entered, the words "syntax error" appear immediately above the syntax menu and a beep will sound. Even though more characters may be entered, the error message remains until the cursor is moved to the left of the incorrect character. If you press the <Enter> key with a syntax error in the command line, the emulator software issues the following error message:

```
Unexpected input:
.... "<user-input>"
```

If you press <Enter> with a semantically incorrect command, the emulator software issues an error message in a format that includes the origin of the error message, the error number, and useful information on what may have caused the error. The format of an error message is:

```
[source-id] number
message
```

Where:

[*source-id*]	specifies the origin of the error message. Source options are: BUILTIN_FUNCTION DBM ICE186 KERNEL KERNEL_CMD MATH_FUNCTION
number	specifies the error message number.
message	describes what caused the error.

If the error message is terminated with [*], extended help for that error is available. You can enter the command "HELP ERR <*source_id*> <*error_number*>" to display the extended information for the error (see Section 3.2.3).

The syntax menu does not display all of the keywords that are available, such as the names of individual registers, parameters, or built-in functions. See Chapter 6 for a complete listing of all emulator command keywords.

The emulator recognizes LITERALLY definitions. If you defined a LITERALLY "sa" to abbreviate SAVE, entering "sa" followed by a space displays the SAVE menu.

At any time, you can activate or deactivate the syntax menu by pressing <Ctrl>V. (Whether the syntax menu is activated when you invoke the emulator software depends on the optional SYNTAXGUIDE={YES | NO} specification in the operating environment configuration file discussed in Section 3.3.1.1. The default setting is YES.)

3.2.2 Command Line Editing

To edit commands, use the control character sequences or special key functions described in Table 3-1. These line-editing keys are available for editing command lines currently under construction or recalled from the command history buffer.

Table 3-1 Special Key Functions

Key Stroke	Description
←	Moves the cursor to the left.
→	Moves the cursor to the right.
↑	Recalls and displays the previous command in the history buffer.
↓	Moves forward and displays the next most recent command in the history buffer.
<Home>	Moves the cursor to the leftmost or rightmost column. If the cursor is already in the leftmost column of the command line, it is moved to the rightmost end of the command line.
<Ctrl>A	Erases all characters to the right of the cursor as well as the character at the cursor position.
<Ctrl>X	Erases all characters to the left of the cursor.

Table 3-1 Special Key Functions (continued)

Key Stroke	Description
<Backspace>	Erases the character immediately to the left of the cursor and, unless the cursor is in the leftmost column already, shifts the remainder of the command line to the left.
<Ctrl>L	Erases the current command line.
<Del>/<Ctrl>F	Deletes the character at the cursor position.
<Ctrl><Break>	Aborts the command under construction and terminates any command in progress.
<Ctrl>C	Aborts the command under construction and terminates any command in progress.
<Ctrl>V	Activates and deactivates the syntax menu.
<Tab>	Pages through the syntax menu.
<Ctrl>S	Stops and resumes the output to the screen.
<Ctrl>B	Activates and deactivates the help mode.
<Ctrl>R	In help mode, scrolls forward one screen of help text.
<Ctrl>K	In help mode, scrolls forward one line of help text.
<Ctrl>U	In help mode, scrolls backward one screen of help text.
<Ctrl>O	In help mode, scrolls backward one line of help text.
<Enter>	Marks the end of input line (shown on the syntax menu as <eol>).

3.2.3 Help Mode

The help mode provides descriptions of commands, command usage examples, and references to related topics. The help mode is activated and deactivated by toggling <Ctrl>B when the syntax menu is activated. (If the syntax menu is deactivated, help mode is not available.)

Pressing <Ctrl>B opens a help window on the top half of the screen with the message "<help mode>" at the top left corner. The default help window size is eight lines. You can change it with the HELPSIZE=*number* specification in the operating environment configuration file discussed in Section 3.3.1.1. Pressing <Ctrl>B again deactivates the help mode, but the screen is not cleared. The windowed text will scroll off the screen with subsequent commands. To clear the screen, use the CLEAR() built-in function.

Screens of text in the help window can be scrolled forward by pressing <Ctrl>R and backward by pressing <Ctrl>U. (To see this, type DEFINE and a space; do not press <Enter>. Then use <Ctrl>R and <Ctrl>U.) Lines of text in the help window can be scrolled forward by pressing <Ctrl>K and backward by pressing <Ctrl>O.

When the help mode is active, the current command-line prompt remains at the bottom of the screen. You can continue entering commands as before, but the help window is filled with the help text that corresponds to the keyword entered at the prompt.

For example, you can enter the command HELP, followed by a space. The options appear in the syntax menu, but help text also appears in the help window, as follows. (This example was created with HELPSIZE set to 15.)

```
Description:

The HELP command displays information about a keyword, command, or
error message.  For screen handling, CTRL-S stops the screen from
scrolling, while any key resumes the screen scrolling.
In addition to an on-line HELP command, there is a help mode that
can be enabled/disabled by toggling CTRL-B.  When in help mode,
help information is displayed in a window at the top of the screen
for the various keywords as they are entered.  The size of the window
may be modified by using the HELPSIZE = <number> directive in the
configuration file at invocation time.

Examples:

------------------------------------------------------------------------
hlt> HELP _
----
<eol>   <help_item>   ERR   ;
```

To see the rest of the help text, press <Ctrl>R. The next screen appears as follows.

```
Examples:

1. Request additional help for an error that has extended information.
   A [*] symbol following the error text indicates that extended
   information is available.
-> 3++

[kernel] Error #27t
Illegal operation.[*]
->HELP ERR kernel 27t
...error help text...
->

2. Request information about the keyword TO.
-> HELP TO
```

Again, to see the rest of the help text for the HELP command, press <Ctrl>R.

```
2. Request information about the keyword TO.
-> HELP TO
...help text ...
->

SEE ALSO:

ERR
```

To scroll back through the help text, press <Ctrl>U.

In addition to the window help mode, help can be obtained for a specific topic by entering the command "HELP <help_item>". The available help items can be displayed by entering the command HELP with no options.

The HELP command can also be used to obtain extended information about a specific error number. If an error message is followed by [*], the error has extended information available. The following example obtains additional help for kernel error 27t.

```
hlt> HELP ERR kernel 27T

kernel error #27t:  Illegal operation.[*]
Left operand type unexpected for operation.
Example:  3++ ;  3 = 2  ;  a++ = 2
```

Note that error numbers are displayed in decimal form. When requesting help for an error, always add the T suffix to the error number if the BASE parameter is not set to decimal. If the T suffix is not used and BASE is not decimal, the error number will not be interpreted as you expect. For example, a message will be displayed indicating that the error number was not found or information about an unrelated error will be displayed.

3.2.4 Command History Buffer

The emulator software stores up to twenty previously issued commands in a last-in, first-out command history buffer. Using this buffer, you need not enter a previous command to re-execute it or change it.

Scroll through the buffer by pressing the up-arrow and/or down-arrow key until you reach the command you want, and use the line-editing keys to modify the command. Pressing the <Enter> key executes that command. The new version of the command becomes the latest entry in the buffer while the old version is still in its original place.

The size of the history buffer can be controlled at software invocation time in two ways. The setting of the HISTORY = *number* specification in the operating environment configuration file (described in Section 3.3.1.1) and the "-h *number*" option used on the invocation line (described in Section 3.3.2) both control the number of history buffer entries. The "-h *number*" option has precedence.

3.2.5 Extending a Command to Another Line

Continue a command to the next line by entering the backslash (\) and pressing the <Enter> key, which is referred to as <eol> in the syntax menu that appears at the bottom of the screen.

Continue commands across multiple lines by pressing <Enter> prematurely. If <Enter> (or <eol>) is not a valid option at that point in the command line, the command is automatically continued. The continuation prompt is indicated by an additional greater-than symbol (e.g., hlt>> or .hlt>>).

3.2.6 Aborting a Command

Abort a command by entering <Ctrl><Break>. Note that aborting a command has no effect on emulation. Emulation continues until you enter the HALT command.

3.2.7 Multiple Commands on a Line

To enter more than one command on a line, separate the commands with a semicolon (;). When the semicolon appears in the syntax menu, it indicates that the command is complete and can be executed by pressing <Enter>, or another command can be added to the line after a semicolon. The following example shows two commands on the same line.

```
hlt> REGS ; ISTEP
```

3.2.8 Comments

Enclose a comment within a slash-asterisk combination. The symbol /* begins the comment, and the symbol */ ends the comment. Comments may be nested, and they may be extended across line boundaries. The following examples show how comments can be used in a debug session.

```
hlt> /* This program was debugged with ICE-186. */
hlt> /* This comment is
hlt>> spread over two lines. */
hlt>
hlt> /* Nested comments are good /* to comment out */ parts
hlt>> of debug procs that contain comments */
hlt>
hlt> VERSION     /* Display software version
hlt>> information. */
```

NOTE

The continuation prompt reminds you to enter the closing asterisk-slash (*/).

3.2.9 Redirection Commands

The emulator software allows you to redirect the output of a command to a file or a device by using the greater-than symbol (>) as the redirection character. To append output to an existing file, use two greater-than symbols (>>). Use a preceding grave (`) to distinguish redirection from expression operators. Use the exclamation character (!) to suppress error messages and cause the specified action to be taken, as shown in the following examples.

`hlt> DIR LITERALLY `> F1`	If the file already exists, an error message is displayed.
`hlt> DIR LITERALLY `>! F1`	No error message is displayed and the file is written.
`hlt> DIR LITERALLY `>> F2`	If the file does not exist, an error message is displayed.
`hlt> DIR LITERALLY `>>! F2`	No error message is displayed and the file is appended to.

Input can also be redirected from a file or from the terminal. When input is redirected from a file, use the grave and less-than symbol combination (`<), as in the following example.

```
hlt> SCANF("%d",&i) `< F3
```

A file named F3 is used as input.

To redirect input from the terminal, use the terminal input redirection notation (‘<<) and a key character or word that identifies the beginning and ending of the text entered. In the following example, "a" is used as the key character.

```
hlt> DEFINE NSTRING ns
hlt> GETS (&ns) `<<a
hlt>> This is my string
hlt>> a
hlt>> <Enter>
hlt> ns
"This is my string"
hlt>
```

3.2.10 Pipe Command

The emulator software allows piping the output of one command into a second command. The pipe notation is the vertical bar (|) preceded by a grave (‘) to distinguish it from expression operators, as shown in the following example.

```
hlt> DEFINE INT2 i2 = 55T
hlt> DEFINE NSTRING ns
hlt> PRINTF("DEBUGVAR i2 = %d", i2) `| GETS(&ns)
hlt> ns
"DEBUGVAR i2 = 55"
hlt>
```

3.2.11 DOS Escape Operator

Enter a DOS command by preceding it with the at-sign (@), which is the DOS escape operator, as shown in the following example:

```
hlt> @ DIR
Volume in drive C is BASE_DIR
Directory of     C:\TESTDIR

.              <DIR>       12-23-86     4:30p
..             <DIR>       12-23-86     4:31p
TEST1     DOC          384  12-23-86     4:33p
TEST2     DOC          384  12-23-86     4:34p
        4 File(s)        14819328 bytes free
```

3.2.12 Block Commands

A block command sets up a group of emulator commands to be executed as a block, that is, none of the commands in the group are executed until the terminating keyword is reached. In the C language, this is called a control construct. The block commands begin with the first keyword shown in the following lists.

I²ICE-Compatible	C-Compatible
DO_ ... END	DO ... WHILE
COUNT ... END	SWITCH
IF_ ... THEN ... ELSE ... ENDIF	IF ... ELSE
REPEAT ... END	WHILE
	FOR

C-compatible control constructs cannot be used inside I²ICE block commands (e.g., you cannot use the SWITCH construct inside a COUNT ... END block).

All emulator commands can be included within a block command except the following:

EDIT
INCLUDE

The DO_/DO block executes all commands. The COUNT, SWITCH, IF_, IF, WHILE, REPEAT, and FOR blocks permit test conditions that determine which commands are executed. The IF_/IF block conditionally selects a group of commands. The REPEAT or FOR block executes a group of commands indefinitely or until an exit condition occurs. COUNT enables you to specify the maximum number of times the command group is executed, and SWITCH enables you to select which command group is executed.

For the I^2ICE-compatible block commands, a period (.) before the continuation prompt (e.g., .hlt>>) indicates that the emulator software recognizes the beginning of a block command and has not yet detected the terminating keyword. The following example shows a block command that steps through five assembly language instructions, beginning at the current execution point, and evaluates the source-code statement number for each instruction.

```
hlt> COUNT 5
.hlt>> ISTEP
.hlt>> EVAL $ LINE
.hlt>> END

... results are displayed...

hlt>
```

For the C-compatible commands, the continuation prompt (e.g., hlt>>) indicates that the emulator software recognizes the beginning of a block command and has not yet detected the terminating keyword or brace (}).

C-compatible block commands that contain more than one command are delimited by braces ({}).

3.3 File Handling

The next three sections describe how to manage configuration files, list files, include files, and the APPEND, PUT, LOAD, and SAVE commands, which operate on files.

3.3.1 Configuration Files

Three configuration files supply parameter settings for the operating environment, for the emulator, and for communication. These files are normally edited by the SETUP program; however, you can bypass the SETUP program and edit the files directly.

Some of the parameter specifications in these files are required and should not be removed. Others are optional. The following subsections describe these files.

Note that if only the filename is specified for *pathname* in any configuration file specification, it is assumed that the file exists or is to be created in the current working directory.

See the *ICE™-186/188 In-Circuit Emulator Installation Supplement* for more information on SETUP.

3.3.1.1 Operating Environment Configuration File (Default: I186.CFG)

The operating environment configuration file, I186.CFG, is automatically searched for and processed during software invocation.

By using the "-c *pathname*" invocation option described in Section 3.3.2, you can specify an alternate operating environment configuration file. An alternate file must contain all required specifications, which include:

GMR = *pathname* specifies the name of a file used by the emulator software. You can change the path portion of *pathname* to reflect the location of the file, but you should not modify the filename itself.

Optional specifications include:

RESTORE = *pathname*

restores a previously saved database file. If this option is used, it must be the first specification in the configuration file. (See also PUT and EXIT in Chapter 6.)

SYNTAXGUIDE = {YES | NO}

specifies whether the syntax menu is active or not. The default setting is active.

HISTORY = *number* specifies the maximum number of command entries saved in the history buffer. Range is 1 to 20. The default setting is 20.

HELPSIZE = *number* specifies the number of lines used by the window help mode. Range is 2 to 20. The default setting is 8.

INCLUDE = *pathname*

causes the emulator commands in the specified file to execute at invocation time. You can have more than one INCLUDE specification, as described in Section 3.3.4.

LOGFILE = *pathname* records all emulator command interaction with the terminal to the specified file, as described in Section 3.3.3.

Note that you can override RESTORE and HISTORY specifications by using an invocation option as described in Section 3.3.2.

3.3.1.2 Emulator Environment Configuration File (Default: ICE186.CFG)

The emulator environment configuration file, ICE186.CFG, contains the tool configuration specifications for a user-defined tool. The emulator is defined as a tool for debugging when the command "DEFINE TOOL ICE186 = ICE186.CFG" is executed.

Required specifications include:

GMR = *pathname*
VAR = *pathname*
PTR = *pathname*

identify files used by the emulator software. You can change the path portion of pathname to reflect the location of the file, but you should not modify the filenames themselves.

TIP_SERVER_NAME = TIP

specifies a parameter used by the emulator software. This specification should not be modified.

IO_DEVICE = {GPIB | COM1 | COM2}

specifies the host-to-emulator communication device.

TIP_CONFIG_FILE = *pathname*

specifies the communications configuration file (see Section 3.3.1.3).

3.3.1.3 Communications Configuration File (Default: V186.CFG)

The communications configuration file, V186.CFG, characterizes host-to-emulator communications. Specifications include:

BAUD = *baud rate* specifies the baud rate to be used for serial communication. Default is 38400. Range is 110 - 57600.

GPIB_DEVNAME = *device name*

specifies the GPIB device name of the emulator. The device name must match the name assigned to the emulator in the device map of the GPIB-PC2A IBCONF.EXE setup program. The default in IBCONF.EXE is DEV1.

QSTAT = {1 | 0} specifies whether the emulator defaults to queue status mode (QSTAT=1) or normal mode (QSTAT=0). This variable is optional and can be deleted from the configuration file. If the variable is deleted, the emulator automatically prompts you for a QSTAT setting each time the software is invoked.

You can override the default QSTAT setting by using the QSTAT tool variable once the emulator software has been invoked (see QSTAT in Chapter 6).

3.3.2 Invocation Options

The following options can be used on the emulator software invocation line or by modifying the RUN186.BAT file to perform certain actions at invocation time, to override certain default settings, or to override certain configuration file parameter specifications. Each option must be preceded by a hyphen (-).

-c *pathname* — specifies an alternate operating environment configuration file to be used instead of the default I186.CFG configuration file. If you use this option more than once, only the last occurrence is observed.

-h *number* — specifies the number of history buffer entries to be maintained. If you use this option more than once, only the last occurrence is observed. The number is interpreted in base 10. The range is 1 to 20.

-i *pathname* — causes the emulator commands in the specified file to execute after initialization. If you use the "-i" option more than once, the files are processed in left-to-right order. Any INCLUDE specification options used in the operating environment configuration file are processed before files included with the "-i" option. (See also Section 3.3.4.)

NOTE

If there are commands in the file included with the "-i" option that require the emulator to be defined and activated prior to execution, ensure that the emulator configuration file has been defined as a tool and activated before the commands are executed. For example: "DEFINE TOOL ICE186 = ICE186.CFG ; ACTIVATE ICE186".

-r *pathname* restores a previously saved database file. The file specified with the "-r" option will override the RESTORE specification option used in the operating environment configuration file. (See the APPEND, PUT and EXIT commands in Chapter 6 for more information on saving a database file.)

NOTE

When a debug procedure (PROC) is defined, a temporary file is used to store information about the PROC. The directory where this temporary file is stored is dependent on the designation of the operating system TMP or TEMP environment parameters. The full pathname of the temporary file is recorded in the database. When a binary representation of the database is saved, this full pathname is also saved. If the database is restored on a system that does not contain this path, attempts to execute or read the contents of the PROC will result in an error.

3.3.3 List Files

List files (or log files) record all emulator command interaction with the terminal during a debug session and save it in the specified file. You can append data to an existing file, overwrite an existing file, or create a new list file by specifying a unique filename.

To create a new list file, enter the command "LIST *pathname*". You should close a list file before opening a new one. If the list file is opened successfully, the emulator prompt will appear without any messages. If you enter EXIT without entering NOLIST, the file is automatically closed.

To discontinue sending data to the list file, enter NOLIST. If you attempt to record data to a new list file, the currently open list file is automatically closed.

A list file can be opened at software invocation by using the LOGFILE=*pathname* specification in the operating environment configuration file. If the named file already exists, it is automatically appended to. If you enter the NOLIST command during the debug session, or if you enter the EXIT command without entering NOLIST, the file is closed.

3.3.4 INCLUDE Files

Include files (or submit files) are text files that contain emulator commands. You can create an include file with a text editor. Commands in the include file can be executed at software invocation by specifying the file in the operating environment configuration file (INCLUDE=subr1.tst), or by using the "-i" option followed by the name of the file on the invocation line (C> I186.EXE -i subr1.tst). See Sections 3.3.1.1 and 3.3.2 for additional information. Commands in an include file can also be executed during a debug session by entering the name of the file with the INCLUDE command, as shown in the following example.

```
hlt> INCLUDE subr1.tst
```

3.3.5 LOAD and SAVE Commands

Use the LOAD command to transfer object files or hexadecimal files into mapped memory. The object files must be in the Intel 8086 absolute Object Module Format (OMF) or Intel 8086 hexadecimal format.

To utilize the full symbolics capabilities of the emulator, modules must be compiled without optimization and with the compiler switches for debug information and symbols on.

The SAVE command saves the contents of a specified memory partition to the file specified by a pathname. Use SAVE to save assembly-level patches for future debugging sessions.

3.4 Customizing the Debug Environment

The emulator software includes a sophisticated database manager that maintains centralized storage of all the objects in the debug environment. These objects include all of the command keywords, parameters, and debug objects as well as other objects defined by the emulator software. You can customize your debug environment by manipulating these objects in a variety of ways. The rest of this chapter describes the object types, object names, object hierarchy, and the emulator commands used to manipulate them.

3.4.1 Object Types

An object type is similar to a data type in that the type of an object determines attributes of the object. In other words, the data type ORD1 identifies an 8-bit memory space that is typed as ordinal and can contain a value from zero up to 255. An object is typed, can contain a signed or unsigned value, and can be used in an expression.

The type of an object is also the name of the directory that contains the objects of that type. For example, KEYWORD is an object type, and the subdirectory KEYWORD// contains all of the emulator keywords. The double slash symbol (//) is recognized by the emulator software as a directory delimiter.

The objects can be referenced by their name, or by a full pathname that includes one of the main directories and the appropriate subdirectories.

3.4.2 Object Names

All object names used in emulator software are distinguished by a leading alphabetic character (A through Z and a through z), the underscore (_), or the dollar sign ($). Names can contain only alphanumeric characters (any alphabetic character and 0 through 9), or the underscore (_). Case is not significant, that is, a name can be entered in upper- or lower-case letters. Object names are grouped in the following categories:

- Keyword
- Parameter
- Debug objects
  - LITERALLY
  - variable
  - PROC
- Tool variable
- Tool pointer
- Program Symbolics

These categories are also used in pathnames for subdirectories containing the objects. Section 3.5 describes the object hierarchy and use of subdirectories in detail.

3.4.3 Keywords

Keywords are special names that are reserved elements of the emulator command language. Lead keywords are used to initiate a command, such as GO, ISTEP, or DEFINE. Other keywords such as FROM, TO, or DEBUG are used with the lead keywords. Some keywords, such as those corresponding to predefined functions, can occur in a variety of positions on the command line.

The emulator keywords are in the KEYWORD// subdirectory, and debug variables (which are user-defined) are in the DEBUG// subdirectory. Thus, it is possible to use a reserved keyword as a variable name as long as you use its full pathname. (However, we do not recommend this, and a message appears on the screen when you define an emulator keyword as a debug variable.)

For example, if you want to use GO as a variable, you can define it as a debug variable. To see its contents, use its fully qualified pathname as "DEBUG//GO". You would define it as a debug variable and reference it as follows:

```
hlt> DEFINE BYTE GO = 4T

WARNING: Specified debug object "go" has the same name as an existing
keyword. To avoid a syntax error while referencing this debug object, use
the "debug" prefix.

hlt> DEBUG//GO
4T
hlt>
```

3.4.4 Parameters

There are two types of parameters: local and tool. These parameters define and control the state of the emulator software. The subdirectory PARAMETER// contains the LOCAL// and TOOL// directories. Local parameters are available when the emulator software has been invoked, and tool parameters are available when the emulator has been defined. To list the local and tool parameters, enter the command DIR PARAMETER.

3.4.4.1 Local Parameters

The following are local parameters and are individual entries in Chapter 6.

BASE	specifies the number base for console input and output.
DISPLAYFLAG	controls whether or not the value to be assigned to a variable is displayed. For example, if the expression "a+=b" is executed, the new value of "a" is displayed when DISPLAYFLAG is TRUE.
EDITOR	specifies which editor is invoked when the lead keyword EDIT is entered.
ERROR	specifies the display format for error messages.

NAMEPATH	specifies the search path used by the emulator command language when looking for an object.
SYMBOLIC	specifies whether addresses in certain commands should be converted to their equivalent symbolic names.

In addition to these parameters, the DIR PARAMETER command displays the following parameters: CASE, CURRENTTOOL, LASTTOOL, and PROMPT. These parameters are recognized internally and cannot be modified.

3.4.4.2 Tool Parameters

The tool parameters define and control the operation of the emulator and its resources as a debug tool. These parameters are briefly described here. See the entry for each of these parameters in Chapter 6.

$	specifies and displays both the address and the symbolic equivalent of the current execution point.
NAMEFRAME	is used to access different procedure activation frames on the stack.
NAMESCOPE	specifies and displays both the address and the symbolic equivalent of the code space memory pointer.
UPDATE	indicates whether the value of $ (execution pointer) should be used to update NAMESCOPE.

3.4.5 Debug Objects

You can define debug objects to simplify and enhance the debugging process. A debug object can be a LITERALLY, a debug variable, or a procedure. There are three subdirectories that correspond to these debug objects: LITERALLY//, DEBUGVAR// and PROC//.

3.4.5.1 LITERALLY Definitions

A debug LITERALLY definition is a user-defined customized name for a previously-defined character string (by emulator software or by the user). Debug LITERALLY definitions provide a means of abbreviating keywords, complete commands, program symbol names, name prefixes, or any token recognized by the emulator software.

To define a LITERALLY, use the DEFINE command as follows:

```
hlt> DEFINE LITERALLY lit = "LITERALLY"
hlt>
```

In this example, "lit" becomes an abbreviation for the keyword LITERALLY and can be used as shown in the following command:

```
hlt> DEFINE LIT H1 = "INCLUDE NOLIST TUDIR/HI_01"
hlt>
```

Use LITERALLY definitions to customize your debug environment and reduce the number of keystrokes needed for frequently-used commands.

3.4.5.2 Debug Variables

A debug variable is a user-defined variable used with emulator commands to store temporary values during a debug session. When you define a variable, you must specify its data type. Data types, allowable operations, and conversions are discussed in the <Data type> entry in Chapter 6.

To define a debug variable, use the DEFINE or REDEFINE command and specify the data type. The following example defines "var1" as a debug variable of the type BYTE.

```
hlt> DEFINE BYTE var1
hlt>
```

To assign a value to a variable when defining it, use the equal sign followed by the value, as follows:

```
hlt> DEFINE BYTE var1 = 35T
hlt>
```

Once a variable has been defined, change its contents by entering the variable name and assigning the new value:

```
hlt> var1 = 14T
hlt>
```

To change the definition of "var1", use REDEFINE as shown in the following example:

```
hlt> REDEFINE DWORD var1
hlt>
```

The predefined emulator keywords reside in directories automatically set up by the emulator software. These directories are searched in the order they appear in the NAMEPATH parameter. If the directory KEYWORD// precedes DEBUG//, and you want to use a reserved keyword as a variable name, reference it by its full pathname, that is, DEBUG//*user-var*. For example, to use the keyword CONSTANT as a variable, define it as follows:

```
hlt> DEFINE BYTE constant
hlt>
```

The emulator software automatically places the variable "constant" in DEBUG// and thereby distinguishes it from the KEYWORD, CONSTANT. Remember to reference this variable by its full pathname; otherwise, it will be interpreted as a keyword.

3.4.5.3 Debug Procedures

A debug procedure (PROC) is a user-defined function that groups emulator commands. Create a PROC by specifying a name with an optional return type and an optional argument list. The return type is any data type or pointer type. The argument list is a set of identifiers separated by commas. Each argument can be defined to be converted to a given data type or pointer type. If an argument is not defined, it defaults to the caller's type.

To define a PROC, use the DEFINE or REDEFINE command with the keyword PROC. To define a debug object within a PROC, use only DEFINE.

```
hlt> REDEFINE PROC INT4 max(a,b)
hlt>> DEFINE INT4 a
hlt>> DEFINE INT4 b
hlt>> {
hlt>> IF (a>b)
hlt>> RETURN a ELSE RETURN b
hlt>> }
hlt>
```

A PROC is executed whenever its name occurs in a command line. Use the PROC name as a keyword. To execute the PROC shown above, which is defined as "max", enter its name and two numbers, for example, max (5,57).

3.4.5.4 Built-In Functions

The emulator software supports six types of built-in functions:

- Math functions, such as ABS(x)
- String functions, such as STRCAT(*&arg1, arg2*)
- Character classification and transformation functions, such as ISALPHA(c)
- Input/output functions, such as GETCHAR()
- Screen functions, such as CLEAR()
- Miscellaneous functions, such as TIME()

Most built-in functions operate on debug variables, tool variables, program symbolics, and memory addresses. See the function entries in Chapter 6 (e.g., Math Functions or String Functions) for more information on each function.

3.4.5.5 Saving Debug Objects

Use the PUT and APPEND commands to save debug objects (PROCS, variables, and LITERALLY definitions) in text files or to capture an interpreted binary representation of the database. Text files can be restored with the INCLUDE command. Binary files can be restored with the "-r" invocation option, or by using the RESTORE option in the operating environment configuration file. (See the PUT and APPEND entries in Chapter 6.)

3.4.6 Tool Variables

A tool variable is a name that refers to a resource provided by the emulator. For example, the CPU registers (AX, FLAGS, IP) are hardware resources that contain values. They are called tool variables and reside in the TOOLVAR// subdirectory automatically set up by the emulator software.

3.4.7 Tool Pointers

A tool pointer is a name that refers to an object of type pointer. A tool pointer consists of numerous components such as segments and byte offsets. The pointer type defines the length and interpretation of these components. The emulator software supports a variety of pointer types and identifies each one by the following object names: OFFSET, PHY, LPOINTER. Tool pointer types reside in the TOOLPTR// subdirectory. See the POINTER section in the <data type> entry of Chapter 6 for information on creating and using objects of type pointer. See the POINTER entry in Chapter 6 for information on accessing memory using tool pointer types.

3.4.8 Program Symbols

Program symbols are the procedure, block, constant, literal, label, variable, and variable type objects that correspond to your program source code.

The emulator software recognizes how source code is subdivided into modules. Each module consists of numbered lines of code for high-level languages, procedures, and data symbols. Additionally, there are symbols that are public across all modules.

When the LOAD command is executed, the emulator software loads the code and data from the specified OMF file into mapped memory. Symbolic information contained in the OMF file is extracted and placed in a symbol table file in the SYMBOLGROUP// subdirectory. If there are public symbols, they are placed in the PUBLIC// subdirectory. Otherwise, the MODULE//SYMBOL// subdirectory contains program symbols that reside within the module, and the MODULE//LINE// directory contains the line numbers (for high-level languages only).

The source code structure is reflected in the following object names.

For symbols within a module:

:mod_name	(e.g., :mod_x)
:mod_name#line_number	(e.g., :mod_x#3)
:mod_name.mod_symbol	(e.g., :mod_x.var1)
:mod_name.procedure	(e.g., :mod_x.proc_a)
:mod_name.level1_proc.level2_proc	(e.g., :mod_x.proc_a.proc_b)
:mod_name.procedure.variable	(e.g., :mod_x.proc.var)

For symbols that are declared public:

```
PUBLIC//pub_symbol
PUBLIC//pub.proc
```

The emulator software uses the parameter NAMESCOPE to allow the reference to a module symbol to be partially qualified. NAMESCOPE displays the current module or innermost procedure within a module whose scope encompasses the address stored in NAMESCOPE. Partially qualified references to program symbols will be resolved by searching for a match from the innermost scope level, as determined by NAMESCOPE, to the module scope level.

For example, assume that the emulator program symbol names correspond to a PL/M high-level language program fragment as follows and that NAMESCOPE contains an address that falls within proc_b (the NAMESCOPE symbolic reference becomes proc_b):

Table 3-2 Emulator Program Code and Symbol Names

PL/M fragment	Program symbols	Program lines
1 mod_x: do;	:mod_x	
2 declare var1 byte;	:mod_x.var1	
3 declare var2 byte;	:mod_x.var2	
4 declare var3 byte;	:mod_x.var3	
5 proc_a: procedure;	:mod_x.proc_a	:mod_x#5
6 declare var1 word;	:mod_x.proc_a.var1	
7 declare var5 word;	:mod_x.proc_a.var5	
8 proc_b: procedure;	:mod_x.proc_a.proc_b	:mod_x#8
9 declare var1 word;	:mod_x.proc_a.proc_b.var1	
10 declare var4 byte;	:mod_x.proc_a.proc_b.var4	
...		
13 end proc_b;		:mod_x#13
...		
17 end proc_a;		:mod_x#17
18 proc_c: procedure;	:mod_x.proc_c	:mod_x#18
19 declare var6 word;	:mod_x.proc_c.var6	
...		
21 end proc_c;		:mod_x#21
...		
27 end mod_x;		:mod_x#27

The "proc_a" symbols can be referenced as:

```
proc_a.var1 ("var1" would reference the variable "var1" in "proc_b")
var5
proc_b
```

The "proc_b" symbols can be referenced as:

```
var1
var4
```

The "proc_c" symbols must be referenced as:

```
proc_c.var6
```

The "mod_x" symbols can be referenced as:

```
:mod_x.var1 ("var1" would reference the variable "var1" in "proc_b")
var2
var3
proc_a
proc_c
```

The "mod_x" line numbers can be referenced as:

```
#5
#8
...
#27
```

The names of program symbols are organized to reflect their source code structure. To view the objects in a directory, specify the object type and then the pathname to the subdirectory of interest, as in the command: DIR *type pathname*.

The following examples show how to use the DIR command to display objects in various directories.

1. This example displays all keywords that begin with D in the local directory.

```
hlt> DIR KEYWORD d*
      LOCAL//KEYWORDGROUP//top
            DEFAULT
            DO
            DO_
      LOCAL//KEYWORDGROUP//rx_command
            DATABASE
            DEACTIVATE
            DEBUG
            DEBUGVAR
            .
            .
            .
```

2. This example loads a program named "messg2" located in directory "omf", and then displays all the symbols in the "messg2.sym" symbol file.

```
hlt> LOAD \omf\messg2
Symbol file "\omf\messg2.sym" is up-to-date.
.....................................................................
.........
Starting address: 1100:0004H
Symbol file "\omf\messg2.sym" attached to database.
hlt> DIR MODSYMBOL SYMBOLFILE//
TOOL//SYMBOLGROUPLINK//ice186.SYMBOLFILE//"\omf\messg2.sym":MESSG2
    MESSAGE_            ptr to int1
    DISPLAY_BUFFER_     array [0..49] of int1
    MAIN_               near proc returns int2
    INITIALIZE_BUFFER_  near proc returns null
        SRC_                    dynamic ptr to int1
        DESTINATION_            dynamic ptr to int1
    DELAY_A_LITTLE_BIT_ near proc returns null
        TIMER_                  dynamic ord4
    ROTATE_MESSAGE_     near proc returns null
        INDEX_PTR_              dynamic int2
        TEMP_                   dynamic int1
    PRINT_BUFFER_       near proc returns null
TOOL//SYMBOLGROUPLINK//ice186.SYMBOLFILE//"\omf\messg2.sym":INTERFACE
    PSTART_             near proc
    STACKTOP            ord2
```

3.4.9 Program Line Numbers

Line numbers are available only in high-level languages. Program line numbers are symbols that correspond to the source code line numbers. (See the example in Section 3.4.8 on referencing line numbers.)

3.5 Managing Debug Object Types

You can maintain the debug object types by using the following commands (which are separate entries in Chapter 6).

DEFINE	defines a new debug object.
REDEFINE	redefines an existing debug object or defines a new debug object.
REMOVE	removes a debug object.
SHOW	shows the definition of a debug object.
EDIT	edits an existing or new debug PROC using an editor of the user's choice.
EVAL	displays the program object (MODULE, LINE, SYMBOL, PROCEDURE) at the requested address.
DIR	lists the names of debug objects or program objects in the specified directory.
PUT/APPEND	saves debug objects to a disk file.
INCLUDE	redefines saved debug objects.

3.6 Wild Name Characters

The emulator supports the ? and * wild characters for referencing debug objects. The ? in a debug object name indicates that any character can occupy that position. The * in a debug object name indicates that any character can occupy that position and all the remaining positions in the debug object name.

Contents

Chapter 4 Emulation Features

Chapter 4
Emulation Features

4.1 Introduction

This chapter describes the emulation features. It shows you how to set up the target environment, control emulation, and access memory and register contents.

This section assumes you are familiar with command entry, as described in Chapter 3. All commands discussed in this chapter are described in detail in Chapter 6 of this manual.

4.2 Setting Up the Target Hardware

During emulation, you can connect the emulator user probe to the crystal power accessory (CPA) or to prototype hardware. If you connect the probe to the CPA, the emulator can simulate memory and I/O resources so that you can test program execution without the use of prototype hardware.

Procedures for connecting the user probe to the CPA or prototype hardware are provided in the *ICE™-186/188 In-Circuit Emulator Installation Supplement.*

4.2.1 Using the Crystal Power Accessory

For early software development and debugging tasks, you may want to use the crystal power accessory (CPA) as a signal source. The CPA permits emulation without using prototype hardware. You can debug your program and simulate memory and I/O resources before your hardware is complete.

A jumper (J1) on the CPA circuit board allows you to specify whether the 80C186 microprocessor runs in Compatible or Enhanced mode. Set this jumper to match the hardware configuration of your prototype. (Refer to the *ICE™-186/188 In-Circuit Emulator Installation Supplement* for instructions on jumper placement.)

When using the CPA, you must map all memory and I/O resources referenced by your program to emulator (ICE) memory and I/O space. The emulator provides 128K bytes of memory space and 64K bytes of I/O space. The mapped program cannot exceed 128K bytes and must be segmented into blocks appropriate for the ICE-mapped segments. Any memory or I/O operations not mapped to ICE will cause an emulator time-out (see the BUSACT, IORDY, and MEMRDY entries in Chapter 6).

Mapping procedures are described in Sections 4.3.1 and 4.3.2.

4.2.2 Using Prototype Hardware

When part or all of the prototype hardware is available, you may want to use the prototype instead of the CPA.

The same steps required to emulate with the CPA also apply to emulation with prototype hardware, except that you can map program memory and I/O resources to either emulator (ICE) or prototype (USER) memory.

4.3 Establishing the Target Environment

This section shows you how to establish the target environment. It describes how to map memory and I/O, set up tool variables, and load your program.

4.3.1 Mapping Memory

The emulator provides 128K bytes of ICE memory space. If your prototype hardware is not yet available, you can map all of program memory to ICE and emulate your program without hardware. On the

other hand, if your prototype hardware is available, you can map selected blocks of program memory between ICE and USER (prototype) memory.

Specify the memory map by using the MAP command. The MAP command syntax allows you to define memory allocation (ICE or USER) and access rights (read-only or read/write) in 4K-byte blocks.

You should define your memory map to correspond as closely as possible to the memory architecture of your prototype. It is wise to map all of the memory in your prototype, even though some parts of the memory may be unrelated to the memory you are debugging. If your prototype contains gaps in address space, your memory map should contain these same gaps. You can set the access rights of unused memory to read-only. This way, the emulator automatically catches any inadvertent writes to unused memory.

For example, suppose you want to create a memory map for the prototype memory architecture shown in Figure 4-1.

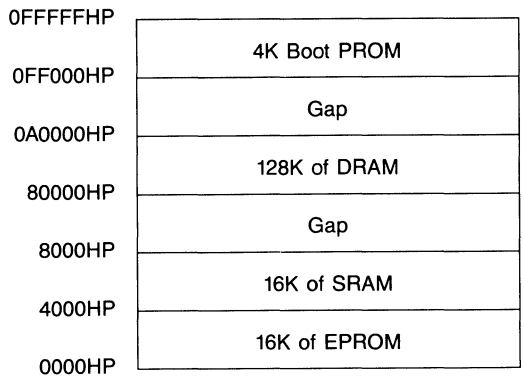


Figure 4-1 Sample Prototype Memory Architecture

Assume that prototype hardware is connected and that the purpose of the debugging session is to test the system firmware on the prototype. Your memory map might be:

```
hlt> RESET MAP  /* set to default */
hlt> MAP 0HP LENGTH 16K ICE READ  /* 16K EPROM */
00000HP   length     4000H    ICE    READ ONLY
04000HP   length    fc000H    USER   READ/WRITE
hlt> MAP 16KP TO 7FFFHP USER  /* 16K SRAM */
00000HP   length     4000H    ICE    READ ONLY
04000HP   length    fc000H    USER   READ/WRITE
hlt> MAP 08000HP TO 7FFFFHP USER READ  /* 1st gap */
00000HP   length     4000H    ICE    READ ONLY
04000HP   length     4000H    USER   READ/WRITE
08000HP   length    78000H    USER   READ ONLY
80000HP   length    80000H    USER   READ/WRITE
hlt> MAP 80000HP LENGTH 128K USER  /* 128K DRAM */
00000HP   length     4000H    ICE    READ ONLY
04000HP   length     4000H    USER   READ/WRITE
08000HP   length    78000H    USER   READ ONLY
80000HP   length    80000H    USER   READ/WRITE
hlt> MAP 0A0000HP TO 0FEFFFHP USER READ  /* 2nd gap */
00000HP   length     4000H    ICE    READ ONLY
04000HP   length     4000H    USER   READ/WRITE
08000HP   length    78000H    USER   READ ONLY
80000HP   length    20000H    USER   READ/WRITE
a0000HP   length    5f000H    USER   READ ONLY
ff000HP   length     1000H    USER   READ/WRITE
hlt> MAP 0FF000HP LENGTH 1000H ICE READ  /* 4K PROM */
00000HP   length     4000H    ICE    READ ONLY
04000HP   length     4000H    USER   READ/WRITE
08000HP   length    78000H    USER   READ ONLY
80000HP   length    20000H    USER   READ/WRITE
a0000HP   length    5f000H    USER   READ ONLY
ff000HP   length     1000H    ICE    READ ONLY
hlt>
```

As illustrated in the example, whenever you enter a MAP command, the resulting memory map is displayed on the screen. The MAP entries are cumulative.

Several parameters are available for expressing the mapped address blocks. You can specify an address block using either the TO or the LENGTH keyword. The starting addresses of the mapped 4K-byte blocks fall on 4K-byte boundaries (e.g., 0000HP, 1000HP, etc.). If you specify a starting address that falls between two of the boundaries, the starting address is rounded down to the lower 4K-byte boundary. The ending address of a LENGTH specification is rounded up to the nearest 4K-byte boundary.

The READ parameter specifies that the address block is assigned read-only access. If no READ parameter is included, the address block is assigned the default read/write access.

If you use the READ option for memory addresses mapped to ICE, you must program any chip-select register associated with the address range to recognize external READY. You do this by setting the R2 bit of the particular chip-select register to 0.

There are five chip-select registers located in the processor Peripheral Control Block. Three of the registers (MMCS, LMCS, and UMCS) are always used to control memory. The other two registers (MPCS and PACS) can be used to control either memory or I/O devices.

You can access the chip-select registers by using the emulator register access commands (see the Register Access entry in Chapter 6). The emulator keywords for the chip-select registers are CSCTRL5 (MPCS), CSCTRL4 (MMCS), CSCTRL3 (PACS), CSCTRL2 (LMCS), and CSCTRL1 (UMCS).

Note that if a read-only violation occurs in memory mapped to ICE, the emulator halts and the memory write is blocked. If a read-only violation occurs in memory mapped to the prototype, the emulator halts, but the offending memory write is not blocked. Thus, specifying that a mapped memory block is read-only prevents writes to ICE memory, but it does not prevent writes to read-only USER memory.

Use the RESET MAP command to restore the memory map to the default values. The default memory map directs all memory references to USER with read/write access.

4.3.2 Mapping I/O

The emulator provides 64K bytes of ICE I/O space. If your prototype hardware is not yet capable of handling I/O, you can map all I/O port addresses to ICE so that the emulator can simulate I/O accesses. On the other hand, if your prototype is capable of handling I/O, you can map selected blocks of I/O port addresses between ICE and USER (prototype) I/O space.

Note that if you plan to map I/O port addresses to ICE, you must program any chip-select register associated with the address range to recognize external READY. You do this by setting the R2 bit of the particular chip-select register to 0.

There are five chip-select registers located in the processor Peripheral Control Block. Two of these registers, MPCS and PACS, can be used to control I/O devices.

You can access the chip-select registers by using the emulator register access commands (see the Register Access entry in Chapter 6). The emulator keywords for the two I/O chip-select registers are CSCTRL5 (MPCS) and CSCTRL3 (PACS).

NOTE

All I/O outputs, whether through USER- or ICE-mapped I/O ports, appear on the pins of the emulator user probe. Ensure that any I/O accesses during emulation cannot harm hardware or software in the target environment.

You specify the I/O map by using the MAPIO command. The MAPIO command syntax allows you to map I/O port addresses in 4K-byte blocks and specify their location as either ICE or USER. You can define the address blocks using the *port* TO *port* format or the *port* LENGTH *number-of-ports* format.

For example, the following two command entries both map I/O port addresses 0000H - 7FFFH to ICE memory. The first entry uses the TO construct and the second entry uses the LENGTH specification.

```
hlt> MAPIO 0H TO 7FFFH ICE
0000H   length   8000H   ICE
8000H   length   8000H   USER

hlt> MAPIO 0H LENGTH 8000H ICE
0000H   length   8000H   ICE
8000H   length   8000H   USER
```

The starting addresses of the mapped 4K-byte blocks fall on 4K-byte boundaries (e.g., 0000H, 1000H, etc.). If you specify a starting address that falls between two of the boundaries, the starting address is rounded down to the lower 4K-byte boundary. The ending address of a LENGTH specification is rounded up to the nearest 4K-byte boundary.

Use the RESET MAPIO command to restore the I/O map to the default values. The default I/O map directs all I/O accesses to USER.

4.3.2.1 Handling ICE-mapped I/O

When I/O port addresses are mapped to ICE, the emulator simulates I/O accesses during emulation. It displays data values written from ports during I/O read cycles, and prompts you for data value inputs during I/O write cycles.

The following example illustrates a request for data value input:

```
hlt> QUERY=0
hlt> GO
  Port 0001H requests byte input (enter value):
```

Emulation stops and interrupts and "Hold" requests are ignored while an I/O request is pending. You must respond to I/O requests by entering a hexadecimal value in the form expected by your program code.

The HOLDIO command allows you to temporarily ignore a pending I/O request. You can then enter any emulator command that does not access the emulator or the processor. Enter the RELEASEIO command to return to the input data prompt.

Data values written from I/O ports are displayed as follows:

```
hlt> GO
    ffH written to IO Port 0002H.
```

The meaning of the output byte depends entirely on your program code. In the above example, 00FFH may represent TRUE or something else altogether. Remember that I/O port outputs will be numerical and may require further processing or interpretation before their meanings are clear.

To change or display the contents of byte-wide or word-wide I/O ports mapped to the prototype (USER), use the PORT and WPORT commands, respectively.

4.3.3 Setting the Tool Variables

Tool variables set up and control the emulator hardware and the processor. They are a cross between commands and variables. Like commands, tool variables initiate operations. Like variables, tool variables are named, have a value, can be assigned and displayed, and can be used in expressions.

Tool variables are predefined by the emulator software and cannot be removed. Their valid value ranges are also predefined. Their values can be changed only within the valid range.

Table 4-1 lists the tool variables. To display the current settings of all of these tool variables, enter the command TOOLVAR. Set and change these tool variables only when the emulator is halted. For more detailed information on each tool variable listed, refer to Chapter 6 of this manual.

Table 4-1 Tool Variables

Tool Variable	Description
ALEMODE	Specifies the timing of the ALE signal.
BTHRDY	Determines the source of the READY signal.
BUSACT	Enables/disables bus activity time-outs.
ENI	Determines the extent of interrupt servicing during halt mode.
FASTBREAK	Enables/disables fastbreak emulation.
IORDY	Enables/disables I/O access time-outs.
IPATINT	Determines the source of the interrupt signal used in iPAT interrupt latency measurements.
MEMRDY	Enables/disables memory access time-outs.
OSYNC	Changes the level of the synchronous output line with the next invocation of the GO command.
QSTAT	Specifies whether the emulator runs in normal or queue status mode.
QUERY	Controls the command-line prompt and emulator-status display during Run mode.
RSTEN	Enables/disables external reset of the processor during emulation.
VERIFY	Enables/disables verification of writes to memory.

4.3.4 Loading Your Program

Once the I/O and memory maps are defined, and the tool variables are set, use the LOAD command to load your program object code into mapped memory. The LOAD command gives you several options when loading the program. You can specify the completeness of the program load (everything, NOCODE, or NOSYMBOLS). Refer to Chapter 6 for more details about the LOAD command.

NOTE

Use only Intel 8086 absolute Object Module Format. Run-time locatable (RTL) and load-time locatable (LTL) programs are not supported.

4.4 Controlling Emulation

Once you have established the target environment and loaded your program, you are ready to begin emulation. This section shows you how to start, stop, and control the emulator. It also describes key emulation features such as the Activity Monitor and fastbreaks.

4.4.1 The Activity Monitor

The emulator capabilities are divided into two areas:

- Emulation -- which controls the execution of program code on the processor.
- Activity Monitor -- which controls the word recognizers, trace hardware, and OSYNC signal.

The Activity Monitor operates separately from emulation. You can stop the Activity Monitor, perform some action, display the trace, and restart the Activity Monitor -- without halting emulation. This means you can perform non-stop emulation while simultaneously performing non-intrusive program analysis.

Throughout the rest of this section a distinction is made between emulation and Activity Monitor functions. "Emulation" refers to emulator capabilities and functions, while "Activity Monitor" refers to program analysis capabilities and functions.

4.4.2 Using Fastbreaks

The first step in emulation is to specify whether or not you want to use fastbreaks. You do this with the FASTBREAK tool variable.

If you set FASTBREAK to TRUE (non-zero), the emulator supports a feature called "fastbreak" emulation. This is actually real-time emulation with a provision for emulation breaks that allows you to access 80C186 microprocessor memory or registers. After a fastbreak occurs, the emulator automatically resumes emulation.

If you set FASTBREAK to FALSE (zero), the emulator does not support fastbreaks. Emulation never stops unless forced to do so by a HALT command, a memory access violation, an I/O or memory access time-out, or an emulation break -- you cannot access processor memory or register contents during emulation.

Use fastbreaks only when the processor may be stopped for short periods. Do not use fastbreaks if you are emulating a system that requires continuous processor activity.

Note that during fastbreaks, the following processor functions are maintained:

- DMA transfers
- Refresh cycles
- Bus HOLD/HLDA

Interrupts that occur during a fastbreak are noted as pending by the emulator and are serviced at the completion of the fastbreak.

4.4.3 Starting and Stopping the Emulator

The emulator and Activity Monitor are controlled via three commands:

- GO -- starts and controls the Activity Monitor and emulation.
- STOP -- stops the Activity Monitor, but does not halt emulation.
- HALT -- halts emulation and stops the Activity Monitor.

The following paragraphs describe the use of these three commands.

4.4.4 Using the GO Command

The GO command controls the emulation session. With this command, you can define the starting emulation conditions, the word recognizers, and the actions to be taken as a result of word recognition.

The GO command has three basic elements, as diagrammed below:

GO [*init-emu*] [*init-AM*] [*AM-programming*]

Where:

init-emu	establishes the initial conditions of emulation.
init-AM	establishes the initial conditions of the Activity Monitor.
AM-programming	specifies the word recognition conditions and actions for the Activity Monitor.

Note that the above diagram does not represent the entire command syntax. For a complete description of GO syntax, refer to the GO entry in Chapter 6.

4.4.4.1 Initial Emulation Conditions

The *init-emu* conditions determine when and where emulation starts. You can specify that emulation starts FROM a specified address and/or WHEN the ISYNC signal goes high (ISYNCH).

If you use the FROM parameter, the emulator starts emulation at the specified address. If the emulator is running when you invoke the FROM parameter, it halts and then restarts at the specified address.

If you use the WHEN ISYNCH parameter, the emulator starts emulation when the ISYNC signal goes high. If the emulator is running when you invoke WHEN ISYNCH, it halts and then waits for ISYNC to go high.

The usual application for WHEN ISYNCH is to coordinate the simultaneous start of several emulators connected via the emulator ISYNC line. All emulators connected to the line will wait for ISYNC to go high before starting.

Once you have invoked the WHEN ISYNCH condition, it cannot be turned off until ISYNC goes high. If you want to escape the condition, you must either DEACTIVATE, then re-ACTIVATE the emulator, or remove the cable from the ISYNC BNC connector. Removing the cable causes ISYNC to go high so that the emulator starts.

If you do not specify the *init-emu* condition and invoke GO while the emulator is halted, the emulator assumes that the condition is FROM $ (the address of the current execution pointer). If you do not specify the *init-emu* condition and invoke GO while the emulator is running, the Activity Monitor is reset according to its initial condition, but emulation is not affected.

4.4.4.2 Initial Activity Monitor Conditions

The Activity Monitor operates separately from emulation. The *init-AM* condition can be TRACE or NOTRACE, combined with OSYNCL or OSYNCH.

If you specify TRACE, the Activity Monitor starts storing data immediately on invocation of GO. It then continues storing data until stopped by *AM-programming*, the STOP command, or the HALT command.

If you specify NOTRACE, the Activity Monitor does not store data even though a GO command has been invoked and emulation has started. It does not store data until it encounters a TRACE action from a new invocation of GO or from a TRACE action with *AM-programming*.

The OSYNC parameter can be combined with TRACE or NOTRACE, or used by itself. This parameter sets the value of the OSYNC output line to OSYNCH (high) or OSYNCL (low).

If you do not specify the *init-AM* condition, the Activity Monitor defaults to the conditions specified in the last invocation of GO. Note that the default OSYNC level is derived either from *init-AM* or from the OSYNC tool variable, depending on which was entered last.

4.4.4.3 Activity Monitor Programming

AM-programming establishes how the Activity Monitor operates during the emulation session. The condition can be FOREVER or TIL *user-state-0* [THEN *user-state-1*].

If you use the FOREVER parameter, the Activity Monitor proceeds as specified by *init-AM*. No events are detected and no actions are taken.

The TIL *user-state-0* [THEN *user-state-1*] condition allows you to specify word recognition and actions. Each *user-state* allows two word recognizers and two actions, diagrammed as follows:

```
event [action] [ORIF event [action] ]
```

The *events* can be defined as execution addresses or bus events (address/data/status combinations). They can also be tied to the status of FULLBUF or a change in the ISYNC input signal.

The *action* parameter determines the action taken when an *event* is recognized. There are three basic *action* types:

- Activity Monitor and emulation control -- TRACE, NOTRACE, STOP, HALT, RESTART, REPEAT, and ORIF.
- OSYNC control -- OSYNCH, OSYNCL, and OSYNCT.
- Data retrieval -- MEM, REGS, and PCB.

The data retrieval *actions* are available only if the FASTBREAK tool variable is set TRUE (non-zero).

Each *user-state* has an occurrence counter associated with it that can be tied to one or both of the *events*. The counter counts the occurrences of the specified *event* and delays the *action* until the count is satisfied.

4.4.5 Using the STOP Command

You can use the STOP command to stop the Activity Monitor, without halting emulation. The command disables acquisition and word recognition, but leaves the emulator in Run mode. You can view the contents of the trace buffer while the Activity Monitor is stopped, and emulation continues uninterrupted.

4.4.6 Using the HALT Command

You can use the HALT command to asynchronously halt emulation and stop the Activity Monitor. The command puts the emulator in Halt mode.

When the emulator is in Halt mode, it maintains the following processor functions:

- DMA transfers
- Refresh cycles
- Bus HOLD/HLDA

The extent of interrupt servicing during Halt mode is established by the setting of the ENI tool variable. The available settings include:

- ENI = 0 -- The emulator does not service any interrupts in Halt mode. Interrupt requests are marked as pending and are serviced once the emulator is re-started. The /RES signal from the prototype is ignored.

- ENI = 1 -- The emulator services internal processor interrupts in Halt mode, except during mapped I/O requests and synchronous emulator start-ups. User interrupts are marked as pending and are serviced once the emulator is re-started. The /RES signal from the prototype is ignored.
- ENI = 2 -- The emulator services both user and internal processor interrupts in Halt mode, except during mapped I/O requests and synchronous emulator start-ups.

The emulator always services interrupts during Run mode.

When you set ENI to 1 or 2 for the first time, you must designate the starting address of a contiguous 2-byte block of RAM reserved for interrupt handling. You can change the ENI interrupt level only when the emulator is in Halt mode. Refer to the ENI entry in Chapter 6 of this manual for more information.

4.4.7 Command-Line Prompts During Emulation

During emulation, the command-line prompts indicate the current mode of the emulator. Table 4-2 lists and defines the various prompts. For more detailed information, refer to the Prompts entry in Chapter 6.

Note that the prompts are appended with a "+" symbol when the Activity Monitor is active.

Table 4-2 Emulator Modes

Mode	Prompt*	Description
Halt	hlt>	Emulation is halted and the Activity Monitor is stopped. Interrupt servicing is disabled with ENI=0.
Halt	idi>	Emulation is halted and the Activity Monitor is stopped. Internal processor interrupts are enabled with ENI=1, or internal processor interrupts and user interrupts are enabled with ENI=2.

* A "+" is appended to the prompt when the Activity Monitor is active.

Table 4-2 Emulator Modes (continued)

Mode	Prompt*	Description
Step	?	The emulator is stepping through code (using either LSTEP or STEP). Step mode must be entered from and will return to Halt mode.
Run	emu>	Emulation is active; FASTBREAK=FALSE.
Run	fst>	Emulation is active; FASTBREAK=TRUE.
Pause	IO?>	Emulation is paused because a mapped I/O request is pending. Use the RELEASEIO command to redisplay the pending request.
Pause	isync?>	Emulation is paused while the emulator waits for the ISYNC line to go high.
Pause	reset?>	Emulation is paused because the processor is being held reset.
Pause	busact?>	Emulation is paused because a BUSACT time-out has occurred.
Pause	memrdy?>	Emulation is paused because a MEMRDY time-out has occurred.
Pause	iordy?>	Emulation is paused because an IORDY time-out has occurred.
Arrested	pwr?>	The prototype power is off. Turn on power to the prototype, then proceed.
Arrested	clk?>	The emulator cannot detect a prototype clock. Fix the prototype clock, then proceed.

* A "+" is appended to the prompt when the Activity Monitor is active.

Table 4-2 Emulator Modes (continued)

Mode	Prompt*	Description
Arrested	hwfail?>	Emulator hardware failure. Contact your Intel service representative, using the service information provided on the inside-back cover of this manual.
Arrested	???>	The emulator is locked up due to a serious, unrecoverable problem. DEACTIVATE, then re-ACTIVATE the emulator.

* A "+" is appended to the prompt when the Activity Monitor is active.

In Arrested or Pause mode, you can enter any commands that do not require access to the emulator hardware or the processor.

4.5 Displaying the Trace Buffer

The trace buffer is controlled by the Activity Monitor programming you enter in the GO command. If the Activity Monitor is set to trace FOREVER, the trace buffer captures acquisitions continuously, overwriting old acquisitions with new. If the Activity Monitor stops or performs some action that stops the trace, the trace buffer discontinues acquisition.

The trace buffer hardware is 4096 acquisitions deep. More than one acquisition is required to form a CYCLES-format frame. A frame displays either an execution or a bus event. The number of frames available is a function of the following:

- Current microprocessor characteristics (i.e., the number of wait states per bus cycle, whether the task is compute-bound or bus-bound, etc.). Less frames are available if wait states are used.
- The Activity Monitor programming in the GO command. Turning TRACE on and off frequently reduces the number of available frames.

The maximum possible number of CYCLES-format frames is 3072. You can view the frames within the trace buffer by using the PRINT command.

4.5.1 Using the PRINT Command

You can use the PRINT command to view the contents of the trace buffer only when the Activity Monitor is stopped. The PRINT command syntax allows you to display all or a portion of the trace buffer, in either an INSTRUCTIONS or a CYCLES format.

The following paragraphs briefly describe the PRINT formats. For a detailed discussion, refer to the PRINT command entry in Chapter 6.

For an example of the two PRINT command formats, suppose you were emulating the following program block:

```
1000:0001H      57          PUSH    DI
1000:0002H      55          PUSH    BP
1000:0003H      89e5        MOV     BP,SP
1000:0005H      b81c00      MOV     AX,001cH
1000:0008H      50          PUSH    AX
1000:0009H      ff361a00    PUSH    WORD PTR 001aH
1000:000dH      e80f00      CALL    $+0012H ;A=001fH NEAR
```

Assume the GO command sequence is "GO FROM 1000:1H TRACE TIL EXECUTION 1000:0DH HALT".

The INSTRUCTIONS display of the program would appear as shown in Figure 4-2. This format shows executed instructions as Address, Data (opcodes), and Mnemonics. Bus cycle data (reads, writes, etc.), as well as any memory or register data stored as the result of a MEM or register action in the GO command, is interleaved between the instructions.

```
hlt> PRINT INSTRUCTIONS ALL
FRAME--ADDRESS---------DATA-----------MNEMONICS------------------------
   2   10001HP           57             PUSH   DI
   5   0fffeHP-W-0000
   6   10002HP           55             PUSH   BP
   8   0fffcHP-W-0000
   9   10003HP           89e5           MOV    BP,SP
  10   10005HP           b81c00         MOV    AX,001cH
  12   10008HP           50             PUSH   AX
  15   0fffaHP-W-1c00
  16   10009HP           ff361a00       PUSH   WORD PTR 001aH
  17   0001aHP-R-1a00
  19   0fff8HP-W-1a00
  20   1000dHP           e80f00         CALL   $+0012H  ;A=01001fH NEAR
  23   0fff6HP-W-1000
```

Figure 4-2 INSTRUCTIONS Display

The CYCLES display format of the program would appear as shown in Figure 4-3. This display shows trace information in time-aligned bus cycle or execution address format. The format is execution address, bus address, data, processor status, user-state status, and time-stamp information. The user-state status (USTATE) refers to the Activity Monitor programming of *user-state-0* and *user-state-1* in the GO command.

```
hlt> PRINT CYCLES OLDEST 7
FRAME -- EXEC ADDR -- BUS ADDR - DATA ------ STATUS -- USTATE ---- CLOCKS
    1                 10001HP      57        14 F          0           4
    2     10001HP
    3                 10002HP    5589        14 F          0           4
    4                 10004HP    e5b8        14 F          0           4
    5                 0fffeHP    0000        16 W          0           4
    6     10002HP
    7                 10006HP    1c00        14 F          0           4
    8                 0fffcHP    0000        16 W          0           5
    9     10003HP
   10     10005HP
   11                 10008HP    50ff        14 F          0           5
   12     10008HP
   13                 1000aHP    361a        14 F          0           4
   14                 1000cHP    00e8        14 F          0           4
   15                 0fffaHP    1c00        16 W          0           4
   16     10009HP
   17                 0001aHP    1a00        15 R          0           7
   18                 1000eHP    0f00        14 F          0           4
   19                 0fff8HP    1a00        16 W          0           5
   20     1000dHP
   21                 10010HP    191f        14 F          0           4
   22                 1001fHP      1f        14 F          0           6
   23                 0fff6HP    1000        16 W          0           4
```

Figure 4-3 CYCLES Display

The time-stamp information shown in the display depends on the setting of the CLK command. If CLK is 0, the display shows the number of clock ticks (up to 257 maximum) that elapsed between bus cycles. If CLK is non-zero, the display shows the time (in nanoseconds) between bus cycles. For more information regarding the CLK command, refer to Chapter 6.

4.6 Using the Stepping Commands

The emulator stepping commands allow you to execute your program without having to decide beforehand on a method of stopping emulation. There are four stepping commands, as listed below. You can use any of these commands to execute your program a step at a time, but the definition of "step" is different for each.

The following definitions apply to the stepping commands:

ISTEP	This command allows you to step through your program by machine-language instructions. You can specify the number of instructions to be stepped, as well as the starting address. ISTEP steps through CALLs to subroutines.
STEP	This command is a variation of the ISTEP command that allows you to continue stepping (by pressing the <Enter> key) or return to Halt mode (by pressing the E key). STEP allows you to step over, rather than through, CALLs to subroutines.
PSTEP	This command allows you to step through your program by high-level language statements. You can specify the number of statements to be stepped, as well as the starting address. PSTEP steps over separate procedures that are initiated by CALL instructions.
LSTEP	This command is a variation of the PSTEP command that allows you to continue stepping (by pressing the <Enter> key) or return to Halt mode (by pressing the E key). LSTEP steps through procedures that are initiated by CALL instructions.

You can use the stepping commands only when the emulator is in Halt mode.

The full syntax and parameters for the stepping commands are provided in Chapter 6.

4.7 Using Symbolics

The emulator software allows you to display or modify portions of your program or areas of program memory in a variety of ways. Program areas and variables can be accessed as address locations (physical or segmented), line numbers, or symbolic names. One of the more powerful features of the emulator is its symbolic debugging support. While the location of a named procedure or the absolute address of a symbolic variable may change, you will always be able to find it by using its symbolic name.

Refer to the Symbolic References entry in Chapter 6 for detailed information regarding the use of symbolics.

4.7.1 Loading Program Symbols

A program being debugged with the emulator must be compiled with a compiler that generates load files that are in Intel 8086 absolute Object Module Format (OMF). Also, to utilize the full symbolics capabilities of the emulator, modules must be compiled without optimization and with the compiler switches for debug information and symbols on.

When compiling a program with a supported Intel compiler, symbolic information is part of the object code. The program symbolic information includes the address and type definition information for each of the defined symbols in the program.

When the LOAD command is executed, the code and symbol information from the specified OMF file is loaded into mapped memory. The LOAD command has options that allow you to specify that the program be loaded without symbols (NOSYMBOLS) or without code (NOCODE). One reason for using the LOAD command with the NOSYMBOLS parameter would be if you did not wish to use symbolic references while debugging.

A symbol is a variable name that evaluates to an address. The memory contents (based on the symbol type definition) at the symbol address are the value of that symbol. The emulator software treats symbols the same as any other variables or addresses.

You do not have to define all of the symbols in your program; you may define them during a debug session. For example, it may be useful to define the start of executable program code with a symbolic name so you can easily return to that point in the program:

```
hlt> DEFINE POINTER BEGIN=1100:4H
hlt> GO FOREVER
  Use S (status) command to determine emulation status.
emu+> HALT
  Emulator Status:  halted (not servicing interrupts).
   Execpointer is :MESSG.DELAY_A_LITTLE_BIT_#129 + eH (1000:005cH)
  Activity Monitor Status:  Stopped.
hlt> GO FROM BEGIN TIL FULLBUF
  Emulator Status:  halted (not servicing interrupts).
   Execpointer is :MESSG.DELAY_A_LITTLE_BIT_#163 + 3H (1000:0051H)
  Activity Monitor Status:  Stopped.
hlt>
```

The PROC facility in emulator software also gives you the opportunity to create and use debug symbols. With PROCs, the symbols you define may be emulation aids or may end up being part of your final program. Once symbol definitions are created, you can save them for future use by using the PUT and APPEND commands. Refer to Chapter 6 for more detailed information on the LOAD, PUT, APPEND, and PROC commands.

4.7.2 Using Symbols During Emulation

The symbols defined in your program, as well as the symbols defined during a debug session, are all available for use during emulation. You can use symbolic names as part of the word-recognition programming in the GO command (e.g., GO FROM :messg.main_#222) or as part of any address partition in any command (e.g., ASM pstart_ LENGTH 3, ORD1 &ARRAY[3] LENGTH 5).

You can also view and change the contents of data structures using C-like constructs. For example:

```
hlt> main_
1000:0000H

hlt> display_buffer_[3]
45H

hlt> *messgptr = 47
hlt> *messgptr
47H
```

4.8 Memory and Register Access

This section describes physical and segmented addressing and shows you how to access memory and register contents.

4.8.1 Address-Pointer Grammar

The emulator software can access addresses in program memory as physical, segmented, or symbolic addresses.

4.8.1.1 Physical Addresses

Physical addresses are 20 bits long. You can explicitly refer to a physical address by appending a "P" to the address as shown in the following example:

```
12345HP          (physical address 12345H)
```

As implied by its name, a physical address directly corresponds to a physical location in hardware. Using physical addressing, the 80C186 microprocessor can directly address 1M byte of memory.

4.8.1.2 Segmented Addresses

If an address lacks the suffix "P" (indicating a physical address), the emulator assumes that the expression is a segmented address.

Segmented addresses are specified as *segment:offset* pairs. The *segment* is a 16-bit base address contained in a segment register, and the *offset* is a 16-bit value provided by an instruction in the program. When the 80C186 microprocessor performs address translation, it shifts the 16-bit *segment* left by four bits and then adds the 16-bit *offset*. The result is a 20-bit physical address. If *segment* is not specified, the default *segment* is the code segment.

4.8.1.3 Symbolic Addresses

As a symbolic debugger, the emulator software allows you to display or modify a program variable by using its symbolic name. Naturally, this form of addressing cannot be used if you specified NOSYMBOLS when loading your program. As an example of symbolic addressing, assume that "quarters" is a program variable of type INT1. Set "quarters" equal to 1 and then display it as follows:

```
hlt> quarters = 1
hlt> quarters
01H
```

4.8.2 Memory Access

The memory access commands allow you to read, write, or copy specific memory locations. Memory is specified by using data types and addresses to indicate an individual location or a partition. A memory read command displays the contents of a memory location as shown in the following example:

```
hlt> ORD4 11H
0011H     14a147e7
```

Contiguous memory is displayed by using the partition option, either "*address* TO *address*" or "*address* LENGTH *number-of-items*", as in the following example:

```
hlt> ORD2 $ LENGTH 16T
0021:0010H   46e7  14a1  e700  a148  0016  4ae7  1251  e700
0021:0020H   8842  5de5  8bc3  4dec  554d  0a9a  5900  9a00
```

Memory write commands are executed by assigning values to memory locations, and memory copy commands are executed by setting one memory location equal to another memory location.

The following example executes two memory writes and then a memory copy:

```
hlt> ORD2 40:04H=1234,0ABCD,3
hlt> NSTRING 87D2:67F4H="INTEL"
hlt> ORD1 0HP LENGTH 128T=ORD1 100HP
```

Note that in memory copy statements, the data type on either side of the "=" symbol must be the same (i.e., ORD1 = ORD1).

4.8.3 Register Access

The emulator software gives you the ability to display or modify the contents of processor registers during an emulation session.

Your register access rights during emulation are governed by the state of the FASTBREAK tool variable. If FASTBREAK is set to 0 (FALSE), access to registers is not allowed during emulation. If set to 1 (TRUE), access to registers is allowed during emulation.

You can display or modify the contents of a register by typing the keyword for the register. For example, the following command sequence first displays, then modifies the contents of the accumulator register pair:

```
hlt> AX
015AH
hlt> AX=0F15AH
hlt> AX
F15AH
```

Refer to the Register Access entry in Chapter 6 of this manual for a list of 80C186 microprocessor register keywords.

Two specific commands (in addition to the individual register keywords) display various 80C186 microprocessor register values. These commands are as follows:

REGS	Displays the current values of the processor general-purpose, flag, and segment registers.
PCB	Displays the contents of the Peripheral Control Block.

4.8.4 Coprocessor Support

When COPROC187 is set to TRUE (non-zero), the emulator provides emulation and debugging support for an 80C187 numeric coprocessor. You can display and modify 80C187 coprocessor registers using several emulator commands.

NOTE

To display the 80C187 stack registers, you must have an 80287 coprocessor installed on your IBM PC AT system board.

The R187 command displays the current contents of the 80C187 registers.

The three registers (FIA, FDA, and FIO) hold address and instruction information. The remaining eleven registers (ST0-7, FSW, FCW, and FTW) hold stack, status, control, and tag information on the 80C187 coprocessor.

By entering the appropriate register keyword, you can display or change the 80C187 registers. Enter the register keyword to display the current contents of a particular register. In order to modify a register value, enter the register keyword followed by an equals sign (=) and the desired new value. Make certain that any new values you enter for 80C187 registers obey the data type requirement for that register. Refer to the Register Access 80C187 entry in Chapter 6 for information on register keywords and register data type requirements.

Contents
Chapter 5 Using External Equipment

Chapter 5
Using External Equipment

5.1 Introduction

You can use the emulator with an Intel iPAT™ Analyst, a general-purpose logic analyzer, or other Intel emulators.

Connectors to these external devices are provided on the emulator controller unit (ECU), as shown in Figure 5-1. The connectors are:

- **iPAT Connector** -- a 60-pin shrouded connector that serves as an interface to the Intel performance analysis tool.
- **Logic Analyzer Connector** -- a 60-pin edge connector that serves as an interface to a general-purpose logic analyzer.
- **ISYNC and OSYNC** -- two BNC connectors that are used for synchronous operation with other Intel emulators.

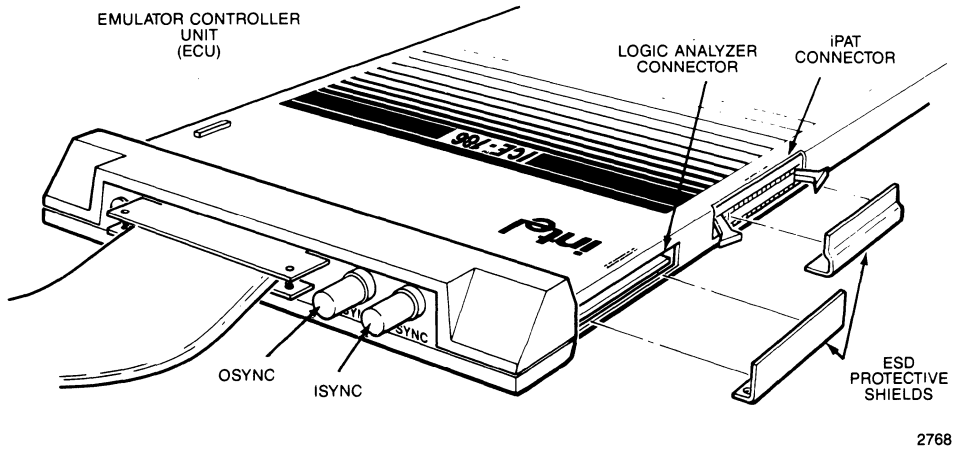


Figure 5-1 Connectors to External Equipment

This section describes the connectors to the external equipment and discusses any operating considerations. Note that all external equipment is user-supplied.

5.2 Logic Analyzer Connector

Table 5-1 lists the pinouts of the emulator logic analyzer interface connector. As you view the logic analyzer connector in its opening in the ECU housing, pin 1 is the lower left pin, pin 2 is the upper left pin, pin 59 is the lower right pin, and pin 60 is the upper right pin.

Table 5-1 Logic Analyzer Interface Connector Listing

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
1	Ground	2	CLKOUT	3	CLK2	4	CLK1
5	N.C.	6	RESET	7	N.C.	8	Ground
9	N.C.	10	N.C.	11	N.C.	12	Ground
13	N.C.	14	N.C.	15	FF	16	Ground
17	AD0	18	A16	19	AD1	20	Ground
21	AD2	22	A17	23	AD3	24	Ground
25	AD4	26	A18	27	AD5	28	Ground
29	AD6	30	A19	31	AD7	32	Ground
33	AD8	34	N.C.	35	AD9	36	Ground
37	AD10	38	N.C.	39	AD11	40	Ground
41	AD12	42	N.C.	43	AD13	44	Ground
45	AD14	46	N.C.	47	AD15	48	Ground
49	/S0	50	/LOCK	51	/S1	52	Ground
53	/S2	54	ISYNC	55	DMA	56	Ground
57	/BHE	58	OSYNC	59	N.C.	60	Ground

Note: Signals preceded by a slash (/) are active low.

CAUTION

Pins of the logic analyzer interface connector that are listed as "N.C." in Table 5-1 must be left unconnected. Failure to do so may result in equipment damage.

The emulator logic analyzer signals are derived from the microprocessor signals. The logic analyzer signals are defined as follows:

RESET	RESET output
FF	Indicates that this bus cycle is the first fetch after a queue flush
AD0-AD15	Multiplexed address/data lines
A16-A19	High order address lines
/S0, /S1, /S2	Bus status lines
/BHE	Bus High Enable line
/LOCK	Bus Lock line
DMA	Indicates that this bus cycle is a DMA transfer

Use the falling edge of CLK1 to sample A0-A15. Use the falling edge of CLK2 to sample D0-D15 and the rest of the signals. The setup time to CLK1 or CLK2 is 60 ns minimum; the hold time is 20 ns minimum.

Note that the logic analyzer interface also has ISYNC and OSYNC signals. These signals operate as described in Section 5.3.

5.3 ISYNC and OSYNC Connectors

With the ISYNC (input SYNC) and OSYNC (output SYNC) connectors, you can synchronize the operation of the emulator and other Intel emulators. The emulator supports a synchronization protocol that uses the two BNC connectors for communication with external emulators.

The OSYNC signal is a tool variable recognized within the emulator command language. You can read or write the OSYNC variable in the same manner as any other emulator tool variable.

ISYNC behaves the same as a "TRIG IN" signal and OSYNC behaves the same as a "TRIG OUT" signal. You can use the OSYNC variable to specify the "TRIG OUT" level to be output with the next invocation of the GO command, for example:

```
hlt> OSYNC=1    /* set the next output to TTL high */
hlt> OSYNC      /* display the current level at the BNC */
Next GO command will use:  OSYNCH
Current OSYNC level at BNC:  0H
hlt> GO         /* enter GO and the output level changes */
  Use S (status) command to determine emulation status.
emu+> HALT      /* halt emulation */
  Emulator Status:  halted (not servicing interrupts).
   Execpointer is :SAMPLE.DELAY_A_LITTLE_BIT_#129 + eH (1000:005cH)
  Activity Monitor Status:  Stopped.
hlt> OSYNC      /* display the current level at the BNC */
Next GO command will use:  OSYNCH
Current OSYNC level at BNC:  1H
hlt>
```

The ISYNC and OSYNC variables can also be used as specifications in the GO command. The following example shows the use of the ISYNC and OSYNC variables in the GO command.

This command starts emulation when the ISYNC signal goes high:

```
hlt> GO WHEN ISYNCH
```

This command causes the OSYNC signal to go high at the start of emulation and toggle (go low) at the end of emulation.

```
hlt> GO FROM :program.start_ OSYNCH TIL EXECUTION \
hlt>> :program.end_ OSYNCT
```

Both the ISYNC and the OSYNC signals are open collector, TTL-level signals. The OSYNC signal is an output only; checking its current value returns the last setting, not the current value of the signal line. A maximum of eight emulators (open collector loads) can be connected to the emulator OSYNC signal line at one time.

The ISYNC signal is normally used as a "TRIG IN" input. It may, however, be connected in a "daisy chain" configuration with other emulators, as shown Figure 5-2. A maximum of eight emulators may be connected to the emulator ISYNC signal line at one time.

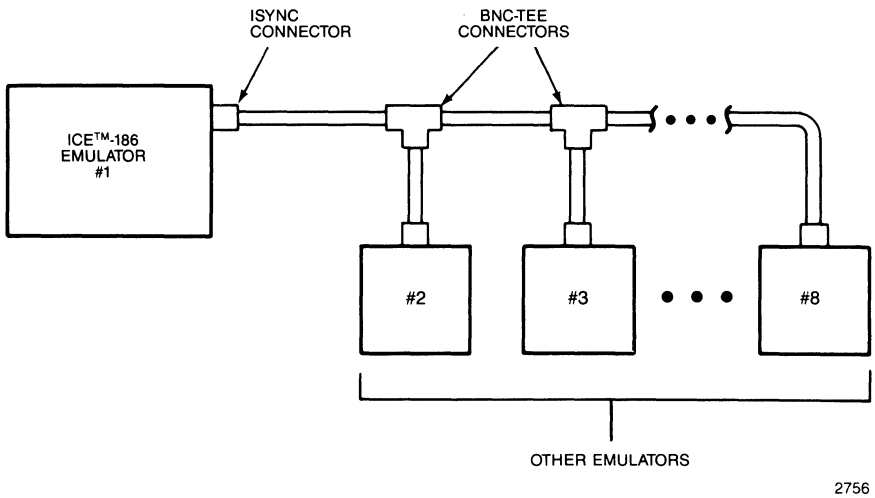


Figure 5-2 Interconnecting an ICE™-186/188 Emulator with Other Emulators

In this "daisy chain" configuration, the ISYNC signal remains low (0) until all emulators on the line attempt (via internal circuitry) to raise the signal level high (1) upon receiving a GO command. When all emulators on the line have received a GO command, the ISYNC signal goes high (1), and all emulators start synchronously.

All of the emulators are held in an extended "wait state" until the last emulator receives its GO command. If you want to release the emulator from waiting for ISYNC to go high (1), remove the cable from the ISYNC BNC connector. Removing the cable causes ISYNC to go high (1).

Note that if the ISYNC/OSYNC lines of the logic analyzer interface are connected to an open collector or other low impedance load, they must each be counted as one of the allowed loads on the BNC ISYNC/OSYNC signal lines.

Contents
Chapter 6 Command Encyclopedia

Chapter 6
Command Encyclopedia

6.1 Introduction

This chapter provides an alphabetical listing of emulator commands and topics. Chapter 6 provides a complete reference for the emulator command language.

6.2 How to Use This Chapter

The following is an example of an entry in this encyclopedia. The entry shows what commands to use and how to enter them. It also provides a discussion of the command, examples, and cross-referenced related subjects.

All entries are referenced alphabetically. Use the syntax section as a reference in creating syntactically correct commands. Use the discussion section to determine the use of the command. Use the examples for further clarification and understanding of the command, and use the Cross-Reference section to locate related entries in this encyclopedia.

COMMAND NAME

Short statement of the
command's function

Syntax

The command syntax shows how to construct a legal emulator command. (Syntax notation is explained on the inside front cover and on the back side of the tabbed divider for this chapter.)

Discussion

The discussion section describes how the command is used, including why and when the command is most useful.

Example

Examples are preceded by explanations of the example. Examples may also have comments enclosed in slash-asterisk delimiters (e.g., /* comment */). User input is preceded by a prompt (for example, hlt>) and underlined to distinguish it from system output. All emulator keywords are capitalized. For example:

The following is an example of user input (underlined).

```
hlt> BASE                                    /* emulator command */
16T                                          /* emulator response */
```

Not all the commands have examples.

Cross-Reference

Cross-reference items are commands and topics related to the current command entry. Not all of the commands are cross-referenced.

Table 6-1 groups the ICE-186/188 commands and topics by function.

Table 6-1 ICE™-186/188 Command Categories

Utility Commands	
Entry	**Description**
@	Escapes to the host operating system.
APPEND	Stores emulator data base information to a file.
BASE	Displays or modifies the number base.
BREAK	Exits from a control block.
CONTINUE	Transfers control from within a control block to the end of the block.
COUNT	Groups and executes commands a specified number of times or until a specified condition occurs.
DEFINE	Creates new debug objects.
DIR	Lists the names of debug, tool, or code objects.
DISPLAYFLAG	Controls object display in assignment statements.
DO ... WHILE	Groups and conditionally executes commands.
DO_	Groups and executes commands.
EDIT	Invokes a disk-resident editor.
EDITOR	Specifies which editor is invoked by EDIT.
ERROR	Controls the display of error information.

Table 6-1 ICE™-186/188 Command Categories (continued)

Utility Commands (continued)	
Entry	**Description**
EVAL	Evaluates an address or an expression to its module, procedure, line, or symbol reference.
EXIT	Exits the emulator software.
FOR	Groups and executes commands in a loop.
HELP / Help Mode	Provides on-line information on keywords, topics, and error messages.
IF ... ELSE	Groups and conditionally executes commands.
INCLUDE	Executes emulator commands defined in a disk file.
LIST	Records a debug session to a file.
NOLIST	Closes a LIST file.
PROC	User-defined procedure.
PUT	Stores emulator data base information to a file.
REDEFINE	Creates or recreates a debug object.
REMOVE	Deletes debug objects.
REPEAT	Groups and executes commands forever or until an exit condition is met.
RESET	Re-initializes specified functions of the emulator.
SAVE	Saves the contents of memory as an object file on disk.
SELFTEST	Runs the customer confidence tests.

Table 6-1 ICE™-186/188 Command Categories (continued)

Utility Commands (continued)	
Entry	**Description**
SHOW	Displays the definitions of debug objects.
SWITCH	Causes execution to branch to one of several CASE statements or the DEFAULT statement.
VERSION	Displays the version numbers of the emulator software.
WHILE	Executes a loop while a condition is TRUE.

Memory Management Commands	
Entry	**Description**
MAP	Determines whether memory accesses are mapped to USER or ICE RAM.
Memory Access	Displays or writes to a location in memory.
PHYSICAL	Converts a segmented address to a physical address.
POINTER <pointer-type>	Determines which pointer data-type is used to access memory.
VERIFY	Enables/disables verification of memory write operations.

Table 6-1 ICE™-186/188 Command Categories (continued)

User Program Commands	
Entry	**Description**
$	Execution address.
ASM / SASM	Displays or modifies memory in ASM86 assembly instructions.
CALLSTACK	Displays the names of procedures on the stack.
LOAD	Loads an object file into mapped memory.
NAMEFRAME	Displays stack frame.
NAMEPATH	Determines the search order for emulator objects.
NAMESCOPE	Shortens symbol referencing.
SAVE	Saves the contents of memory as an object file on disk.
SYMBOLIC	Enables/disables symbolic display.
Symbolic References	References to program addresses and variables.
UPDATE	Updates NAMESCOPE when $ is updated.

Table 6-1 ICE™-186/188 Command Categories (continued)

Emulation/Trace Commands	
Entry	**Description**
CAUSE	Displays the reason emulation stopped.
CLK	Determines the format of the time-stamp information in the PRINT CYCLES display.
FASTBREAK	Interupt and resume emulation to collect specific system data.
GO	Starts emulation and controls break and trace functions (the operations of the Activity Monitor).
HALT	Halts the emulator and stops the Activity Monitor.
IPATINT	Selects the interrupt source for iPAT interrupt latency measurements.
ISTEP	Steps through a program by machine-language instructions.
LSTEP	Steps through a program by high-level language statements.
PRINT	Formats and displays the contents of the trace buffer.
PSTEP	Steps through high-level language statements, executing procedures as one statement.

Table 6-1 ICE™-186/188 Command Categories (continued)

Emulation Trace Commands (continued)	
Entry	**Description**
QUERY	Determines whether commands can be entered while emulator is in Run mode.
STATUS / S	Displays the current emulator status.
STEP	Steps through a program by machine-language instructions.
STOP	Stops the Activity Monitor without stopping emulation.

I/O Control Commands	
Entry	**Description**
HOLDIO	Suspends I/O requests to ICE-mapped ports.
IORDY	Enables an emulator time-out when an I/O access takes more than one second.
MAPIO	Determines whether I/O ports are mapped to USER or ICE.
PORT	Displays or modifies the contents of byte-wide USER-mapped I/O ports.
RELEASEIO	Allows pending I/O requests to be serviced.
WPORT	Displays or modifies the contents of word-wide (16-bit) I/O ports.

Table 6-1 ICE™-186/188 Command Categories (continued)

Processor Control Commands	
Entry	**Description**
ALEMODE	Specifies whether the processor ALE signal goes active high at the beginning or end of each bus cycle.
BTHRDY	Allows emulator memory to simulate EPROM (or slower) memory via the READY signal.
BUSACT	Enables an emulator time-out when the emulator processor bus is inactive for approximately one second.
COPROC187	Notifies the emulator of the presence of a 80C187 numeric processor extension unit.
ENI	Controls the status of interrupt servicing during Halt mode.
Flags	Displays or modifies the microprocessor flags register.
MEMRDY	Enables an emulator time-out based on memory access time.
OSYNC	Sets the synchronous line output.
PCB	Displays the register contents of the Peripheral Control Block.

Table 6-1 ICE™-186/188 Command Categories (continued)

Processor Control Commands (continued)	
Entry	**Description**
R187	Displays the 80C187 numeric processor extension unit register contents.
QSTAT	Enables/disables emulator 80C187 numeric processor extension unit support.
Register Access	Displays or modifies the contents of microprocessor registers.
REGS	Displays the 80C186 microprocessor register and flag contents.
RSTEN	Enables/disables external reset of the processor during emulation.
TOOLVAR	Displays the settings of the tool variables.

Invocation Commands	
Entry	**Description**
ACTIVATE	Establishes communication between the host and the emulator.
DEACTIVATE	Deactivates the emulator.
Invocation	Procedures for invoking the emulator software and activating the emulator hardware.

Table 6-1 ICE™-186/188 Command Categories (continued)

Topical Entries	
Entry	**Description**
Address Translation	Describes segmented and physical address translation.
Character Functions	Built-in functions for character classification and transformation.
<data type>	Basic program data types used in commands and displays.
<expr>	One or more numbers, variables, or functions separated by operators.
Functions	Built-in functions or user-defined procedures.
I/O Functions	Built-in functions for input and output.
Math Functions	Built-in mathematical functions.
Miscellaneous Functions	Built-in miscellaneous functions.
Object Hierarchy	Illustrates software organization.
Parameters	Define and control the state of the emulator software.
Partition	An address or range of addresses.
Prompts	The command-line prompts that indicate the emulator mode.
String Functions	Built-in functions for string manipulation.
Register Access 80C187	Displays and modifies 80C187 numeric processor extension unit registers.

$
Execution address

Syntax

```
$ [= addr-expr]
```

Where:

$	if entered without an option, displays the current execution pointer and the fully qualified symbolic equivalent.
addr-expr	specifies an expression which must evaluate to a segmented or symbolic address. *Addr-expr* specifies the address (pointer value) that the execution pointer is set to. (See Address Translation in this chapter for more information on address formats.)

Discussion

Use the $ parameter to display or modify the current execution pointer. Upon invocation, $ is set to a NULL value. If you load a program with code, $ is then automatically updated to the starting point of the program code, if the information is available in the Object Module Format (OMF) file.

The $ parameter is automatically updated when the emulator switches from Run mode to Halt mode, after code is loaded into memory, or after RESET REGS or RESET ICE is issued. The $ parameter is not updated when the CS or IP registers are directly modified.

If you modify $ using a segment address, the *segment* part of the address is optional. If you omit the *segment* part, the contents of the code segment (CS) register are assumed. The *offset* part of the address specifies the instruction pointer (IP).

When the UPDATE parameter is set to TRUE, NAMESCOPE is updated whenever $ is changed.

Examples

1. The following example shows how to display the current execution point.

```
hlt> $
1000:0000H  ":mainmod.proca.procx#200"
hlt>
```

2. The following example shows how to change the current execution point.

```
hlt> $ = 1000:0000H
hlt>
```

Cross-References

Address Translation
NAMEFRAME
NAMESCOPE
Parameters
UPDATE

@

Escapes to the host operating system

Syntax

@ *host-command* [@]

Where:

@	(the at-sign) signals the emulator software that the following command is a host operating system command.
host-command	specifies any valid host operating system command.

Discussion

The at-sign is a delimiter that is used to pass a host operating system command to the host operating system. If the host command is followed by <Enter>, the second at-sign is not required. If more than one host command is entered on the command line (separated by semi-colons), each host command must be delimited by a preceding and following at-sign. (See the first example.)

Text to be passed to the host operating system is expanded with current LITERALLY definitions. However, you can suppress literal substitution (LITERALLY expansion) by enclosing the alias in single quotes.

Redirection (< or >), appending data to a file (>>), or piping (|), do not need to be preceded by a grave (‘) if they are operations to be performed by the host operating system. However, if redirection, appending data to a file, or piping is to be performed by the emulator software, the appropriate symbol must be preceded by a grave (‘). The following examples explain how to use the at-sign in a variety of contexts.

NOTE

There are memory limitations when escaping to the host operating system. The escape function requires a minimum of 30K bytes of DOS-recognized free memory. If there is insufficient free memory, the specified function does not execute and the emulator prompt is returned. If there is insufficient memory, delete any extraneous operating programs, e.g., Sidekick®.* Enter the @ CHKDSK command to determine the amount of DOS-recognized free memory that is currently available.

Examples

1. The following example shows how to pass the CD and DIR commands to the host operating system. The display shows the current working directory.

```
hlt> @ CD C:\ASM\MOD\WORK @ ; @ DIR

Volume in drive C is  BASE_DIR
Directory of  C:\ASM\MOD\WORK

.            <DIR>      12-23-86   4:30p
..           <DIR>      12-23-86   4:31p
TEST1    DOC        512  12-23-86   4:33p
TEST2    DOC        512  12-23-86   4:34p
SAMPLE1  DOC        512  12-23-86   4:40p
BAKWORK  DOC        512  12-23-86   4:45p
       6 File(s)   12847936 bytes free
hlt>
```

*Sidekick is a registered trademark of Bordland, International.

2. The following example shows how to define work LITERALLY as w and pass the command DIR w to the host operating system.

```
hlt> DEFINE LITERALLY w = "work"
hlt> @ DIR w

Volume in drive C is  BASE_DIR
Directory of  C:\ASM\MOD\WORK

.            <DIR>      12-23-86   4:30p
..           <DIR>      12-23-86   4:31p
TEST1    DOC        512  12-23-86   4:33p
TEST2    DOC        512  12-23-86   4:34p
SAMPLE1  DOC        512  12-23-86   4:40p
BAKWORK  DOC        512  12-23-86   4:45p
       6 File(s)   12847936 bytes free
hlt>
```

3. The following example shows how to suppress literal substitution and pass the command DIR 'w' to the host operating system.

```
hlt> @ DIR 'w'

File not found
hlt>
```

4. The following example shows how to pass the command DIR to the host operating system and, transfer the result to SORT. The transfer is performed by the DOS operating system pipe function. (SORT is a DOS operating system command.)

```
hlt> @ DIR | SORT

       6 File(s)    12847936 bytes free
Directory of  C:\ASM\MOD\WORK
Volume in drive C is  BASE_DIR
.            <DIR>      12-23-86   4:30p
..           <DIR>      12-23-86   4:31p
BAKWORK  DOC        512  12-23-86   4:45p
SAMPLE1  DOC        512  12-23-86   4:40p
TEST1    DOC        512  12-23-86   4:33p
TEST2    DOC        512  12-23-86   4:34p
hlt>
```

5. The following example shows how to escape to the host operating system and execute system commands, then return to the emulator software.

```
hlt> @ COMMAND.com

Host identification sequence
C: Enter DOS commands
C: EXIT   /* return to the emulator software */
hlt>
```

ACTIVATE

Establishes communication between the host and the emulator

Syntax

```
ACTIVATE  [USING io-device]  ICE186  [RESTART]
```

Where:

ACTIVATE	if entered without optional parameters, it displays the currently active tool and I/O device specification.
USING	precedes the identification of an I/O device. Communication between the emulator and the IBM PC AT occurs over this I/O device.
io-device	identifies the I/O device as either COM1, COM2, or GPIB.
ICE186	identifies the emulator as the tool to be activated. The emulator is defined as "DEFINE TOOL ICE186=ICE186.CFG" in the ICE186.INC include file, which is called at invocation. (See the Invocation entry in this chapter.)
RESTART	attempts to establish communication without resetting the target environment.

Discussion

The emulator must be activated before its hardware can be accessed and used.

Initially, the emulator is activated by the invocation batch file, RUN186.BAT. This file calls the ICE186.INC file, which in turn executes the ACTIVATE ICE186 command. The ACTIVATE ICE186 command activates the emulator by performing the following functions:

- Resets the target environment.
- Establishes communication between the IBM PC AT and the emulator controller unit.

- Verifies that power-up diagnostics were passed.
- Downloads the emulator control unit and user probe software.

You need only re-enter the ACTIVATE ICE186 command if you encounter an I/O error during invocation. If you deactivate the emulator, but do not turn-off its power, then want to re-activate it with the current emulator environment, enter ACTIVATE ICE186 RESTART (see DEACTIVATE in this chapter).

The RESTART option forces the emulator to activate without resetting the target environment, without executing power-up diagnostics, and without downloading emulator control unit and user probe software.

If RESTART fails to activate the emulator, the host automatically resets the target environment, executes power-up diagnostics, and downloads software.

The default I/O device is determined by the IO_DEVICE keyword in the ICE186.CFG configuration file. You can use the optional USING parameter to override the default I/O device specification as:

COM1	I/O device specification indicates that communication between the emulator and the IBM PC AT occurs via the PC COM1 serial channel.
COM2	indicates that communication between the emulator and the IBM PC AT occurs via the PC COM2 serial channel.
GPIB	indicates that communication between the emulator and the IBM PC AT occurs via the GPIB-PC2A board and GPIB.COM software driver.

If you encounter an I/O error during invocation, perform the following steps:

1. Make sure that the emulator is powered on.
2. Test the interface cable between the IBM PC AT and the emulator to make sure it is connected and seated properly. Then enter the ACTIVATE ICE186 command.

3. If activation still fails, exit the emulator software and check the configuration files to make sure that the IO_DEVICE variables are correct for the I/O device you are using. Then re-invoke the emulator software. (Refer to the *ICE™-186/188 In-Circuit Emulator Installation Supplement* for more information.)
4. If activation still fails, exit and re-invoke the emulator software and check the power-up sequence of the LEDs on the emulator controller unit. The red (power) LED should turn on and remain on, while the green LED should turn on briefly and then turn off. If the green LED doesn't turn on, or if it turns on and remains on, the emulator has detected a hardware failure.

If activation continues to fail, contact your local Intel service representative as indicated on the inside-back cover of this manual.

Examples

1. This example activates the emulator using the default I/O device defined in the ICE186.CFG configuration file.

```
-> ACTIVATE ICE186
IO_DEVICE: COM1
BAUDRATE: 38400

...emulator control software is downloaded...
...version information displayed...

hlt>
```

2. This example activates the emulator and establishes I/O communication via PC COM2. The USING parameter overrides the I/O device defined in the ICE186.CFG configuration file.

```
-> ACTIVATE USING COM2 ICE186
IO_DEVICE: COM2
BAUDRATE: 38400

...emulator software is downloaded...
...version information displayed...

hlt>
```

3. In this example, the emulator is activated and MAP is set. The emulator is then deactivated and reactivated with the RESTART option. The MAP setting is retained.

```
-> ACTIVATE ICE186
IO_DEVICE: COM1
BAUDRATE: 38400

...emulator software is downloaded...
...version information displayed...

hlt> MAP OP LENGTH 128K ICE
00000HP length 20000H ICE READ/WRITE
20000HP length e0000H USER READ/WRITE
HLT> DEACTIVATE ICE186
->
-> ACTIVATE ICE186 RESTART
IO_DEVICE: COM1
BAUDRATE: 38400

...version information displayed...

hlt> MAP
00000HP length 20000H ICE READ/WRITE
20000HP length e0000H USER READ/WRITE
```

4. This example displays the currently active tool and I/O device specification.

```
hlt> ACTIVATE
ice186 using COM2 in TIP # 0
hlt>
```

When activation takes place, the command-line prompt changes from the command language prompt "->" to the emulator prompt "hlt>". The "hlt>" prompt means that the emulator is active and in Halt mode. (This prompt does not always appear when you use the RESTART option.)

Cross-References

DEACTIVATE
DEFINE
Invocation

Address Translation

Describes segmented and physical address translation

Discussion

The emulator supports both segmented and physical addressing. It allows addressing of up to 1M byte of physical memory. The last legal address is 0FFFFFHP.

The translation of segmented and physical addresses are described below.

Segmented Addresses

Segmented addressing is in the format of segment:offset pairs. Each pair represents a real address. The segment consists of the upper 16 bits of the 20-bit base address. The offset is a 16-bit value that represents the offset from the base address into the segment. If only the offset is specified, the default segment is the contents of the code segment register (CS).

When translating segmented addresses, the emulator shifts the 16-bit segment left by four bits and then adds the 16-bit offset. The result is a 20-bit physical address. For example:

```
1000:0234H    =>    10234HP
```

Note that the 80C186 microprocessor rolls over the offset part of a real address when the offset exceeds 0FFFFH. When the offset is 0FFFFH + 1, the microprocessor rolls the offset to 0000H. For example, if real address 1000:FFFDH is incremented by 1 four times, the resulting addresses are as follows:

```
1000:FFFEH    =>    1FFFEHP
1000:FFFFH    =>    1FFFFHP
1000:0000H    =>    10000HP
1000:0001H    =>    10001HP
```

The emulator does not allow real addresses that evaluate to physical addresses greater than 0FFFFFHP. For example, if real address 0FFF0:00FDH is incremented by 1 three times, the resulting addresses are as follows:

```
0FFF0:00FEH    =>    0FFFFEHP
0FFF0:00FFH    =>    0FFFFFHP
0FFF0:0100H    =>    100000HP    ;the emulator displays an error
```

Physical Addresses

Physical addresses are specified by appending a "P" to the address number. Otherwise, the emulator assumes that the address is a real address. Valid physical addresses are 20-bits wide, with 0FFFFFHP the last legal address.

Physical addresses are contiguous. A memory read operation with a physical address returns contiguous data, up to the 1M-byte physical memory boundary. Physical addresses are not rolled over. For example, if physical address 1FFFDHP is incremented by 1 four times, the resulting addresses are as follows:

```
1FFFEHP
1FFFFHP
20000HP
20001HP
```

The emulator does not allow physical addresses greater than 0FFFFFHP. For example, if physical address 0FFFFDHP is incremented by 1 three times, the resulting addresses are as follows:

```
0FFFFEHP
0FFFFFHP
100000HP                    ;the emulator displays an error message
```

Cross-References

<data type>
<expr>
Memory Access

ALEMODE

Specifies whether the processor ALE signal goes active high at the beginning or end of each bus cycle.

Syntax

```
ALEMODE [ = { START } ]
        [   { END   } ]
```

Where:

ALEMODE	if entered without parameters, it displays the current ALEMODE setting.
START	specifies that the ALE signal goes active high at the beginning of each bus cycle.
END	specifies that the ALE signal goes active high at the end of each bus cycle.

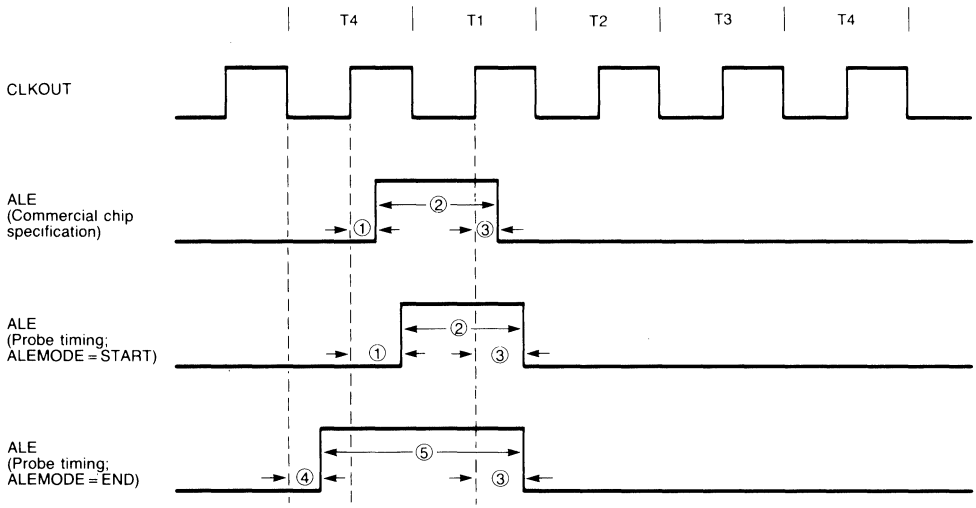
Default

Start

Discussion

The ALEMODE tool variable allows you to specify whether the processor ALE signal goes high at the START or the END of each bus cycle. Figure 6-1 illustrates the differences between these two settings.

The ALEMODE=START setting corresponds to how the processor normally outputs ALE. With this setting, the ALE signal goes high at the beginning of each bus cycle. The ALE transition is tied to the rising edge of the CLKOUT signal preceding T1.



NOTE

① T_{CHLH} (ALE Active Delay)
② T_{LHLL} ALE Width)
③ T_{CHLL}(ALE Inactive Delay)
④ CLKOUT low to ALE high for ALEMODE = END
⑤ Variable Width; ALE remains high during T_{IDLE}, T_{HLDA}, and ICE accesses.

For time specifications of T_{CHLH}, T_{LHLL} and T_{CHLL}, refer to the Release Note that was enclosed with this manual.

Figure 6-1 ALE Signal START/END Timing

The ALEMODE=END setting allows you to change the ALE timing so that the signal goes high at the end of each bus cycle. With this setting, the ALE transition is tied to the falling edge of the CLKOUT signal of the preceding T4 cycle. The ALE signal remains high during any TIDLE or THLDA bus cycles that occur before the next T1 period. It also remains high during any ICE breaks.

ALEMODE (continued)

Typically, you would use the ALEMODE=END setting if you needed to compensate for the TCHLH (ALE Active Delay) associated with the START setting. By causing ALE to go high at the end of a bus cycle, you minimize the ALE Active Delay for the next cycle.

Example

This example displays the current ALEMODE setting, then changes the variable to END.

```
hlt> ALEMODE
START
hlt> ALEMODE = END
hlt>
```

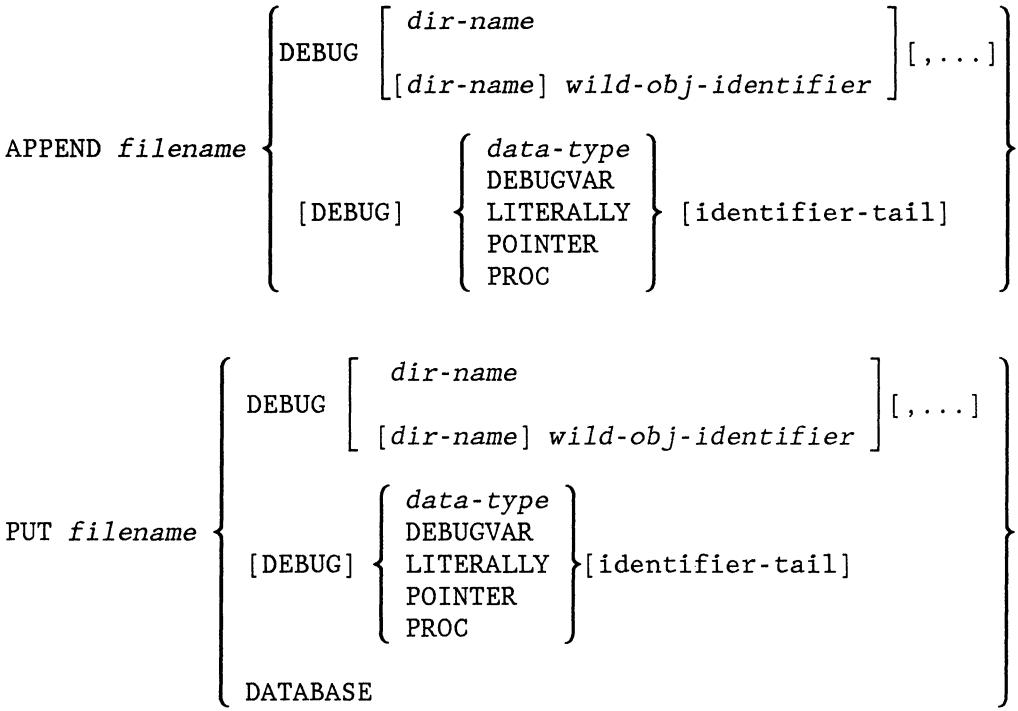
Cross-Reference

TOOLVAR

APPEND / PUT

Stores emulator data base information to a file

Syntax



identifier-tail::= [*dir-name*] [*wild-obj-identifier*] [,...]

Where:

APPEND causes debug information to be stored in the specified file by appending to an existing file. The specified file must have write permission. If the file does not exist, it is created.

PUT	causes debug information to be stored in the specified file. If the file already exists, a query to overwrite the existing file is displayed. If the PUT command is executed from within an INCLUDE file (see the INCLUDE entry), and the file already exists, the command fails.
filename	specifies the path and file name where information is to be stored.
DEBUG	if entered without options, specifies that all debug definitions are stored.
data-type	specifies that all debug variables of that data type are stored. (see the <date type> entry.)
DEBUGVAR	specifies that all debug variable definitions are stored in the file.
LITERALLY	specifies that all LITERALLY definitions are stored in the file.
POINTER	specifies that all POINTER definitions are stored in the file.
PROC	specifies that all PROC definitions are stored in the file.
dir-name	specifies an emulator object directory such as GLOBAL// or LOCAL//.
wild-obj-identifier	specifies the name of a debug object. The * and ? can be used as wild-card characters.
DATABASE	stores a binary representation (interpreted by the emulator software) of all currently defined debug objects and the emulator parameters to the file specified.

Discussion

Use the APPEND or PUT commands to store objects created during a debug session in a disk file. This information can be stored in two forms: a text file or an emulator data base file. Objects stored in the text file format are stored using the REDEFINE command. The text file can then be included (see the INCLUDE entry) during later debug sessions. A data base file can be restored by using the -r invocation option when invoking the emulator software.

When the emulator data base is stored, full path names are stored for the supporting files unless they reside in the current working directory. When a data base is restored, if the full path names are no longer valid, the data base is not properly restored.

Also, when a debug procedure (PROC) is defined, a temporary file is used to store information about the PROC. The directory where this temporary file is stored depends on the designation of the host operating system TMP or TEMP environment parameters. The full path name for the temporary file is recorded in the data base. When a binary representation of the data base is stored with the EXIT or PUT command, this full path name is also stored. If the data base is restored on a system that does not contain this path, any attempt to execute or read the contents of the PROC results in an error.

When storing PROC definitions on disk using the APPEND or PUT command, LITERALLY definitions used within the PROC are not expanded. When restoring the PROCs at a later time using the INCLUDE command, if the LITERALLY definitions do not exist, an error results.

APPEND/PUT (continued)

Examples

1. The following example shows how to check the listing of the current DEBUG definitions and use the PUT command to store all the current LITERALLY definitions to a file.

```
hlt> DIR DEBUG
LOCAL//
    def         literally
    lit         literally
      i           ord1
      j           ord2
      k           int2
hlt> PUT C:myfile LITERALLY
hlt>
```

2. The following example shows how to use the APPEND command to store all the PROC definitions to an existing file.

```
hlt> APPEND C:myfile PROC
hlt>
```

3. The following example shows how to store a binary representation of the data base to a file.

```
hlt> PUT C:\tmp\mybase DATABASE
hlt>
```

Cross-References

Chapter 2
DEFINE
INCLUDE
PROC
PUT
REDEFINE

ASM / SASM

Displays or modifies memory in ASM86 assembly instructions

Syntax

```
{      }  { addr [= instructions [,...]] }
{ ASM  }  { addr TO addr                 }
{ SASM }  { addr LENGTH expr             }
```

Where:

ASM	assembles or disassembles memory.
addr	specifies a single address or an expression that evaluates to an address (see the Address Translation entry).
TO	specifies an ending address. If the ending address is in the middle of an instruction, for disassembly, that instruction is not completely disassembled. For assembly, the instruction is truncated.
LENGTH	specifies the number of instructions to be assembled or disassembled.
expr	specifies a number or an expression which indicates the number of instructions to be disassembled or assembled (not the number of code bytes). (See the <expr> entry.)
instructions	specifies one or more ASM86 assembly instructions delimited by a comma. Symbolic addresses are allowed when <expr> is displayed in the screen's syntax menu.

Discussion

You can use either the ASM or the SASM command to display and modify program memory in ASM86 assembly instructions. The two commands are functionally identical and interchangeable.

ASM / SASM (continued)

Displaying Memory

To display a portion of memory as assembler mnemonics, specify a *partition* when entering ASM or SASM. A single address reference displays the first instruction at that address. A range of addresses, specified as *address* TO *address* or *address* LENGTH *number-of-instructions*, displays all instructions that start within that range.

The disassembled instructions are displayed in columns, as shown below:

```
hlt> ASM 31KP LENGTH 5
07c00HP     e504          IN      AX,04H
07c02HP     a30e00        MOV     WORD PTR 000eH,AX
07c05HP     8b0e0c00      MOV     CX,WORD PTR 000cH
07c09HP     2bc8          SUB     CX,AX
07c0BHP     ea29002100    JMP     0021H:0029H FAR
```

From left to right, the display columns contain the instruction address, hexadecimal object value, opcode mnemonic, and operands. Some of the operands may include comments that provide additional information, such as the types of jumps, calls, and returns, the address of a branch relative to the current execution point, and the decimal equivalents of hexadecimal values.

The disassembly also includes symbols, modules, and line number information when the following conditions are true:

- SYMBOLIC is set TRUE (non-zero).
- The *segment* and *offset* values can be determined from the *address* (if the program is in 8086 absolute OMF).
- The symbol table contains an exact match to the beginning of an instruction in the *partition*.

The disassembly does not include symbolic information if a physical address is used to specify the *partition*.

Modifying Memory

To modify a portion of program memory, assign one or more assembler-mnemonics to a specific address when entering ASM or SASM. For example:

```
hlt> ASM 200HP = 'JMP BX'
00200HP     ffe3     JMP     BX NEAR
```

The emulator single-line assembler loads the specified address with the assigned 80C186 instructions.

The instructions must be specified in ASM86 macro assembler mnemonics, as described in the *ASM86 Assembly Language Reference Manual*. The syntax, in most cases, is identical to ASM86.

Assembler Directives

The single-line assembler does not support assembler directives.

Assembler Operators

The assembler type operators recognized by the single-line assembler are the following:

$	specifies the current offset. ($ is the execution pointer.)
BYTE PTR	specifies a 1-byte number.
WORD PTR	specifies a 16-bit number.
DWORD PTR	specifies a 4-byte number.
FAR PTR	specifies that both the CS and IP take part in a JMP or a CALL.
FAR	specifies an inter-segment operation.
NEAR	specifies an intra-segment operation.
segment override prefixes	specifies that an operand is to be taken from a non-default segment.

Jumps and Calls

ASM86 supports five types of jumps: direct-short, direct-near, indirect-near, direct-far, and indirect-far. You can use the single-line assembler to produce all of these jumps, except for a direct-short. Substitute direct-near jumps for any direct-short jumps.

The single-line assembler uses an offset from the current IP ($) as the operand for a direct-near jump. For example, to load at physical address 100HP with a direct-near instruction that jumps to 105HP, enter the following:

```
hlt> ASM 100HP = 'JMP $ +5'
00100HP    eb03          JMP     $ +05H ;A=00000105H
```

The relative offset need not be calculated if you use expressions. To load 014CHP with a direct-near jump to 1FFH, enter the following:

```
hlt> ASM 014CHP = 'JMP $ + (1FF - 14C)'
0014cHP    e9b000      JMP  $ +00b3H ;A=000001ffH NEAR
```

To load address 100HP with a direct-near instruction that jumps to offset 00FCH, enter the following:

```
hlt> ASM 100HP = 'JMP $ - (100 - 0FC)'
00100HP    ebfa        JMP    $ -04H ;A=000000fcH
```

To load 200HP with an instruction that jumps to the offset contained in BX, enter the following:

```
hlt> ASM 200HP = 'JMP BX'
00200HP   ffe3          JMP    BX NEAR
```

Use the operator WORD PTR for indirect-near jumps. For example, to load 200HP with an instruction that jumps to the offset stored in the memory location offset in BX, enter the following:

```
hlt> ASM 200HP = 'JMP WORD PTR [BX]'
000200HP   ff27         JMP   WORD PTR [BX] NEAR
```

The single-line assembler recognizes a direct-far jump by the FAR PTR operator. For example, to load location 3:300H with an instruction that jumps to location 12:34H, enter the following:

```
hlt> ASM 3:300H = 'JMP FAR PTR 12:34H'
0003:0300H   ea34001200   JMP    0012:0034H FAR
```

As an example of an indirect-far jump, load 400HP with an instruction that jumps to the segment and offset stored in the memory location offset in BX. Enter the following:

```
hlt> ASM 400HP = 'JMP DWORD PTR [BX]'
00400HP    ff2f      JMP     DWORD PTR [BX] FAR
```

RET (RETURN)

To return from a far jump or call, the single-line assembler requires that you append the operator NEAR or FAR. ASM86 detects whether a procedure is near or far, and consequently, it generates the appropriate return. Because the single-line assembler does not have this information, a near return or a far return must be specified. With the single-line assembler, specify a near return as RET NEAR and a far return as RET FAR. For example, to load 500HP with a far return that discards three bytes from the stack after returning, enter the following:

```
hlt> ASM 500HP = 'RET FAR 3'
00500HP    ca0300    RET    0003H FAR
```

Absolute Addresses

The single-line assembler allows an absolute address within a JMP or CALL instruction. For example, the single-line assembler recognizes the instruction JMP FAR PTR 12:34H. This instruction is a direct-far jump.

Indirect Addressing

ASM86 and the single-line assembler support a variety of indirect address specifications. For example, the following instructions assemble to the same value:

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

Patching Code

You can use the single-line assembler to patch your program by replacing an instruction with a jump to an unused memory area and filling in this unused area with necessary code. The final instruction in this area must be a jump back to the program.

For example, assume that your program resides within the first 30K bytes of memory. The patch is made at location 0021:0023H. At this point in the code, the program is reading I/O ports. Assume that some additional I/O ports need to be read.

The initial program code is as follows:

```
hlt> ASM :cmaker#4 TO :cmaker#7+1
:CMAKER#4
0021:0019H  e502          IN    AX,02H
0021:001bH  a30c00        MOV   WORD PTR 000cH,AX
:CMAKER#5
0021:001eH  e502          IN    AX,02H
0021:0020H  a30a00        MOV   WORD PTR 000aH,AX
:CMAKER#6
0021:0023H  8b0e0c00      MOV   CX,WORD PTR 000cH
0021:0027H  2bc8          SUB   CX,AX
0021:0029H  890e0600      MOV   WORD PTR 0006H,CX
:CMAKER#7
0021:002dH  89c8          MOV   AX,CX
```

Insert a jump at location 21:23H:

```
hlt> ASM 21:23H = 'JMP FAR PTR 0:31K'
0021:0023H  ea007c0000      JMP    0000H:7c00H FAR
```

This is a five-byte instruction. Add a NOP to get the instruction stream back into sequence:

```
hlt> ASM 21:28H = 'NOP'
0021:0028H     90         NOP
```

Now the initial code looks as follows:

```
hlt> ASM :cmaker#4 TO :cmaker#7+1
:CMAKER#4
0021:0019H  e502          IN    AX,02H
0021:001bH  a30c00        MOV   WORD PTR 000cH,AX
:CMAKER#5
0021:001eH  e502          IN    AX,02H
0021:0020H  a30a00        MOV   WORD PTR 000aH,AX
:CMAKER#6
0021:0023H  ea007c000000  JMP   0000H:7c00H FAR
0021:0028H  90            NOP
0021:0029H  890e0600      MOV   WORD PTR 0006H,CX
:CMAKER#7
0021:002dH  89c8          MOV   AX,CX
```

Now put in the patch. Read a value from I/O port 4 and load it into data segment offset 000EH:

```
hlt> ASM 0:31K = 'IN AX,4','MOV WORD PTR 000EH,AX'
0000:7c00H  e504          IN    AX,04H
0000:7c02H  a30e00        MOV   WORD PTR 000eH,AX

hlt> ASM 0:31K LENGTH 2
0000:7c00H  e504          IN    AX,04H
0000:7c02H  a30e00        MOV   WORD PTR 000eH,AX
```

Once the desired code is added, include the instructions that were replaced before jumping back to the initial code. This example assumes that only one additional port is to be read:

```
hlt> ASM 31K + 5P = 'MOV CX,WORD PTR 000CH',\
hlt>> 'SUB CX,AX','JMP FAR PTR 21:29'
07c05HP     8b0e0c00      MOV    CX,WORD PTR 000cH
07c09HP     2bc8          SUB    CX,AX
07c0bHP     ea29002100    JMP    0021:0029H FAR

hlt> ASM 31KP LENGTH 5
07c00HP     e504          IN     AX,04H
07c02HP     a30e00        MOV    WORD PTR 000eH,AX
07c05HP     8b0e0c00      MOV    CX,WORD PTR 000cH
07c09HP     2bc8          SUB    CX,AX
07c0bHP     ea29002100    JMP    0021:0029H FAR
```

The patch is in. Now the program jumps to absolute location 07C00HP, executes the patch code, then returns to the program.

The Default Base

The single-line assembler assumes the current base (as determined by BASE). To override the current base for an individual number, append one of the following suffixes to the number:

Suffix	Base
Y	binary
Q	octal
T	decimal
H	hexadecimal

String Moves

ASM86 provides the mnemonics MOVS, MOVSB, and MOVSW. However, the SI and DI registers must be previously loaded.

The MOVS mnemonic requires two operands: the name of the destination string (the symbolic name for the first location) and the name of the source string. ASM86 uses these operands to determine whether bytes or words are being moved.

The MOVSB and MOVSW mnemonics do not require operands because the B and W identify whether bytes or words are being moved.

The single-line assembler accepts MOVS, MOVSB, and MOVSW mnemonics. If MOVS is used, the single-line assembler prompts for BYTE PTR or WORD PTR operators prior to entering registers.

Symbolic References

Symbolic references are allowed as part of the partition in memory disassembly. For example:

```
hlt> ASM main_ LENGTH 20T
```

or

```
hlt> ASM :program.start_ LENGTH 5
```

Symbolic references are also allowed for "CALL FAR PTR *segment:offset*" and "JMP FAR PTR *segment:offset*" single-line assembly instructions. For example:

```
hlt> ASM 7C01HP = 'JMP FAR PTR main_'
```

Coprocessor Support

The emulator software does not support the assembly or disassembly of coprocessor instructions.

Examples

1. This example displays the assembly instructions starting at physical address 206HP.

```
hlt> ASM 206HP LENGTH 3T
00206HP   fa          CLI
00207HP   50          PUSH    AX
00208HP   8c1e0a00    MOV     WORD PTR 000aH,DS
```

ASM / SASM (continued)

2. This example modifies memory at physical address 7C00HP.

```
hlt> ASM 7C00HP = 'JMP WORD PTR [BX]','NOP'
07c00HP   ff27          JMP       WORD PTR [BX] NEAR
07c02HP   90            NOP
```

Cross-References

$
Address Translation
BASE
<expr>
Memory Access
Partition
SYMBOLIC
Symbolic References

BASE

Displays or modifies the number base

Syntax

```
BASE [= expr]
```

Where:

BASE — if entered without an option, displays the current setting of the number base.

expr — specifies a number or an expression which evaluates to one of the following values. If any other value is entered, an error message is displayed.

- 2T indicates binary
- 8T indicates octal
- 10T indicates decimal
- 16T indicates hexadecimal

Discussion

Use the BASE parameter to display or change the number base. All input is interpreted according to the current base except in the presence of a base suffix. All numeric output is displayed in the current base except for some special cases (e.g., real numbers are always displayed in decimal).

The override suffixes are as follows:

- Y binary
- Q octal
- T decimal
- H hexadecimal

NOTE

Use a base suffix when setting the base to ensure correct results. For example, BASE = 10 is ineffective and results in no change to the base regardless of the current base.

BASE (continued)

Examples

1. The following example shows how to display the current number base.

```
hlt> BASE
0010H
hlt>
```

2. The following example shows how to set the current number base to decimal.

```
hlt> BASE = 10T
hlt> BASE
10T
hlt>
```

3. The following example shows how to set the processor FLAGS register by using the FLAGS register command with a base suffix to override the default number base.

```
hlt> FLAGS = 0010011000001001Y
hlt>
```

Cross-References

<expr>
Parameters

BREAK

Exits from a control block

Syntax

BREAK

Discussion

The BREAK statement causes termination of the smallest enclosing WHILE, DO ... WHILE, FOR, or SWITCH control block. Control passes to the statement following the terminated block. This statement only affects control blocks. To break from emulation, see the HALT entry.

Example

The following example shows how a BREAK statement is used to terminate the WHILE loop when the variable n equals 0.

```
hlt> DEFINE INT2 n = 5
hlt> WHILE (1)
hlt>> {
hlt>> n -= 1
hlt>> IF (n == 0)
hlt>> BREAK
hlt>> }
```

Cross-References

CONTINUE
DO ... WHILE
FOR
SWITCH
WHILE

BTHRDY

Allows emulator memory to simulate EPROM (or slower) memory via the READY signal

Syntax

BTHRDY [= *bool-expr*]

Where:

BTHRDY	if entered without an optional expression, displays the current BTHRDY setting.
bool-expr	is any expression that evaluates to TRUE (non-zero) or FALSE (zero).

Default

FALSE

Discussion

Use the BTHRDY tool variable to specify the source of the processor READY signal for ICE-mapped memory accesses. The READY signal can originate from your prototype (or target system) or the emulator.

If BTHRDY is TRUE, the emulator uses the logical AND of the target system READY signal and the emulator READY signal to determine the number of wait states. Since the emulator READY signal is always valid (zero wait states), the processor essentially waits for a valid READY signal from the target system. Thus, the timing of ICE-mapped memory accesses is dependent on the target system.

If BTHRDY is FALSE, the processor waits for the emulator READY signal only. It ignores the target system's READY signal during ICE-mapped memory accesses.

BTHRDY has no effect on USER-mapped memory accesses. In this case, the READY signal always originates from the target system.

You can set BTHRDY only when the emulator is halted; however, you can display the current BTHRDY setting at any time.

Example

This example displays the current setting of BTHRDY, then changes the setting.

```
hlt> BTHRDY
TRUE
hlt> BTHRDY=0
hlt>
```

Cross-References

<expr>
MAP
Memory Access
TOOLVAR

BUSACT

Enables an emulator time-out when
the emulator processor bus
is inactive for approximately one second

Syntax

BUSACT [= *expr*]

Where:

BUSACT	if displayed without an option, displays the current setting.
expr	specifies a number or an expression which must evaluate as either TRUE (non-zero) or FALSE (zero). The default setting is TRUE.

Discussion

Use the BUSACT tool variable to enable or disable an automatic break from emulation when there is no bus activity for more than one second.

When BUSACT is set to TRUE, the emulator software attempts to cause a break whenever the processor bus has been inactive for approximately one second. A message describing the condition that caused the bus inactivity is displayed after a successful break. If the emulator software cannot cause a break, a message that the processor is hung appears on the screen.

When BUSACT is set to FALSE and the bus has been inactive for approximately one second, the emulator issues a warning message that the processor is hung and emulation continues. This tool variable is useful for emulating through a processor reset or emulating through long periods of bus inactivity.

A BUSACT time-out can be caused by any of the following: a processor halt/shutdown state, HLDA granted by the processor, BUSY line active, and mapped I/O accesses when the emulator is servicing an interrupt in ENI mode. When a BUSACT time-out occurs, the emulator displays an error message appropriate to the cause of the time-out.

Examples

1. The following example shows how to display the current value of BUSACT.

```
hlt> BUSACT
TRUE
hlt>
```

2. The following example shows how to disable automatic break when the bus is inactive for more than one second.

```
hlt> BUSACT = 0
hlt>
```

3. The following example shows how to assume the program in memory causes a shutdown to occur and a target system reset does not occur within the one second after shutdown occurs. Set BUSACT to TRUE and use the GO FOREVER command to cause the emulator software to break emulation and return to interrogation when the shutdown occurs.

```
hlt> QUERY = FALSE
hlt> BUSACT = TRUE
hlt> GO FOREVER

/* bus failure occurs and an error message is displayed*/
hlt>
```

Cross-References

CAUSE
<expr>
GO
IOREADY
MEMRDY
STATUS
TOOLVAR

CALLSTACK

Displays the names of procedures on the stack

Syntax

```
CALLSTACK [expr]
```

Where:

CALLSTACK	displays the names of the procedures on the stack (in order of call).
expr	specifies the number of references in the display.

Discussion

Use the CALLSTACK command with high-level languages to display the current chain of procedure calls in the program being executed. A CALLSTACK is a fully qualified list of references to procedures. The reference listed first is the current execution address. The second entry is the return address for the procedure that called the current procedure, etc. CALLSTACK shows the dynamic, run-time nesting of the program as opposed to the static, lexical nesting.

The display format, one procedure per line, is as follows:

```
:module-name [.procedure-name] [#line-number] [+ offset]
```

The *offset* is the number of bytes from the preceding line, procedure, or module.

The emulator CALLSTACK command can operate only if the current execution pointer ($) is inside a module for which the emulator software has symbol information.

The CALLSTACK command cannot operate correctly if the current execution pointer is within a procedure prelude, if the nesting sequence includes a procedure in assembly language, or if the last executable statement of the main module calls a procedure.

CALLSTACK supports the Intel 8086 absolute Object Module Format.

CAUTION

The emulator software can evaluate the stack differently at any given time. This evaluation is based on the current CS, SS, and SP. Manually changing any of these values as well as the execution pointer ($) can invalidate the CALLSTACK command results.

Example

The following example shows how to display the current chain of procedure calls. Assume a program with symbolic information has been loaded and executed.

```
hlt> CALLSTACK
:parse.get_token#21 + 45H
:parse.factor#42 + 7H
:parse.term#25 + 18H
:parse.expr#77 +73H
:parse#49
hlt>
```

Cross-References

<expr>
LOAD
LSTEP
NAMEFRAME
PSTEP
Symbolic References

CAUSE

Displays the reason emulation stopped

Syntax

```
CAUSE
```

Discussion

Use the CAUSE command to display the reason for the last break in emulation. The CAUSE command is useful when a break message has scrolled off the screen (due to entering other commands). The CAUSE message describes the location and reason for the break. The possible reasons for an emulation break are listed below:

- Activity Monitor break occurred (see the GO command entry).
- HALT command was entered (see the HALT command entry).
- Emulator attempted to write to read-only memory (see the MAP command entry).
- Emulator is waiting for user input (see the MAPIO, HOLDIO, or RELEASEIO entry).
- Processor bus is idle (see the BUSACT entry).
- READY signal is not responding on I/O access (see the IORDY entry).
- READY signal is not responding on memory access (see the MEMRDY entry).
- Single-step break occurred (see the ISTEP, LSTEP, PSTEP, or STEP command entry).
- No user clock.
- Power failure occurred.

Example

The following example illustrates a typical display using the CAUSE command.

```
hlt> CAUSE
Emulator last stopped because:
  of HALT command.
  EXECPOINTER is:  :MESSG:DELAY_A_LITTLE_BIT_#163 + 3H (1000:0051H)
hlt>
```

Cross-References

BUSACT
GO
HALT
HOLDIO
IORDY
ISTEP
LSTEP
MAP
MAPIO
MEMRDY
PSTEP
RELEASEIO
STEP

Character Functions

Built-in functions for character classification and transformation

Discussion

Two classes of character functions exist: character classification and character transformation.

The character classification functions return a BOOLEAN data type with the value non-zero (TRUE) or zero (FALSE). They take a single argument (*char-expr*) that must be compatible with the INT4 data type. The character classification functions include:

ISALPHA	ISUPPER	ISLOWER	ISDIGIT
ISXDIGIT	ISSPACE	ISPUNCT	ISPRINT
ISCNTRL	ISASCII	ISALNUM	

The character transformation functions return an INT4 containing an ASCII-coded value. They take a single argument that must be compatible with the INT4 data type. The character transformation functions include:

TOUPPER	TOLOWER	TOASCII

Descriptions for all of the character functions are in Table 6-2. All of the character functions use the following syntax:

[*obj-identifier* =] *function* (*char-expr*)

Where:

obj-identifier	specifies the debug object to which the function's return value is assigned. If *obj-identifier* is not specified, or the return value is not used by another command, the return value is displayed on the next line of the screen.
function	specifies the name of the function.
(*char-expr*)	specifies a quoted character, a string of characters, or an expression specifying a character or a string of characters. The parentheses are required.

Table 6-2 Character Functions

Function	Discussion
ISALPHA	The ISALPHA function returns TRUE when *char-expr* is alphabetic. The ASCII hexadecimal values for these characters are 41 - 5A (A ... Z) and 61 - 7A (a ... z).
ISUPPER	The ISUPPER function evaluates to TRUE when *char-expr* is an upper-case alphabetic character. The ASCII hexadecimal values for these characters are 41 - 5A (A ... Z).
ISLOWER	The ISLOWER function evaluates to TRUE when *char-expr* is a lower-case alphabetic character. The ASCII hexadecimal values for these characters are 61 - 7A (a ... z).
ISDIGIT	The ISDIGIT function evaluates to TRUE when *char-expr* is a numeric digit. The ASCII hexadecimal values for these characters are 30 - 39 (0 ... 9).
ISXDIGIT	The ISXDIGIT function evaluates to TRUE when *char-expr* is a hexadecimal digit. The ASCII hexadecimal values for these characters are 30 - 39 (0 ... 9), 41 - 46 (A ... F), and 61 - 66 (a ... f).
ISALNUM	The ISALNUM function evaluates to TRUE when *char-expr* is alphanumeric. The ASCII hexadecimal values for these characters are 41 - 5A (A ... Z), 61 - 7A (a ... z), and 30 - 39 (0 ... 9).
ISSPACE	The ISSPACE function evaluates to TRUE when *char-expr* is a blank. This blank can be a single space (ASCII hexadecimal value 20), carriage return, line feed (new line or '\n'), tab ('\t'), vertical tab, or form feed (new page or '\p').

Table 6-2 Character Functions (continued)

Function	Discussion
ISPUNCT	The ISPUNCT function evaluates to TRUE when *char-expr* is a punctuation mark (neither a control nor an alphanumeric character). The ASCII hexadecimal values for these characters are 21 - 2F, 3A - 40, 5B - 60, and 7B - 7E.
ISPRINT	The ISPRINT function evaluates to TRUE when *char-expr* is a printable character. The ASCII hexadecimal values for these characters are 20 - 7E.
ISCNTRL	The ISCNTRL function evaluates to TRUE when *char-expr* is a delete character (ASCII hexadecimal 7F) or any control character (ASCII hexadecimal 0 - 1F).
ISASCII	The ISASCII function evaluates to TRUE when *char-expr* is an ASCII-coded value (hexadecimal 0 - 7F).
TOUPPER	The TOUPPER function returns the upper-case value of *char-expr*. If *char-expr* does not contain a lower-case letter, the result is the original *char-expr*, unchanged. The *char-expr* itself is not changed.
TOLOWER	The TOLOWER function returns the lower-case value of *char-expr*. If *char-expr* does not contain an upper-case letter, the result is the original *char-expr*, unchanged. The *char-expr* itself is not changed.
TOASCII	The TOASCII function clears all bits of *char-expr* that are not part of a standard ASCII character, and returns this value. The *char-expr* itself is not changed.

Examples

1. This example demonstrates the use of character classification functions.

```
hlt> DEFINE CHAR cvar = 'a'
hlt> DEFINE INT4 ivar
hlt> ivar = cvar
hlt> ivar
00000061H
hlt> ISALPHA (cvar)
TRUE
hlt> ISALPHA (ivar)
TRUE
hlt> DEFINE INT4 answer = ISALPHA (cvar)
hlt> answer
00000001H
hlt> cvar
'a'
hlt> ISUPPER (cvar)
FALSE
hlt> ISLOWER (cvar)
TRUE
hlt> cvar = 'A'
hlt> ISUPPER (cvar)
TRUE
hlt> ISDIGIT (cvar)
FALSE
hlt> ISXDIGIT (cvar)
TRUE
hlt> ISALNUM (cvar)
TRUE
hlt> ISSPACE (cvar)
FALSE
hlt> ivar = 20
hlt> ISSPACE (ivar)
TRUE
```

```
hlt> cvar = '!'
hlt> ISPUNCT (cvar)
TRUE
hlt> ISPRINT (ivar)
TRUE
hlt> cvar = 5
hlt> cvar
'\005'
hlt> ISPRINT (cvar)
FALSE
hlt> ISCNTRL (cvar)
TRUE
hlt> ISASCII (cvar)
TRUE
```

2. This example demonstrates character transformation functions.

```
hlt> DEFINE INT4 ivar = 5
hlt> DEFINE CHAR cvar
hlt> cvar = TOASCII (ivar)
hlt> cvar
'\005'
hlt> cvar = TOASCII (61)
hlt> cvar
'a'
hlt> TOASCII (cvar)
00000061H
hlt> TOUPPER (cvar)
00000041H
hlt> cvar
'a'
hlt> cvar = TOUPPER (cvar)
hlt> cvar
'A'
hlt> ivar = 41
hlt> cvar = TOLOWER (ivar)
hlt> cvar
'a'
hlt>
```

3. The following example demonstrates using a character classification function in a procedure.

```
hlt> REDEFINE LSTRING var1 = "abc"     /* define a string of characters */
hlt> REDEFINE PROC checker              /* define a PROC to verify that */
hlt>> {                                        /* var1 is a number     */
hlt>> IF (ISDIGIT (var1))
hlt>> PRINTF ("Number accepted, continue... \n") ELSE
hlt>> PRINTF ("\007ERROR: Enter a number... \n")
hlt>> }
hlt> checker                                          /* execute the PROC */
(beep) ERROR: Enter a number...
hlt> var1 = "123"                              /* change the value of var1 */
hlt> checker                                          /* execute the PROC */
Number accepted, continue...
hlt>
```

Cross-References

C: A Reference Manual
DEFINE
<expr>
Functions
IF ... ELSE
I/O Functions
PROC
REDEFINE

CLK

Determines the format of the time-stamp information in the PRINT CYCLES display

Syntax

```
CLK  [= expr]
```

Where:

CLK	if entered without an optional expression, it displays the current CLK setting.
expr	is a REAL4 expression that indicates the CLKOUT processor clock rate of the target system. Frequency is assumed to be expressed in MHz.

Default

0.0 (No frequency specified; time-stamp format in PRINT CYCLES display is CLOCKS instead of TIME.)

Discussion

Use the CLK command to specify the format of the time-stamp column in the PRINT CYCLES display (see the PRINT entry in this chapter).

The format of the time-stamp column can be either CLOCKS (the number of clock ticks between acquisitions) or TIME (the amount of time between acquisitions).

To specify the CLOCKS format, set CLK to 0.0. To specify the TIME format, set CLK to the frequency, in MHz, of the CLKOUT processor clock signal in your prototype (or target system). The emulator uses the specified clock speed to convert the clock ticks into time.

You can change or display the CLK setting at any time.

Example

In this example the CLK value is set to 8.5 MHz.

```
hlt> CLK = 8.5
  8.50000t mHz
hlt>
```

Cross-References

<expr>
PRINT

CONTINUE

Transfers control from within a control block to the end of the block

Syntax

```
CONTINUE
```

Discussion

The CONTINUE statement causes a jump to the end of the immediately enclosing iteration statement (WHILE, DO, or FOR).

Example

This example shows a CONTINUE statement within an IF statement that is nested in a FOR loop. The variable "i" contains the amount of numbers between 0 and 12 whose modulo equals 2.

```
hlt> DEFINE INT2 a
hlt> DEFINE INT2 i = 0
hlt> FOR (a = 0 ; a <= 12 ; a += 1)
hlt> {
hlt>> IF ((a MOD 3) != 2)
hlt>> CONTINUE
hlt>> i = i + 1
hlt>> }
```

Cross-References

BREAK
DO ... WHILE
<expr>
FOR
WHILE

COPROC187

Notifies the emulator
of the presence of a 80C187
Numeric Processor Extension Unit

Syntax

COPROC187 [= *bool-expr*] [*address*]

Where:

COPROC187	if entered without an optional expression, displays the current COPROC187 setting.
bool-expr	is any expression that evaluates to TRUE (non-zero) or FALSE (zero).
address	is the starting location in mapped memory for saving or restoring the 80C187 register contents.

Default

FALSE

Discussion

Use the COPROC187 command to enable and disable accesses to 80C187 numeric processor extension unit registers. Setting the variable to TRUE enables accesses to 80C187 numeric processor extension unit registers, while setting it to FALSE disables accesses.

If you set COPROC187 to TRUE and a 80C187 numeric processor extension unit is not installed in your prototype, the emulator may lock up when it attempts to access coprocessor registers.

You can display or change COPROC187 at any time.

The Address Option

The first time COPROC187 is set to TRUE, the *address* option must be specified. This option specifies the starting address of a contiguous 110 byte block of mapped memory RAM. The program may also use this memory, since 80C187 commands store the 110

bytes of code before saving or restoring the 80C187 registers and then restore the previous user code after register operations.

It is recommended that you not use the memory for ENI service area, or interrupt code, since the memory could be swapped out during an interrupt service request.

NOTE

To display 80C187 stack registers (ST0 - ST7), you must have an 80287 coprocessor installed on your IBM PC AT system board.

Floating point operations may fail if a 80C187 numeric processor extension unit is not present in the host development system. For example, on a DOS system, entering the following emulator commands will generate a system floating point error that will cause an exit to DOS:

```
-> DEFINE REAL4 R4
-> R4=1.0e100

ERROR 2100: FLOATING POINT ERROR: OVERFLOW

C>
```

In this second example, trying to access the 80C187 stack registers (ST0 - ST7), the emulator commands may generate a system error that will also cause an exit to DOS:

```
hlt> ST0

hlt> ERROR 2003: INTEGER DIVIDE BY 0

C>
```

The solution is to install an 80287 math coprocessor in the host system. This problem does not affect support for the 80C187 numeric processor extension unit in a target system except to prohibit the display of stack elements.

Example

This example fails to access 80C187 numeric processor extension unit registers because COPROC187 is FALSE. After setting COPROC187 to TRUE, access to the registers is successful.

```
hlt> R187

[ICE186] Error #150t
COPROC187 is 0, indicating that there is no 80C187 numeric processor
extension unit in the prototype.
hlt> COPROC187=1 100:0
hlt> R187
FCW = 037fH
  Exceptions Mask:   PM  UM  OM  ZM  DM  IM
                      1   1   1   1   1   1
  Precision Control: 64-bit Significand
  Rounding Control: Round to nearest or even
FSW = 7901H    Top of stack: Reg7
FTW = 38f0H
FIA = 00040010H
FIO =     04c3H
FDA = 000cd340H
ST0:  VALID     00000000000000c0ff3f   1.50000000000000000e+0
ST1:  EMPTY     00000000000000a00040
ST2:  ZERO      00000000000000000000   0
ST3:  VALID     00000000000000a00040   2.50000000000000000e+0
ST4:  VALID     00989999999999d90040   3.40000000000000000e+0
ST5:  EMPTY     00989999999999d90040
ST6:  SPECIAL   00989999999999d90040   Infinity
ST7:  VALID     00989999999999d90040   3.40000000000000000e+0
```

Cross-References

<expr>
R187
Register Access 80C187

COUNT

Groups and executes commands
a specified number of times
or until a specified condition occurs

Syntax

```
COUNT expr

 [commands]

[ WHILE bool-cond ] [...]
[ UNTIL bool-cond ]

END[COUNT]
```

Where:

COUNT	initiates the execution of the command in the loop. COUNT can be nested as deeply as host memory allows.
expr	specifies an integer or expression that evaluates to a positive integer. *Expr* specifies the number of times the loop executes.
commands	specifies one or more emulator commands that are executed until the test condition is met or the terminating count is reached. The DO, EDIT, FOR, IF, INCLUDE, and SWITCH commands cannot be used.
WHILE	causes the loop to terminate immediately when *bool-cond* evaluates to FALSE. Execution continues with the next command after the end of the COUNT block. Multiple WHILE statements can be used in a COUNT block.
UNTIL	causes the loop to terminate immediately when *bool-cond* evaluates to TRUE. Execution continues with the next command after the end of the COUNT block. Multiple UNTIL statements can be used in a COUNT block.

COUNT (continued)

bool-cond	specifies a boolean condition, a number, or an expression that evaluates to TRUE (non-zero) or FALSE (zero).
END	(or ENDCOUNT) terminates the COUNT block.

Discussion

The COUNT statement groups and executes commands a specified number of times or until a condition occurs. Unless it is within a procedure definition, a COUNT block is executed as soon as the END statement is entered. COUNT blocks are executed no more than the number of times specified by expr. COUNT blocks containing WHILE or UNTIL clauses exit when the test condition bool-cond is satisfied or the count value is reached, whichever comes first. If expr <= 0, commands are not executed.

Examples

1. The following example shows how to set a variable, *a*, to 1, then execute the following commands 10 times. The COUNT executes 10 times even though the statement *a++* is executed 4 times. The dots show the command block nesting level.

```
hlt> DEFINE INT2 a = 1
hlt> COUNT 10T
.hlt>> IF (a < 5) THEN
.hlt>> a++
.hlt>> ENDIF
.hlt>> ENDCOUNT
0001H
0002H
0003H
0004H
hlt>
```

2. This example executes a command 10 times unless the condition m >= 0 becomes FALSE. The value of i after the execution is 5; m is -10.

```
hlt> DEFINE INT2 a = 10T
hlt> DEFINE INT2 m = 50T
hlt> DEFINE INT2 i = 0
hlt> COUNT 10T
.hlt>> WHILE (m >= 0)
.hlt>> m -= a
.hlt>> i += 1
.hlt>> END
hlt> I
0005H
hlt> M
0FFF6H
hlt>
```

Cross-References

DO_... END
DO ... WHILE
<expr>
FOR
IF_
IF ... ELSE
SWITCH
WHILE

< data type >

Basic program data types used in commands and displays

Syntax (List of Keywords)

```
BITn
BYTE  (also ORD1)
WORD   (also ORD2)
DWORD  (also ORD4)
QWORD  (also ORD8)
INTn
REALn
BOOLEAN
BCDn
CHAR
LSTRING
NSTRING
POINTER ptr-type
```

Where:

BIT*n*	specifies an *n*-bit unsigned value where *n* can be 1 through 32.
BYTE (ORD1)	specifies a 1-byte unsigned value.
WORD (ORD2)	specifies a 2-byte unsigned value.
DWORD (ORD4)	specifies a 4-byte unsigned value.
QWORD (ORD8)	specifies an 8-byte unsigned value.
INT*n*	specifies an *n*-byte signed value where *n* can be 1, 2, 4, or 8.
REAL*n*	specifies an *n*-byte real number where *n* can be 4, 8, or 10. The emulator software requires a math coprocessor in the host development system to use REAL*n* numbers.
BOOLEAN	specifies a boolean value where TRUE is non-zero and FALSE is zero.
BCD*n*	specifies an *n*-digit binary-coded decimal number where *n* can be 1 or 2.

CHAR	specifies a 1-byte single ASCII character as one byte.
LSTRING	specifies a character string. First byte of string indicates number of characters in the string.
NSTRING	specifies a null-terminated character string.
POINTER	specifies a pointer value of type *ptr-type*.
ptr-type	specifies a pointer type, where *ptr-type* can be one of the following:

OFFSET	(16-bit offset)
LPOINTER	(16-bit segment: 16-bit offset)
PHY	(20-bit physical address)

Discussion

Data types classify an item of data according to its composition (for example, integers, characters, or real numbers). Every data object accessed by the emulator software has a type. Except where noted, memory can be accessed by using these data types. The emulator debug objects must be assigned a type when they are specified using the DEFINE command. The data types used by the emulator correspond to the types used by the emulator processor.

Types and Type Classes

Table 6-3 lists the emulator data types by category with their basic definitions. The table also classifies the types by common attributes. In addition to the data types supported by the 8086 family architecture, the emulator supports 32-bit and 64-bit signed and unsigned integers, bit fields from 1 to 32 bits, and 1 and 2 packed-digit binary coded decimals.

Table 6-3 Emulator Processor Data Types

Category	Amount of Memory				
	1 byte	2 bytes	4 bytes	8 bytes	10 bytes
Ordinals	BYTE ORD1	WORD ORD2	DWORD ORD4	QWORD ORD8	
Integers	INT1	INT2	INT4	INT8	
Boolean	BOOLEAN				
Real			REAL4	REAL8	REAL10
Character	CHAR				
Binary Coded Decimal	BCD1 (1 digit) BCD2 (2 digits)				

Category	Amount of Memory				
	1bit	2bits	3 bits	4 bits	5 bits to 32 bits
Bit	BIT1	BIT2	BIT3	BIT4	BIT5 to BIT32

Category	Data Type	Description
String	NSTRING	variable length, null terminated
	LSTRING	variable length
Pointer	POINTER OFFSET	16-bit offset, e.g., 1234
	POINTER LPOINTER	16-bit segment: 16-bit offset, e.g., 1234:6789, (2 field full pointer)
	POINTER PHY	20-bit physical address, e.g., 12345P

String Constants

A character string; constant is a sequence of one or more ASCII printing characters enclosed in double quotation marks. Special or non-printing characters may be included in a string by preceding it with a backslash (\). If the backslash precedes a character that has no special meaning, the two-character sequence is equivalent to the single character following the backslash. (See the PRINTF function under the I/O Functions entry for more information on non-printing characters.) Strings may be continued across line boundaries only by means of the backslash (\) followed by <Enter>. The following are examples of character string constants:

```
"ABCD"
"This is a test."
"The total is 534.876."
```

Debug Pointer Types

The POINTER keyword can be used to define a pointer debug object. When defining a pointer debug object, the actual pointer type (i.e. PHY, OFFSET, LPOINTER) is defined by the emulator software when the object is assigned a value.

Allowable Operations by Type

The data type of an operand dictates what type of operation can be performed on it. This implicit control is illustrated in Table 6-4 by the type of operator that is allowed to be used for the data types recognized by the emulator. The operator groups are defined in Table 6-5.

Type Conversions

Type conversions occur automatically. If the two operands associated with a binary operator are of different types, an implicit type conversion is done to make them both the same type. Before a conversion takes place, however, the object to be converted is expanded to its maximum precision. An error message is generated if the conversion is not allowed.

Table 6-4 Allowable Operations by Operand Type

Operator Groups	Operand Types Ord	Integer	Boolean	Real	BCD	Bit	String	Character	Point
assignment	X	X	X	X	X	X	X	X	X
bitwise	X	X				X		X	
additive	X	X		X		X		X	X
multiplicative	X	X		X*		X		X	
relational	X	X	X	X	X	X	X	X	X
logical	X	X	X			X		X	
& . (address of)	X	X	X	X	X	X	X	X	X

*The modulus operator (%) is not supported for REAL data types.

Table 6-5 Operators by Group

Operator Group	Symbols							
assignment	=							
bitwise	&	|	^	<<	>>	~	&=	|=
	^=	<<=	>>=					
additive	+	-	++	--	+=	-=		
multiplicative*	*		/	%	*=	/=	%=	
relational	==	!=	<	>	<=	>=		
logical	!	&&	||	^^				
(address of)	&	.						

*The modulus operator (%) is not supported for REAL data types.

Examples

1. The following examples list the displays produced by single objects of each type. The displays assume that the contents of the 10 bytes starting at 0 physical are FA, 2E, 8E, 16, 00, 00, BC, 72, 00, and 2E (hexadecimal). The default base is assumed to be hexadecimal.

```
hlt> BCD1 OP
00000HP   10
hlt> BCD2 OP
00000HP   160
hlt> BIT1 OP
00000HP   0
hlt> BIT2 OP
00000HP   2
hlt> BIT32 OP
00000HP   168e2efa
hlt> BOOL1 OP
00000HP   TRUE
hlt> BOOL2 OP
00000HP   TRUE
hlt> BOOL4 OP
00000HP   TRUE
hlt> BOOL8 OP
00000HP   TRUE
hlt> CHAR OP
00000HP   \372
hlt> INT1 OP
00000HP   fa
hlt> INT2 OP
00000HP   2efa
hlt> INT4 OP
00000HP   168e2efa
hlt> INT8 OP
00000HP   72bc0000168e2efa
hlt> ORD1 OP
00000HP   fa
hlt> ORD2 OP
00000HP   2efa
hlt> ORD4 OP
00000HP   168e2efa
hlt> ORD8 OP
00000HP   72bc0000168e2efa
```

```
hlt> REAL4 OP
00000HP   2.297098e-25
hlt> REAL8 OP
00000HP   4.77963297498010e+244
hlt> POINTER PHY OP
00000HP   8e2efaHP
hlt> POINTER LPOINTER OP
00000HP   168e:2efaH
hlt> POINTER OFFSET OP
00000HP   2efaH
```

2. Define a debug variable, "code_begin", of type POINTER, and assign it the value of the current execution point, $.

```
hlt> DEFINE POINTER LPOINTER code_begin=$
```

3. Define a debug variable of type NSTRING and use it in the PRINTF function.

```
hlt> DEFINE NSTRING header = "Row 1   Row 2   Row 3"
hlt> PRINTF ("%s\n", header)
Row 1   Row 2   Row 3
hlt>
```

Cross-References

Address Translation
DEFINE
DIR
<expr>
I/O Functions
Memory Access
POINTER
SHOW

Syntax

```
DEACTIVATE  ICE186
```

Where:

DEACTIVATE	deactivates the emulator.
ICE186	is the name of the emulator (as used with the ACTIVATE command).

DISCUSSION

Use the DEACTIVATE command to terminate communication between the IBM PC AT and the emulator controller unit, without exiting the emulator software.

Deactivation is useful if you encounter a communication error and need to perform a reset. You can deactivate the emulator to terminate its communication link to the IBM PC AT, and then re-activate it (see the ACTIVATE entry in this chapter). This action re-establishes the communication link and resets the commands and variables that access the emulator controller unit (e.g., MAP and MAPIO).

While the emulator is deactivated, you cannot execute commands that directly access either the emulator controller unit or the processor. You can, however, execute commands that are used for the human interface and database or object management (e.g., DEFINE, DIR, PROC, SHOW).

You can execute the DEACTIVATE command at any time.

DEACTIVATE (continued)

Example

This example deactivates and then re-activates the emulator.

```
hlt> DEACTIVATE ICE186
Symbol files(s) removed from data base.
-> ACTIVATE ICE186
IO_DEVICE: COM1
BAUDRATE: 38400

...emulator control software is loaded...

hlt>
```

Cross-References

ACTIVATE
DEFINE

Syntax

```
                ┌ LITERALLY [dir-name] obj-identifier = "string"                    ┐
                │ PROC (see the PROC entry)                                         │
                │                  ┌          ┐                                     │
                │ [STATIC] {data-type} [dir-name]obj-identifier[= expr]             │
                │                  └ POINTER  ┘                                     │
DEFINE          │                                                                   │
                │ SYMBOLGROUP obj-identifier                                        │
                │ KEYWORDGROUP filename                                             │
                │                                                                   │
                │ SYMBOLFILE filename                                               │
                └ TOOL obj-identifier = filename [NOSYMBOLGROUP]                    ┘
```

Where:

DEFINE	signals the creation of a debug object. An error message results if DEFINE is used for an object that already exists. If this occurs, use REDEFINE.
LITERALLY	is used to create an another name or alias for a keyword, command, or debug object.
PROC	specifies a user-defined function or procedure. (See PROC in this chapter.)
dir-name	specifies an emulator directory name, for example, LOCAL// or TOOL//. (See the Object Hierarchy and NAMEPATH entries.)
obj-identifier	specifies a unique, user-defined name for the object being defined.
"string"	specifies the string of characters replaced by *obj-identifier*. The quotation marks are required.
STATIC	indicates that a debug variable is recognized outside a block. Data types are static unless they are defined inside a block (e.g., DO...END).

DEFINE (continued)

data-type	specifies an emulator data type (see the <data type> entry).
POINTER	indicates that a debug pointer is defined (see the <data type> entry).
expr	specifies a number or an expression (see the <expr> entry).
SYMBOLGROUP	Specifies an object hierarchy directory containing user program symbolics (see the NAMEPATH and Object Hierarchy entries for more information).
KEYWORDGROUP	Specifies an object hierarchy sub-directory containing the emulator software keywords (see the NAMEPATH and Object Hierarchy entry for more information).
SYMBOLFILE	makes a symbolfile available.
TOOL	specifies that a tool containing an emulator environment specified in *filename* is to be defined.
NOSYMBOLGROUP	suppresses the creation of the SYMBOLGROUP// directory.

Discussion

Use the DEFINE command to create debug objects. Debug objects customize the debugging environment. Debug objects are LITERALLY definitions, debug variables, and PROCs. Reference a debug object by entering its name.

Object Hierarchy is an entry in this chapter discussing software organization. Along with NAMEPATH, these sections explain the available object types, where they exist and why.

Use the EDIT command to create or change a PROC. Use the PUT command to place debug objects in a text file and the INCLUDE command to include them in future debug sessions. Use the SHOW command to display the definition of a debug object. Use the REMOVE command to eliminate a debug object from the debug session. A debug object can be redefined using the REDEFINE command without first being removed from the debug session. Use the DIR command to view debug objects.

Examples

1. The following examples show how to use the LITERALLY option to abbreviate commands.

```
hlt> DEFINE LITERALLY def = "DEFINE"
hlt> DEF LITERALLY lit = "LITERALLY"
hlt>
```

2. Assuming the LITERALLY definitions in Example 1 have been defined, the following examples show how to use the LITERALLY option to rename data types.

```
hlt> def lit wd = "WORD"
hlt> def lit dw = "DWORD"
hlt>
```

3. The following example shows how to create a debug variable named first and assigns the value 45H to it. Note that the previously defined LITERALLY, *wd*, is used as the data type.

```
hlt> DEFINE wd first = 45H
```

4. The following examples show how to define debug variables.

```
hlt> DEFINE BYTE zero = 0
hlt> DEFINE INT2 max = 400
hlt>
```

DEFINE (continued)

5. The following example uses the PROC option to define a procedure that raises the first number (arg1) to the power of the second number (ar2).

```
hlt> DEFINE PROC power (arg1,arg2)
hlt>> DEFINE INT1 arg1                    /* define arg1 as an integer */
hlt>> DEFINE INT1 arg2                     /* define ar2 as an integer */
hlt>> {
hlt>> DEFINE INT1 index
hlt>> DEFINE DWORD result = 1                /* initialize result to 1 */
                                               /* and reserve 4 bytes */
hlt>> FOR (index = 1; index <= arg2; index += 1)     /* loop as long as */
                                                       /* index <= arg2 */
hlt>> result=result * arg1                /* calculate the final result */
hlt>> RETURN(result)                   /* return the result to the caller */
hlt> }
```

6. The following example shows how to define a tool.

```
hlt> DEFINE TOOL ICE = tool.cfg
hlt>
```

Cross-References

<data type>
DIR
EDIT
<expr>
INCLUDE
NAMEPATH
Object Hierarchy
POINTER
PROC
REDEFINE
REMOVE
PUT/APPEND
SHOW

Syntax

```
                 ┌ DEBUG [identifier-spec]                                    ┐
                 │                                                            │
                 │          { LITERALLY  }                                    │
                 │          { DEBUGVAR   }                                    │
                 │ [DEBUG]  { PROC       } [identifier-spec]                  │
                 │          { data-type  }                                    │
                 │          { POINTER    }                                    │
                 │                                                            │
                 │ { PARAMETER }                                              │
                 │ { KEYWORD   }                                              │
                 │ { TOOLVAR   } [identifier-spec]                            │
                 │ { TOOLPTR   }                                              │
DIR [FULL]       │ SYMBOLFILE [dir-name] [filename]                           │
                 │                                                            │
                 │ { TOOL         }                                           │
                 │ { KEYWORDGROUP } [wild-obj-identifier]                     │
                 │ { SYMBOLGROUP  }                                           │
                 │                                                            │
                 │ { PUBLIC    }                                              │
                 │ { SYMBOL    } [object] [identifier-spec]                   │
                 │ { MODSYMBOL }                                              │
                 │                                                            │
                 │ MODULE [identifier-spec]                                   │
                 │ LINE [dir-name] [line-number [,...] ]                      │
                 │ :module-name                                               │
                 └                                                            ┘
```

```
identifier-spec ::=[identifier-spec [,...] ]
```

Where (alphabetical):

data-type — limits the display to defined debug variables of a particular *data-type*. (See the <data-type> entry.)

DIR (continued)

DEBUG	displays the names of all the defined debug object. Limit the display by specifying what type of debug object you want to see.
DEBUGVAR	limits the display to the names of defined debug variables.
DIR	if entered without an option, displays the symbolics for the current module as determined by NAMESCOPE.
dir-name	specifies an emulator object directory name, for example; LOCAL//, GLOBAL//, and TOOL//, or in some cases a sub-directory such as SYMBOLFILE//.
filename	specifies the host development system path and file name where the symbol file is located.
FULL	expands complex types into their component scalar types. With the FULL option specified, name and type information is displayed for the sub-fields within a structure.
KEYWORD	displays the names of the predefined keywords (such as DIR, GO, and LOAD).
KEYWORDGROUP	displays the name of the file that contains the keywords and their grammars.
LINE	displays line numbers for the module defined by NAMESCOPE.
line-number	specifies a line number in the current symbol file.
LITERALLY	limits the display to the names of defined LITERALLYs.
MODSYMBOL	displays the names of all the symbols in the current module defined by NAMESCOPE.
MODULE	displays the names of all modules in the current symbol file.
module-name	specifies the names of the modules in the current symbol file.

object	is one of the following: <data type>, PROCEDURE, BLOCK, LABEL, FILE, ARRAY, ENUMERATION, SET, SCALAR, SUBRANGE, VARIANT, CONSTANT. Only symbols of the specified object type are displayed.
PARAMETER	displays the names of the predefined parameters (such as BASE, NAMEFRAME).
POINTER	limits the display to all defined memory pointers.
PROC	limits the display to the names of all the defined debug procedures.
PUBLIC	displays symbols in the current symbol file with the PUBLICS attribute for all modules. If *object* is specified, only symbols of that type are displayed.
SYMBOL	displays the names of all program symbols in the current NAMESCOPE.
SYMBOLFILE	displays the name of the file that contains symbolic information contained in the loaded OMF file of a user program.
SYMBOLGROUP	displays the name of the current symbol group.
TOOL	displays the name of the file that is defined as the tool (for example, ICE).
TOOLPTR	displays all of the predefined tool pointers (such as LPOINTER, PHY, OFFSET).
TOOLVAR	displays the names of the predefined tool variables (such as AX, FLAGS, IP).
wild-obj-identifier	specifies the name or identification of a particular object. Use single periods to separate the object path components (for example, proc1.proc1_b.var1). The use of the * and ? for wild characters is supported.

Discussion

Use the DIR command to display a list of debug objects, user program symbols, tool-specific objects, emulator directories, or pre-defined objects.

Examples

1. The following example lists all of the LITERALLY definitions currently defined.

```
hlt> DIR LITERALLY
LOCAL//
   len      literally     "LENGTH"
   lit      literally     "LITERALLY"
hlt>
```

2. List all of the keywords in the local directory.

```
hlt> DIR KEYWORD
LOCAL//KEYWORDGROUP//top
       BREAK
       CASE
       .
       .
       .
hlt>
```

3. Display all the module symbols in the current symbolfile. This command is useful when the NAMESCOPE is invalid.

```
hlt> DIR MODSYMBOL SYMBOLFILE//
TOOL//SYMBOLGROUPLINK//ICE.SYMBOLFILE//"plmsym.sym":MAIN
     VAL1          ord1
     VAL2          ord4
     PROC1         near32 proc
        P1         dynamic ord1
   .
   .
   .
hlt>
```

4. Display the symbols for the module main_mod.

```
hlt> DIR MODSYMBOL :main_mod
:main_mod
   VAR1   byte
   VAR2   byte
   .
hlt>
```

5. Display the line numbers for the module mod_bld.

```
     DIR LINE :mod_bld
:mod_bld
    7
   10
   11
   12
   .
hlt>
```

Cross-References

<data type>
DEFINE
NAMEFRAME
NAMESCOPE
Object Hierarchy
POINTER
PROC
REDEFINE
REMOVE
PUT/APPEND
SHOW
Symbolic References

DISPLAYFLAG

Controls object display
in assignment statements

Syntax

```
DISPLAYFLAG [= expr]
```

Where:

DISPLAYFLAG	if entered without an option, displays the current setting.
expr	specifies a number or an expression that must evaluate to TRUE (non-zero) or FALSE (zero). If *expr* is TRUE, the results of assignment operations are displayed. The default is FALSE.

Discussion

Use the DISPLAYFLAG parameter to control whether or not the value resulting after an assignment operation is displayed.

Example

The following example demonstrates the effect of DISPLAYFLAG.

```
hlt> DISPLAYFLAG
FALSE
hlt> DEFINE BYTE a
hlt> a = 3                                  /* result is not displayed */
```

```
hlt> DISPLAYFLAG = 1                    /* result (1H) is displayed */
0000000000000001H
hlt> a = 3
0000000000000003H
hlt> DISPLAYFLAG = 0                    /* result is not displayed */
hlt> a = 5
hlt>
```

Cross-References

<expr>
Parameters

DO ... WHILE

Groups and conditionally executes commands

Syntax

```
DO
      { commands }
WHILE (expr)
```

Where:

DO	specifies the beginning of a command block.
{*commands*}	specifies one or more emulator commands. The braces are required punctuation when multiple commands are entered.
WHILE (*expr*)	specifies that the loop ends if *expr* is FALSE. *Expr* specifies a number or an expression that must evaluate to TRUE (non-zero) or FALSE (zero). The parentheses are required punctuation.

Discussion

Use the DO ... WHILE statement to define a loop which is executed at least once. The test for continued execution (evaluation of *expr*) comes after the command (or group of commands) is executed. Always enclose the loop body {*commands*} in braces when there is more than one command. The commands are re-executed while the expression evaluates to TRUE. The INCLUDE command is not executable inside the DO ... WHILE statement.

Example

This example displays the upper-case alphabetic characters.

```
hlt> BASE = 16T
hlt> DEFINE INT4 a = 41H
hlt> DEFINE CHAR c
hlt> DO
hlt>> {
hlt>> c = TOASCII (a)
hlt>> c
hlt>> a += 1
hlt>> }
hlt>> WHILE (a <= 5AH)
'A'
'B'
.
.
.
'Y'
'Z'
```

Cross-References

BREAK
CONTINUE
DO_
<expr>
FOR
IF ... ELSE
IF_
PROC
SWITCH
WHILE

DO_

Groups and executes commands

Syntax

```
DO_
      commands
END
```

Where:

DO_	executes one or more commands in a block. The underscore is required punctuation.
commands	specifies one or more valid emulator commands.
END	specifies the end of the DO_ block.

Discussion

The DO_ ... END statement is used to signify that one or more commands are entered in a command block. Unless the DO_ ... END block is inside a PROC definition, execution of the commands occurs immediately after the END statement is typed. (See the DO ... WHILE entry.) The DO_ ... END command is like the PL/M or Pascal DO command.

NOTE

The DO_ ... END statement is an I^2ICE-compatible command. Therefore, the non-I^2ICE-compatible commands (WHILE, IF, FOR, DO ... WHILE, SWITCH, and INCLUDE) are not executable within the DO_ ... END statement.

Example

DEFINE a PROC to calculate the average of three numbers. The PROC is automatically removed when the DO_ block terminates.

```
hlt> DO_
hlt>> DEFINE PROC avg (a,b,c)
hlt>> DEFINE INT4 a
hlt>> DEFINE INT4 b
hlt>> DEFINE INT4 c
hlt>> {
hlt>> RETURN (a + b + c) / 3
hlt>> }                                          /* end of PROC avg */
hlt>> avg (4,2,4)              /* execute avg with four sets of values */
hlt>> avg (4,2,1)
hlt>> avg (3,2,4)
hlt>> avg (2,2,5)
hlt>> END                                   /* end of DO_ ... END block */
0000000000000003H
0000000000000002H
0000000000000003H
0000000000000003H
hlt>
```

Cross-References

DO ... WHILE
<expr>
FOR
IF ... ELSE
IF_
PROC

EDIT

Invokes a disk-resident editor

Syntax

```
EDIT [USING "invocation string"] PROC proc-name
```

Where:

EDIT	uses the specified or default editor to create or modify user-defined procedures (PROCs).
USING	indicates a user-specified editor name follows.
"*invocation string*"	specifies the path and invocation name of an editor on the host development system. The quotation marks are required punctuation.
PROC	indicates a user-defined procedure name follows.
proc-name	specifies the unique name for the procedure to be created or edited.

Discussion

Use the EDIT command to create or revise a debug procedure. The EDIT command signals the emulator software that an editor is requested for creating or modifying a procedure (PROC). If you are creating a PROC interactively, when the emulator syntax checker comes to an error, the entire PROC is removed (erased from the emulator database). A PROC created with an editor is saved in the emulator database even if it contains a mistake.

After a PROC is created using EDIT, and the editor's exit command is issued, the emulator software interprets the PROC and checks for errors. When an error is encountered, it is flagged, but the original text is saved for re-edit. Use the EDIT command to re-edit the PROC and correct the mistakes found by the emulator.

To choose an editor other than the default, make sure the editor resides on the host development system, then specify the USING option with the EDIT command. The name of the default editor is

contained in the command language parameter EDITOR. The default value of EDITOR is "EDLIN". An empty PROC is created even if the editor is aborted.

NOTE

Depending on the configuration of the host development system and the editor size, there may be memory limitations when using the EDIT command. Review the editor's memory requirements carefully before choosing to use it with the emulator. You can also use shell escape (@) to access another editor.

Examples

1. The following example shows how to edit a PROC with EDLIN as the editor.

```
hlt> EDIT PROC menux
New file
*i
          1 :* /* edit file */
          2 :*
         16 :* ^c
*w
              /* edit file */
*e
hlt>
```

2. The following example shows how to select an alternate editor (iEDIT) with the USING option.

```
hlt> EDIT USING "C:IEDIT" PROC menux
/* edit file using IEDIT */
hlt>
```

Cross-References

DEFINE	PROC
EDITOR	REDEFINE

EDITOR

Specifies which editor
is invoked by EDIT

Syntax

```
EDITOR [= "expr"]
```

Where:

EDITOR	if entered without an option, displays the current setting.
expr	specifies the invocation string (path and invocation name) of an editor available on the host development system. The default is "EDLIN". Quotation marks are required punctuation to delimit the invocation string.

Discussion

Use the EDITOR parameter to specify which editor is invoked when the EDIT command is executed. The editor can also be specified using the installation procedure.

Examples

1. Display the current setting of EDITOR.

```
hlt> EDITOR
"edlin"
hlt>
```

2. Specify "IEDIT" as the default editor.

```
hlt> EDITOR = "IEDIT"
hlt>
```

Cross-References

EDIT
Parameters

ENI

Controls the status of interrupt servicing during Halt mode

Syntax

```
ENI  [= expr [address] [INTRPT5 mask] ]
```

Where:

ENI	is the command keyword. If entered without parameters, it displays the current ENI setting and address.
expr	is any expression that evaluates to 0, 1, or 2.
address	is the starting location in mapped memory of a 2-byte block of contiguous RAM. The emulator software uses this memory block to handle user interrupts.
INTRPT5	is a keyword indicating that the following mask value is to be written to the INTRPT5 register. This parameter is valid only when ENI=1.
mask	is a value written to the INTRPT5 register. If this parameter is not used, the default mask value is assumed. This parameter is valid only when ENI=1.

Default

ENI defaults to 0.
address has no default.
mask defaults to 00F0H in Master (non-iRMX™) mode or to 0000H in Slave (iRMX) mode.

Discussion

The ENI tool variable allows you to specify the extent of interrupt servicing in Halt mode. You can specify:

- ENI=0 -- the emulator does not service interrupts during Halt mode.

- ENI=1 -- the emulator services internal processor interrupts during Halt mode, but not external interrupts, depending on the mask value specified in the INTRPT5 register. (The emulator never services NMIs when ENI is set to 1.) You can modify the mask value to allow or mask out specific interrupts.
- ENI=2 -- the emulator services both internal processor and external user interrupts during Halt mode.

You can display the current ENI setting at any time, but can change it only when the emulator is in Halt mode.

If you set ENI to 0 while the emulator is halted, the change in interrupt servicing takes effect immediately. If you set ENI to 1 or 2 while the emulator is halted, the change in interrupt servicing takes effect only after the next emulator halt.

The following paragraphs describe the three ENI settings and their effects on interrupt servicing in Halt mode. (Interrupts are always serviced in Run mode.)

NOTE

Before using the ENI=1 or ENI=2 setting, ensure the following:

1. You have mapped a 2-byte block of contiguous RAM as readable and writable.
2. You have thoroughly debugged your interrupt routines.

Write violations or emulator time-outs will disrupt interrupt servicing (see the Effects of Emulator Time-Outs later in this section).

With the ENI=0 setting, the emulator does not service any interrupts in Halt mode. It marks interrupts as pending and services them once emulation is re-started. The /RES signal from the prototype is ignored.

Certain Halt mode functions are maintained with ENI=0. They include:

- DMA transfers
- Bus HOLD/HLDA
- Refresh cycles (80C186 Enhanced operation mode only)

These Halt mode functions are always maintained, regardless of the setting of ENI.

The contents of the INTRPT5 register are not affected by the ENI=0 setting.

The Halt mode prompt for ENI=0 is "hlt>".

With the ENI=1 setting, the emulator performs the functions listed for ENI=0, plus it services internal processor interrupts (except during mapped I/O requests and synchronous emulator start-ups). The /RES signal from the prototype is ignored and NMIs are not serviced.

The INTRPT5 register is modified when ENI is set to 1. The default value of the INTRPT5 register depends on the mode of the interrupt controller section of the processor Peripheral Control Block. In Master (non-iRMX) mode, the default mask value is 00F0H, which masks out external interrupts and allows internal interrupts. In Slave (iRMX) mode, the default value is 0000H, which allows internal interrupts. In Slave mode, external interrupts other than NMIs are not maskable unless you mask them in your external interrupt controller.

You can enter a new mask value whenever you set ENI to 1. If you do not provide a mask value when setting ENI to 1, the default mask value is assumed.

If you enter a new mask value while the emulator is halted with ENI=1, the change in interrupt masking takes effect immediately. Writes to the INTRPT5 register (via a register access command or an interrupt routine) may also dynamically change the interrupt masking.

The Halt mode prompt for ENI=1 is "idi" (Interrupts During Interrogation).

ENI (continued)

With the ENI=2 setting, the emulator services both internal processor and external user interrupts (except during mapped I/O requests and synchronous emulator start ups). The /RES signal from the prototype is acknowledged and no interrupts are masked out.

The contents of the INTRPT5 register are not affected by the ENI=2 setting.

The Halt mode prompt for ENI=2 is "idi>" (Interrupts During Interrogation).

When you set ENI to 1 or 2 the first time, you must enter the address parameter. This parameter specifies the starting address of a contiguous 2-byte block of mapped memory reserved for interrupt handling. You can display the current ENI setting and address information at any time by entering the command without parameters.

Always debug your interrupt routines before attempting to use the ENI=1 or ENI=2 setting during Halt mode. A read-only violation or emulator time-out will disrupt interrupt servicing.

Read-only violations occur when the program attempts to write to read-only memory. Emulator time-outs occur under any of the following conditions:

- The processor takes more than one second to complete a memory access cycle (MEMRDY=TRUE).
- The processor takes more than one second to complete an I/O access cycle (IORDY=TRUE).
- The processor bus is idle for more than one second (BUSACT=TRUE).

If a read-only violation or time-out occurs while the emulator is halted with ENI=1 or ENI=2 (the Halt mode prompt is "idi>"), the emulator automatically disables interrupt servicing. The ENI variable remains at its current setting, but the emulator ignores any further interrupts, and the Halt mode prompt changes to "hlt>" the next time you enter a command. Interrupt servicing is re-enabled when you start the emulator.

You can disable time-outs by setting the MEMRDY, IORDY, and BUSACT variables to FALSE. If time-outs are disabled, however, the emulator cannot perform a successful break request while an interrupt is in progress. It can perform break requests only while an interrupt is not in progress. If an unsuccessful break request occurs, the emulator displays an error message the next time you enter a command that requires a break (e.g., a memory access command). You must reset the emulator by entering the RESET ICE command or by DEACTIVATEing then re-ACTIVATEing the emulator hardware.

To avoid unsuccessful breaks, make sure your interrupt routines end with an IRET instruction. Also, debug your routines to make sure they do not cause read-only violations, access ICE-mapped I/O ports, or cause memory and I/O access delays.

Example

This example illustrates setting up a section of memory for interrupt handling at ENI level 2. Then the ENI feature is turned off (ENI is set to 0).

```
hlt> ENI = 2 1000:2FF0H
hlt> GO
 Use S (status) command to determine emulation status.
emu+> HALT
 Emulator Status:  halted (servicing internal CPU and user interrupts
                                  at 1000:2ff0H).
  EXECPOINTER is :MESSG.DELAY_A_LITTLE_BIT_#163 + aH   (1000:0058H)
 Activity Monitor Status:  Stopped.
idi> ENI = 0
hlt> GO
 Use S (status) command to determine emulation status.
emu+> HALT
 Emulator Status:  halted (not servicing interrupts).
  EXECPOINTER is :MESSG2.DELAY_A_LITTLE_BIT_#163 + eH   (1000:005cH)
 Activity Monitor Status:  Stopped.
hlt>
```

ENI (continued)
Cross-References

HALT
PCB
Prompts
Register Access
TOOLVAR

ERROR

Controls the display of error information

Syntax

ERROR [= *expr*]

Where:

ERROR — if entered without expr, displays the current setting.

expr — specifies an expression that must evaluate to one of the following:

- 0 Displays only minimal error message information. The default is 0.
- 1 Displays internal error information.

NOTE

The ERROR parameter determines how much error information is displayed when an error occurs. There are three levels of error information: minimal, internal, and extended. If ERROR is set to 0 (zero), only a minimal amount of error information is displayed; the error number and what the error was. If ERROR is set to 1, all internal error information is displayed, usually indicating the error handling actions taken by the emulator software. Extended error information (if available) is displayed only by using the HELP command. Error messages with extended help contain the following symbol at the end of the message: [*].

ERROR (continued)

Example

Set the ERROR parameter to display standard information and cause an error. Then reset the ERROR parameter to minimal display and cause the same error. Then display the extended help information for the error message.

```
hlt> ERROR = 1
hlt> PUT \non_exist\directory\myfile DEBUG
[kernel_cmd] Warning #6t
Unsuccessful PUT.  Database information not saved.
[dbm] ERROR #-300t
Error in opening file. [*]
hlt> ERROR = 0
hlt> PUT \non_exist\directory\myfile DEBUG
[dbm] ERROR #-300t
Error in opening file. [*]
hlt> HELP ERR dbm -300T
```

Cross-References

HELP
Parameters

EVAL

Evaluates an address or an expression to its module, procedure, line, or symbol reference

Syntax

```
        ⎧ addr        ⎧ LINE      ⎫  ⎫
        ⎪             ⎨ MODULE    ⎬  ⎪
EVAL    ⎨             ⎪ PROCEDURE ⎪  ⎬
        ⎪             ⎩ SYMBOL    ⎭  ⎪
        ⎪                            ⎪
        ⎩ EXPR expr [data-type addr] ⎭
```

Where:

EVAL	matches the specified address to the closest approximate symbolic reference in the current symbol file, or if EXPR is specified, displays the specified expression in all the supported number bases and ASCII.
addr	specifies a number or an expression that evaluates to an address.
LINE	evaluates *addr* as a line-number reference. The resulting message is in the form: `:module_name[.procedure_name]#line_number[+offset]`
MODULE	evaluates *addr* as a module reference. The resulting message is in the form: `:module_name[+offset]`
PROCEDURE	evaluates *addr* as a procedure reference. The resulting message is in the form: `:module_name.proc_name[+offset]`
SYMBOL	evaluates *addr* as a symbolic reference (label or variable). The resulting message is in the form: `:module_name.proc_name.symbol_name[+offset]`

EVAL (continued)

EXPR	specifies that *expr* is to be displayed in hexadecimal, decimal, binary, and ASCII equivalents. If the *expr* is of type REAL, NSTRING, or LSTRING, the display shows the *expr* in hexadecimal. If the *expr* is of type BOOLEAN, the display shows TRUE or FALSE. If the *expr* is of type BCD, the display shows *expr* in decimal only.
expr	specifies a number or an expression to be displayed; can also be of format <*datatype*> address.
datatype	specifies the data at the address is displayed in the *datatype* format.

Discussion

Use the EVAL *addr* command to evaluate an address to its module, procedure, line number, or symbol reference in the current symbol file. Only the first match is displayed. If a segmented address is specified with only the offset, and the SYMBOL option is selected, the default is the data segment (DS). If a segmented address is specified with only the offset, and the LINE, MODULE, or PROCEDURE option is selected, the code segment (CS) is the default.

If the address expression does not evaluate to an exact match, the closest symbolic reference with an address lower than the specified address is displayed with a byte offset. The offset is displayed in the current base.

Use the EVAL EXPR *expr* command to display a representation of an expression in all of the supported bases and ASCII.

Examples

1. The following examples uses the MODULE option to display the current execution point.

```
hlt> EVAL $ MODULE
:module_a + 25
hlt>
```

2. The following example uses the LINE option to display the module and line number of an address. The current code segment (CS) is assumed.

```
hlt> EVAL 0F70H LINE
:module_main.proc_1#29
hlt>
```

3. The following example uses the PROCEDURE option.

```
hlt> EVAL :module_main#45 PROCEDURE
:module_main.set_scan + 24
hlt>
```

4. The following example uses the SYMBOL option. The data segment (DS) is assumed.

```
hlt> EVAL 0200:0001+20 SYMBOL
:module_main.big_array + 015
hlt>
```

5. The following example uses the EXPR option.

```
hlt> EVAL EXPR 41H
00000000000000000000000000000000000000000000000000000000001000001Y
00000000000000000101Q 65T 0000000000000041H "A......."
hlt>
```

6. If an option is selected without having a reference, an error message is displayed.

```
hlt> EVAL 0176:7830 SYMBOL

[Kernel_cmd] Warning #42t
Unsuccessful EVAL.
[dbm] Error #-28t
Symbolic match for specified pointer not found in symbolfile.
```

7. The following example displays the contents of a byte at address 1000:0000.

```
hlt> EVAL EXPR ORD1 1000:0
01010110Y 126Q 86T 56H "V"
```

EVAL (continued)
Cross-References

Address Translation
DIR
<expr>
Symbolic References

Syntax

```
EXIT  [ [OVERWRITE] filename ]
```

Where:

EXIT	terminates the debug session and returns control to the host operating system.
filename	specifies the name of the disk file where the entire binary representation of the debug environment is stored (see the PUT entry).
OVERWRITE	replaces the specified file (*filename*) if it exists. If OVERWRITE is not specified, and the file exists, a message is displayed, the debug session is not terminated, and the data base is not saved.

Discussion

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

Saving the Data Base

Using the *filename* option specifies the path and filename of the disk file where the binary representation of the emulator database is saved. The database includes all defined debug objects: LITERALLYs, debug variables, and PROCs, as well as a TOOL definition. Parameter values are also saved.

Restoring the Data Base

Restore a saved file by invoking the emulator with the -r filename option or by specifying a data base file during the installation procedure (see the Invocation and PUT entries).

EXIT (continued)

When restoring the data base file at invocation time it is possible to receive an error message (e.g., Name already exists). Ignore this error message. The restoration process attempts to restore an environment definition defined by default during the initialization process.

When a debug PROC is defined, a temporary file is used to store information about the PROC. The directory where this temporary file is stored depends on the designation of the host operating system TMP or TEMP environment parameters. The full path name for the temporary file is recorded in the data base. When a binary representation of the data base is saved with either the EXIT command or the PUT command, this full path name is also saved. If the database is restored on a system that does not contain this path, attempts to execute or read the contents of the PROC results in an error.

The DEACTIVATE command can be entered before the EXIT command to enable you to reinvoke the emulator without first resetting the emulator controller unit.

Example

The following example exits the session and saves the debug environment (emulator data base).

```
hlt> EXIT c:\dbug
DATABASE SAVED ON C:\DBUG

goodbye!
C>
```

Cross-References

Chapter 3, Sections 3.3.1 and 3.3.2
DEACTIVATE
DEFINE
Invocation
PUT

<expr>

One or more numbers, debug objects, or functions separated by operators

Syntax

```
{ [left-unary-operator] operand          }
{                                        } [option-tail]
{ operand [right-unary-operator]         }
```

```
option-tail::=

{ binary-operator }  { [left-unary-operator] operand          }
{                }  {                                        } [...]
{ assign-operator }  { operand [right-unary-operator]         }
```

Where:

left-unary-operator	acts on a single operand (Table 6-6 defines left unary operators).
operand	specifies one of the following: • a number • a quoted string (e.g., "string") • the name of a user-defined debug object • a built-in or user-defined function (see the Functions section and the PROC command in this chapter) • a memory access operation using data types (see the Memory Access section of this chapter) Some operands are user-defined; others are system-defined.
right-unary-operator	acts on a single operand (Table 6-6 defines right unary operators).

assign-operator	acts on two operands. The result is a single operand (Table 6-6 defines assignment operators).

Discussion

An expression is a single value or a combination of operands (one or more numbers, variables, strings, or control variables) separated by operators. Evaluating an expression applies the operators to the operands until a single result is obtained. When the *<expr>* entry appears on the syntax guide, enter an expression appropriate for the command.

An expression entered as a command (not as a part of a command) is evaluated directly and the result is displayed in the current base.

The contents of a variable or a memory location can be used in an expression.

Operands

The three classes of operands are constants, debug variables, and functions. Within each class, some operands are user-defined and others are built-in. An expression can be a single operand without any operators.

Constants do not change value during execution. Constants can be any supported emulator data type. User-defined constants include ordinals, integers, reals, and strings. These constants contain one or more valid digits and (optionally) a suffix character indicating the number base. Without the suffix, digits are interpreted according to the current setting of BASE (see the BASE entry).

To avoid confusion with debug objects and symbolic references, a number must not begin with a letter as the first digit. For this reason, a hexadecimal number may require a leading zero. For example, 0AH is valid whereas AH is considered to be the AH register.

Integers of the form nK (Kilo-byte) and nM (Mega-byte) are valid constants, where n is an unsigned decimal integer, K is 1024 decimal, and M is 1048576 decimal.

Debug variables are user-defined variables of a particular data type.

Functions can be referenced within an expression. The return value of the function is assigned to the expression in the return data type of the function. Functions are user-defined PROCs and built-in, emulator supplied functions (see the PROC command and the Functions section in this chapter).

Operators

Table 6-6 lists the operators supported by the emulator.

Table 6-6 Emulator Operators

Category	Symbol	Function
left- unary	*	indirection
	&	address of
	–	unary minus
	!	logical NOT
	~	bitwise NOT *
	+ +	pre-increment
	—	pre-decrement
right- unary	+ +	post-increment
	—	post-decrement
pointer	:	separates pointer components

Table 6-6 Emulator Operators (continued)

Category	Symbol	Function
binary	*	multiplication
	/	division
	%	modulus (MOD*)
	+	addition
	–	subtraction
	< <	shift left
	> >	shift right
	<	less than
	>	greater than
	< =	less than or equal
	> =	greater than or equal
	= =	equivalence
	!= (or < >)	non-equivalence
	&	bitwise AND *
	|	bitwise OR *
	^	bitwise XOR *
	&&	logical AND
	^^	logical XOR
	||	logical OR
primary	( )	
	[ ]	
ternary	? :	three-element conditional expression (for example: (a>b)?(a):(b) displays the greater value)

Table 6-6 Emulator Operators (continued)

Category	Symbol	Function
assignment	=	simple assignment
	+=	implied operand addition
	-=	implied operand subtraction
	*=	implied operand multiplication
	/=	implied operand division
	%=	implied operand modulus
	>>=	implied operand right shift
	<<=	implied operand left shift
	&=	implied operand bitwise AND
	^=	implied operand bitwise XOR
	|=	implied operand OR

* AND, MOD, NOT, OR, XOR are available for I²ICE compatibility

Table 6-7 lists operator order of precedence and how evaluation occurs. Precedence is determined similar to the C programming language. Expressions containing the logical operators && and || evaluate left-to-right and terminate as soon as a result can be determined. This limit means that under some circumstances, the right side of the operation may not be evaluated. For example, the left side of an && operation is evaluated. If the result is zero, the right side is not evaluated.

Table 6-7 Order of Precedence

PRIMARY OPERATORS [left-associative]

```
()      []
```

UNARY OPERATORS [right-associative]

```
*      &      -      !      ~      ++      --
```

BINARY OPERATORS [left-associative]

```
*       /       %
+       -
>>        <<
>       <       >=       <=
==       !=       <>
&
^
|
&&
^^
||
```

TERNARY OPERATOR (three-element conditional expression) [right-associative]

```
?:
```

ASSIGNMENT OPERATORS [right-associative]

```
=       +=       -=       *=       /=       %=       >>=       <<=
&=      ^=       |=
```

Examples

1. The following example displays the results of expressions.

```
hlt> BASE = 10T                              /* set number base to decimal */
hlt> 5 * 5                                          /* binary operation */
25T
hlt> 5 * (6/2)                          /* binary and primary operations */
15T
hlt>
hlt> (3>6) ? (3) : (6)                              /* ternary operation */
6T
hlt>
hlt> BASE = 2T                               /* set number base to binary */
hlt> DEFINE BYTE i = 1010
hlt> <<=i                               /* binary operation (shift left) */
hlt> i
0101Y
hlt> ++i ; i                      /* left unary operation (pre-increment) */
0110Y
hlt>
```

2. The following example uses an expression in an emulator command. Increment the execution pointer ($) by the fourth power of two using binary addition. Then disassemble code from that execution point using the binary modulus of a number to specify the range.

```
hlt> $
1000:0001H
hlt> $ = $ + POW(2,4)              /* POW is an emulator math function */
hlt> $
1000:0011H
hlt> ASM $ LENGTH 347T%2                  /* % is the modulus operator */
/* ...display... */
hlt>
```

Cross-References

C: A Reference Manual
<data type>
Memory Access

FASTBREAK

Interupt and resume emulation
to collect specific system data.

Syntax

FASTBREAK [= *bool-expr*]

Where:

FASTBREAK	if entered without an optional expression, displays the current FASTBREAK setting.
bool-expr	is any expression that evaluates to TRUE (non-zero) or FALSE (zero).

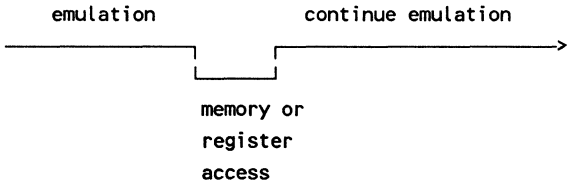
Default

FALSE

Discussion

The fastbreak feature allows the emulation processor to stop emulation, perform the required action (such as a memory or register access), then return to emulation as quickly as possible. You enable fastbreaks by setting the variable to TRUE; you disable them by setting the variable to FALSE. You can change the setting of the variable only when the emulator is in Halt mode.

The following diagram illustrates how a fastbreak looks to the emulation processor:



This feature is primarily focused at embedded applications which need to emulate in real-time without halting the emulation processor for extended periods of time.

Fastbreak emulation is actually real-time emulation with a provision for emulation breaks of 5625 CPU clock cycles or less. If fastbreaks are enabled, you can interrupt emulation to execute commands, such as REGS or PCB, that require access to processor registers or memory. You can execute these commands from the keyboard while emulation is in process or as part of a GO command entry (see GO in this chapter).

If execution of a command requires more than 5625 CPU clock cycles, the emulator performs sequential fastbreaks until the command is completed. After each fastbreak, the emulator automatically resumes emulation.

If fastbreaks are disabled, you cannot enter any commands that access processor memory or registers while emulation is process. In addition, you cannot use the MEM, REGS, and PCB options as part of a GO command entry. The emulator will only stop if forced to do so by a HALT command, a memory access violation, a time-out (I/O, memory access, or bus activity), or an emulation break.

The setting of FASTBREAK determines which prompt is displayed during Run mode. If FASTBREAK is set to FALSE, the prompt is "emu>" or "emu+>". If FASTBREAK is set to TRUE, the prompt is "fst>" or "fst+>". The "+" indicates that the Activity Monitor is running.

Note that during fastbreaks, the emulator supports the following processor functions:

- DMA transfers
- Refresh cycles
- Bus HOLD/HLDA

Interrupts that occur during fastbreaks are noted as pending by the emulator and are serviced at the completion of the fastbreak.

There are two types of fastbreaks: synchronous and asynchronous. These differ in how they are initiated, how they affect the emulation processor, and how much time they require to perform a specified task. The primary difference is whether the fastbreak is issued as an event in the GO command or it is issued at the command line. The characteristics of these two types are summarized as follows:

Synchronous Fastbreak:

- Occurs when memory access and register commands are synchronized with an event in the GO command. Synchronous fastbreaks can only be requested using the GO command.
- Allows control of the time between fastbreaks as specified in the GO command. The time between fastbreaks is dependent upon the position of each recognized event (which causes the fastbreak action) that occurs during emulation.
- Ensures that the processor is not out of emulation for more than 5625 clocks. The FASTBREAK command is truncated if it exceeds this limit. For example, in a command to read 1K of memory, only 40T bytes will be read during the first 5625 clocks; no other bytes will be retrieved.
- Does not generate multiple fastbreaks to complete a command.
- Stores the requested information in the trace buffer, which provides contextual information about that fastbreak.

Asynchronous Fastbreak:

- Occurs when memory access or register commands are issued at the command line (at the fst> prompt when ACTIVITY MONITOR is off, or at the fst+> prompt when ACTIVITY MONITOR is on), for example, REGS (including individual registers), PCB (including individual registers), and FLAGS.
- Occurs when commands that require memory access or address conversion are issued at the command line, for example, EVAL, PORT, WPORT, ORDn, or ASM.
- Does not allow control of the time between fastbreaks as they are entered at the command line. Specifically, the time between multiple fastbreaks required to implement a single command is **very** small. Thus, asynchronous fastbreaks can be detrimental in real-time applications because of performance degradation. The amount of time it takes to execute the asynchronous fastbreak disrupts the execution of the application program.
- Requests the processor to interpret the command as a sequence of fastbreaks, none of which will exceed 5625 clocks.

- Displays the requested information on the screen (without any contextual information).
- Provides limited information because the context of the information may be unknown. For example, the value of a variable is seldom useful if you do not know where the program counter is and what has been executed. However, it may be useful to check the stack pointer for an overflow condition.
- May cause multiple fastbreaks to occur. For example, commands that use addresses may generate a fastbreak to convert the address. Thus, the fastbreak interval for commands that cause multiple fastbreaks (such as ORD1 CS:IP LEN 1000H) cannot be calculated.

While the emulator provides very extensive fastbreak capabilities, explicitly specifying a maximum amount of time that the emulation processor may be out of emulation is not possible in these emulators. Although fastbreaks will not exceed 5625 clocks (450 microseconds at 12.5 MHz with 0 wait state memory), your application may require that the processor return to emulation before this time limit is reached. Section 1.4 contains recommendations to reduce the amount of time your processor is out of emulation. The following section provides timing information to help you calculate how long the processor will be out of emulation to perform specific tasks, depending on whether you use a synchronous fastbreak or an asynchronous fastbreak.

The Tool Variable: FASTBREAK

Globally, a fastbreak is controlled by the FASTBREAK tool variable. If FASTBREAK is FALSE, any command (other than HALT) entered during emulation, or with the GO command, which requires the emulation processor to stop will not be performed and an error will be generated.

Calculating Execution Time

The timing tables in this section can be used to estimate how much time is required for each type of fastbreak. The time between synchronous fastbreaks **can** be controlled, but the time between asynchronous fastbreaks **can not**.

Timing information for the ICE-186 emulator is shown in Table 6-8, and information for the ICE-188 emulator is in Table 6-9. The timings for the ICE-188 emulator are approximately 30% slower than the ICE-186 emulator due to the narrower bus. Both tables contain information for synchronous fastbreaks first, followed by information for asynchronous fastbreaks. There are four major columns in the tables: Fastbreak Type, Max Bytes, Time in Microseconds (which has three subheadings), and Fastbreak Interval. Each column (including the three subheadings) denotes the following:

Fastbreak Type	is the action that the fastbreak performs.
Max Bytes	are the maximum number of bytes that can be accessed by a single fastbreak.
Time in microseconds	
1 Byte:	is the number of microseconds required to perform a fastbreak that accesses a single byte.
Max Bytes:	is the number of microseconds required to perform a fastbreak that accesses the maximum number of bytes allowed.
Delta/Byte	is the number of microseconds required to access each additional byte after the first one, up to the maximum.
Fastbreak Interval	is the minimum time required between fastbreaks when a large number of fastbreaks is necessary to complete execution of a single command.

Not all of the possible fastbreak types are included in the tables, but they may be characterized as follows: if the command accesses memory, time is approximately a few milliseconds. If the command accesses registers, time is approximately hundreds of milliseconds.

The last type in the tables, which writes to the Peripheral Control Block (PCB), occurs when asynchronous commands are entered. There are no rules governing timing for this situation. This is strictly an asynchronous event between two processors.

NOTE

The timing information was calculated by using a performance analyzer connected to the ICE-186 emulator. This information does not have a safeguard band and should not be taken as a guarantee. Use the timing as an estimate and provide a safety band corresponding to problems you may induce by violating timing on your system.

Table 6-8: Timing Information for the ICE™-186 Emulator

Fastbreak Type	Max Bytes	Time in Microseconds*			Fastbreak Interval
		1 Byte	Max Bytes	Delta/Byte	
Synchronous:					
memory trace	40	353.1	387.5	0.88	-
register trace	-	-	363.1	-	-
PCB trace	-	-	540.0**	-	-
Asynchronous:					
memory fill with verify	6	363.9	437.1	14.64	2,260
memory fill w/o verify	18	328.3	447.9	7.04	1,990
memory read	40	308.5	452.0	3.68	2,450
register read	-	-	377.2	-	-
register write	-	318.4	-	-	-
PCB read	-	-	432.2	-	-
PCB write	-	331.0	-	-	-

* For target running at 12.5 mHz with 0 wait status.
** Exceeds 5625 CPU clock cycles.

Table 6-9: Timing Information for the ICE™-188 Emulator

Fastbreak Type	Max Bytes	Time in Microseconds*			Fastbreak Interval
		1 Byte	Max Bytes	Delta/Byte	
Synchronous:					
memory trace	40	565.0	599.3	0.87	-
register trace	-	-	576.6	-	-
PCB trace	-	-	876.5**	-	-
Asynchronous:					
memory fill with verify	6	574.8	683.2	21.7	2,260
memory fill w/o verify	18	517.0	692.5	10.32	1,990
memory read	40	468.7	664.5	5.02	2,450
register read	-	-	581.4	-	-
register write	-	503.1	-	-	-
PCB read	-	-	709.7	-	-
PCB write	-	526.6	-	-	-

* For target running at 12.5 mHz with 0 wait status.

** Exceeds 5625 CPU clock cycles.

Recommendations:

To reduce the risk of fastbreaks being too long for your application, follow these guidelines:

1. Minimize the use of asynchronous fastbreaks.

2. Minimize the use of commands that require an application-specific number of bytes, for example, ASM.
3. When using synchronous fastbreaks (by specifying multiple fastbreak commands during a GO command), ensure that the interval between events is kept to a minimum.
4. Minimize the number of memory accesses.
5. If memory accesses are necessary, do not attempt to access large blocks of memory or registers.
6. In a real-time system, use synchronous fastbreaks only.

Examples

1. Issue a synchronous fastbreak that interrogates the registers immediately following the execution of the first instruction corresponding to line #23.

```
hlt> FASTBREAK = TRUE
hlt> GO TIL EXECUTION #23 REGS REPEAT
```

2. Issue a synchronous fastbreak to capture four bytes of memory into the trace buffer after an access (read or write) occurs at location 10044H or break if a FULLBUF occurs.

```
hlt> GO TRACE TIL BUS 10044HP MEM 200H LEN 4 REPEAT ORIF FULLBUF HALT
```

3. Issue a synchronous fastbreak to interrogate the registers.

```
hlt> GO TIL EXECUTION 1471:10H REGS REPEAT
```

4. Issue an asynchronous fastbreak to view memory locations while emulator is running.

```
fst+> byte 1000:0 len 4
1000:0000H 56  57  55  8B                "VWU."
```

Cross-References

<expr>
Flags
GO
Memory Access
PCB
Register Access
REGS
TOOLVAR

Flags

Displays or modifies the microprocessor flags register

Syntax

```
{ FLAGS  }
{ FL     } [= expr]
{ FH     }
{ flag   }
```

Where:

FLAGS	displays the current value of the 80C186 or 80C188 flags register (see Figure 6-2).
FL	displays the lower (least significant) byte of the 80C186 or 80C188 flags register.
FH	displays the upper (most significant) byte of the 80C186 or 80C188 flags register.
flag	displays the current value of a flag and is one of the keywords shown in Figure 6-2.
expr	is an expression (of the correct data type) used to set flag values.

Discussion

Display flag values by entering their individual keywords or by entering the keyword FLAGS. Flag values are displayed as Boolean values. The REGS command displays the flag mnemonic of all flags set to 1; if no flags are set, the word "none" is displayed.

Modify individual flags in one of two ways. Enter the word FLAGS ORed or ANDed with the proper bit pattern (only the least-significant 16 bits of the expression are used), or assign a Boolean value to the individual flag.

BIT	KEYWORD	DESCRIPTION	MEMORY TYPE
15	X	Don't care	
14	X	Don't care	
13	X	Don't care	
12	X	Don't care	
11	OFL	Overflow flag	BOOLEAN
10	DFL	Direction flag	BOOLEAN
9	IFL	Interrupt flag	BOOLEAN
8	TFL	Trap flag	BOOLEAN
7	SFL	Sign flag	BOOLEAN
6	ZFL	Zero flag	BOOLEAN
5	X	Don't care	
4	AFL	Auxiliary flag	BOOLEAN
3	X	Don't care	
2	PFL	Parity flag	BOOLEAN
1	X	Don't care	
0	CFL	Carry flag	BOOLEAN

Figure 6-2 80C186 Flags Register Bit Pattern

Flags (continued)

Examples

1. This example displays the current value of the flags register, and then modifies the register contents by ORing the current value with 1.

```
hlt> FLAGS
0002H
hlt> FLAGS = FLAGS OR 1
hlt> FLAGS
0003H
```

2. This example displays and modifies the trap flag (TFL).

```
hlt> TFL
0H
hlt> TFL = TRUE
hlt> TFL
1H
```

Cross-References

<expr>
FASTBREAK
Register Access
REGS

Syntax

FOR (*command1* ; *expr* ; *command2*) [{*commands*}]

Where:

FOR ( )	groups and executes commands in a loop construct. Parentheses are used to enclose the commands and expressions that control the loop.
command1	is usually an assignment statement, or makes a function call, but can be any valid emulator command. *Command1* is optional, but the semi-colon (;) must remain as a place holder.
;	(semicolon) separates statements and acts as a place holder. Use two semi-colons even if there are not three statements (e.g., FOR (; i<25 ; i++)).
expr	specifies the test condition. *Expr* must evaluate to TRUE (non-zero) or FALSE (zero). *Expr* is optional. However, if it is omitted, the test condition evaluates to TRUE, and the semi-colon (;) must remain as a place holder.
command2	is usually a re-assignment, an increment, or a function call, but can be any emulator command. *Command2* is optional.
{*commands*}	are one or more emulator commands that are executed when the test condition *expr* is TRUE. Braces ({ }) indicate the start and end of multiple commands controlled by the FOR statements. An empty {*commands*} specification, indicated by a semi-colon (;) is valid.

FOR ( ) (continued)

Discussion

The FOR statement is an iteration construct that causes the specified commands to be executed one or more times as long as *expr* evaluates to non-zero (TRUE) in the following order: *command1* is executed once; then if *expr* evaluates to TRUE, {*commands*} is executed. After that, *command2* is executed and *expr* is evaluated. This process is repeated as long as *expr* evaluates to TRUE.

Examples

1. The following FOR statement sets an index to zero, sets the test condition to the index being less than 10, and finally causes the index to be incremented and displayed. Assume "i" has been defined as data type BYTE. The last semi-colon is required (it represents an empty command list).

```
hlt> FOR (i = 0; i < 10 ; i++);
```

2. Execute the commands ISTEP and EVAL three times. First initialize the index to zero. The test condition is that the index must be less than 3. Then the index is incremented by one after each iteration.

```
hlt> FOR (i = 0; i < 3; i++)
hlt>> {
hlt>> ISTEP; EVAL $ LINE
hlt>> }
```

3. Increment the values of the BX, CX, and DX registers using the FOR statement with the AX register as a counter. Initialize the AX register to zero. The test condition is the value of the AX register being less than or equal to 25, and the AX register is incremented by one on each iteration. As long as the AX register is less than 25, the BX, CX, and DX registers are also incremented by one. The AX register is incremented after the BX, CX and DX registers are incremented.

```
hlt> FOR (AX = 0; AX <= 25 ; ++AX) { BX++; CX++; DX++; }
```

Cross-References

BREAK
CONTINUE
COUNT
IF .. ELSE
IF_
DO_ .. END
DO .. WHILE
REPEAT

Functions

Built-in functions or
user-defined procedures

Syntax

```
[obj-identifier=] function-name [ (expr [,...]) ]
```

Where:

obj-identifier	specifies the debug object to which the function's return value is assigned.
function-name	specifies either the keyword for a built-in function, or by the user-defined name of a predefined procedure.
expr	specifies one or more numbers or expressions separated by commas. The parentheses are required punctuation to delimit *expr*.

Discussion

The emulator functions are divided into the following categories. Each category is explained in more detail in its own entry.

Character Functions

ISALNUM	ISALPHA	ISASCII	ISCNTRL
ISDIGIT	ISLOWER	ISPRINT	ISPUNCT
ISSPACE	ISUPPER	ISXDIGIT	
TOASCII	TOLOWER	TOUPPER	

I/O Functions

GETS	PUTS	PRINTF	SCANF
GETCHAR	PUTCHAR	SPRINTF	SSCANF

Math Functions

LOGE	COS	SIN	TAN
LOG10	ACOS	ASIN	ATAN
ABS	EXP	POW	ATAN2
SQRT			

Miscellaneous Functions

CLRTOBOT	TIME	RAND	CLEAR
CLRTOEOL	CTIME	SRAND	MOVE
SLEEP			

String Functions

STRCAT	STRCMP	STRCPY	STRLEN
STRNCAT	STRNCMP	STRNCPY	

A user-defined function is a PROC. Refer to the PROC entry for more information.

Cross-References

C: A Reference Manual
Character Functions
<expr>
I/O Functions
Math Functions
Miscellaneous Functions
PROC
String Functions

GO

Starts emulation and controls break and trace functions

Syntax

```
GO   [ DISPLAY                                                    ]
     [ [init-emu] [init-AM]  [AM-programming]                     ]
```

```
init-emu ::=  [FROM address] [WHEN ISYNCH]
```

```
init-AM ::=        [ TRACE   ]  [ OSYNCH ]
                   [ NOTRACE ]  [ OSYNCL ]
```

```
AM-programming ::=     [ FOREVER                                          ]
                       [ TIL user-state-0 [THEN user-state-1]             ]
```

user-state (-0 and -1) ::=

```
[ event  [OCCURS expr] [actions] [ORIF event [actions] ]                  ]
[ event  [actions] [ORIF event [OCCURS expr] [actions] ]                  ]
[ (event ORIF event)  [OCCURS expr]  [actions]                            ]
```

```
event ::=  [ wr-part [synch-input] [FULLBUF]  ]
           [ synch-input [FULLBUF]            ]
           [ FULLBUF                          ]
```

wr-part ::=

```
{EXECUTION xpartition                                                     }
{BUS xpartition [device-type] [access-type] [data-field]                  }
{IOPORT io-partition [device-type][io-access-type][data-field]            }
```

actions ::=

```
[ TRACE    ]  [ OSYNCL ]  [ MEM partition ]  [ STOP    ]
[ NOTRACE  ]  [ OSYNCH ]  [ REGS          ]  [ HALT    ]
              [ OSYNCT ]  [ PCB           ]  [ RESTART ]
                                             [ REPEAT  ]
```

Where:

GO — if entered without parameters, it assumes default values. These default values change depending on when GO is entered.

If you do not specify any parameters the first time you enter GO after activating the emulator, the command assumes the following default condition: GO FROM $ (current execution pointer) TRACE OSYNCH FOREVER.

If you enter GO without parameters after you have run and halted the emulator, the command restarts the emulator FROM $ and restarts the Activity Monitor using the same init-AM and AM-programming parameters specified in the previous GO.

If you enter GO without parameters after you have run and stopped the Activity Monitor, but not halted the emulator, the command restarts the Activity Monitor using the same init-AM and AM-programming parameters specified in the previous GO. It does not interrupt emulation.

DISPLAY — displays *init-AM* and *AM-programming* conditions of the last entered GO command. It evaluates expressions and displays these values rather than the actual expressions entered.

init-emu — specifies the initial conditions for emulation.

init-AM — specifies the initial conditions for the Activity Monitor.

AM-programming specifies the Activity Monitor programming (i.e., the *events* to search for and the *actions* to take once the specified *events* occur).

Initial Emulation (init-emu) Conditions

FROM *address* specifies the point at which program execution starts when the GO command is invoked. If omitted, the default execution start address is the current execution pointer.

WHEN ISYNCH starts emulation when the external synchronous input line becomes a logical high. The emulator waits for the input line without servicing any interrupts, regardless of the setting of the ENI tool variable. Use this parameter to start more than one emulator synchronously. If this parameter is omitted, emulation starts up regardless of the level of the ISYNC line.

Initial Activity Monitor (init-AM) Conditions

TRACE specifies that once the GO command is entered, trace collection is on. If no trace specification is entered (either TRACE or NOTRACE), the Activity Monitor uses the last entered trace specification. If you have not entered a trace specification since activating the emulator, the default is TRACE.

NOTRACE specifies that once the GO command is entered, trace collection is off.

OSYNCH causes the synchronous output level to go high on the first instruction boundary that occurs after emulation starts. If no OSYNC level is entered (OSYNCH or OSYNCL), the Activity Monitor uses the last entered OSYNC specification, either from *init-AM* or from the OSYNC tool variable. If you have not entered an OSYNC specification since activating the emulator, the default is OSYNCH.

OSYNCL	causes the synchronous output level to go low on the first instruction boundary that occurs after emulation starts.

Activity Monitor Programming (AM-programming) Conditions

FOREVER	causes emulation to continue until a HALT command is entered or an emulator time-out (e.g., BUSACT, MEMRDY, etc.) occurs. The Activity Monitor remains active until emulation halts or a STOP command is entered.
TIL	signals that *user-state-0* will follow.
THEN	signals that *user-state-1* will follow.
user-state (*0* and *1*)	consists of one or two *events* to recognize, each followed by the *actions* to occur once the *event* is recognized.
event	is a pattern to be recognized by the Activity Monitor. The pattern can be either an execution address or a bus cycle (*wr-part*), combined with a particular ISYNC level and an indication that the trace buffer is full.
wr-part	describes the CPU bus patterns the Activity Monitor can detect (using a word recognizer). Two types of *wr-part* patterns exist: bus cycles and execution addresses. The new terms and variables introduced in the *wr-part* syntax are each discussed in detail in the GO syntax descriptions that follow.
EXECUTION	designates that the following *xpartition* is to be interpreted as the address of the first byte of an instruction.
BUS	designates that the following *xpartition* is to be interpreted as the address of a bus cycle initiated by the BIU portion of the 80C186 microprocessor. In other words, the address of a bus cycle that is not an I/O read or write.
IOPORT	designates that the following *io-partition* is to be interpreted as an I/O port address.

xpartition — is either a standard address *partition* or a *don't-care-address*.

partition — specifies an address or a range of addresses in the format *address* TO *address* or *address* LENGTH *number-of-bytes* (see the Partition entry).

io-partition — specifies a 16-bit I/O port address or a range of 16-bit I/O port addresses in the format IOPORT TO IOPORT or a bit bit I/O port with don't care bits.

don't-care address — describes a physical address containing don't care bits, that is, bits whose value could be either 0 or 1. The radix can be either binary (Y) or hex (H); a bit (binary) or hex digit that is don't care is indicated by replacing it with the letter X or x. For example, 1234XHP indicates that physical addresses 12340-1234F or desired. Likewise, 1X235HP means that addresses 10245H, 11235H, 12235H,... IF235H are acceptable. 0000000000000000 000xYP is an example of a physical address in binary format that would select either physical addresses 00000HP or 00001HP. 20 bits are valid for the 80186/80188.

For IOPORTS, the don't care field is a 16-bit number. For example, IOPORT 200XH would select IOPORTS in the range 2000H-200FH.

device-type — as part of the *wr-part* syntax, specifies the source of the bus access as follows:

- DMA0 -- any access type made by DMA channel 0
- DMA1 -- any access type made by DMA channel 1
- DMA -- any access made by either DMA channel
- CPU -- any non-DMA access

access-type — as part of the *wr-part* syntax, specifies the type of bus access made. The *access-type* syntax is as follows:

- WRITE -- memory write
- READ -- memory read
- ACCESS -- memory read or write
- FETCH -- instruction pre-fetch

io-access-type — as part of the *wr-part* syntax, specifies the type of I/O access made. The *io-access-type* syntax is as follows:

- IREAD -- I/O read
- IWRITE -- I/O write
- IO -- any type of I/O bus cycle

data-field — specifies a data-line pattern to be recognized by the Activity Monitor. The syntax is as follows:

```
{ [data-type] expr [TO expr]   }
{ don't-care-data-value        }
```

data-type — describes the width of the data. Use this parameter to avoid data-width ambiguity when specifying a data value. If *data-type* is omitted, a WORD *data-type* is assumed for the 80186; for the 80188, the default is BYTE. Thus, 0H is interpreted as 0000H. Legal *data-types* are: BYTE, WORD, INT1, INT2, ORD1, and ORD2.

expr — is an expression that evaluates to a number (see the <expr> entry).

don't-care data-value
describes a 16-bit data field containing don't care bits, that is, bits whose value could be either 0 or 1. The radix can be either binary (Y) or hex (H); a bit (binary) or hex digit that is don't care is indicated by replacing it with the letter X or x. For example, 0AX9H selects data values in the set [0A09H, 0A19H, 0A29H,...]

synch-input
is the value of the synchronous input line. The syntax of *synch-input* is as follows:

- ISYNCH -- synchronous input line is TTL high
- ISYNCL -- synchronous input line is TTL low
- ISYNCT -- synchronous input line toggles

FULLBUF
programs the Activity Monitor to perform an *action* when the trace buffer is full.

OCCURS *expr*
specifies that an *event* must occur *expr* times. One 8-bit counter is available per *user-state*. Thus, only one OCCURS clause can occur in a *user-state*, and only two OCCURS clauses can occur in a single GO command invocation. The counter delays the generation of an EVENT-TRUE signal until *expr* occurrences of the *event* take place.

actions
is a list of actions performed by the emulator after the Activity Monitor recognizes the specified *event*. If no actions are specified in the last entered *user-state*, the default *action* is to HALT. If no actions are specified, and *user-state-1* is specified, the default *action* in *user-state-0* is to switch to *user-state-1*.

There are three basic *action* types:

- Activity Monitor and emulation control (i.e., TRACE/NOTRACE, STOP, HALT, RESTART, REPEAT, and ORIF).
- OSYNC control (i.e., OSYNCH, OSYNCL, and OSYNCT).
- Data retrieval (i.e., MEM, REGS, and PCB).

The following paragraphs describe each *action* parameter.

MEM *partition* — specifies that when the Activity Monitor recognizes the occurrence of the event, it copies the contents of the specified memory *partition* into the trace buffer. The *partition* can be no longer than 40 bytes. This parameter is available only if FASTBREAK = TRUE.

REGS — specifies that when the Activity Monitor recognizes the occurrence of the *event*, it copies the register and flag contents into the trace buffer. This parameter is available only if FASTBREAK = TRUE (see the REGS entry in this chapter).

PCB — specifies that when the Activity Monitor recognizes the occurrence of the event, it copies the contents of the Peripheral Control Block registers into the trace buffer. This parameter is available only if FASTBREAK = TRUE. (See the PCB entry in this chapter for more information.)

STOP — specifies that the Activity Monitor stops when it recognizes the occurrence of the *event*. Emulation is not affected.

HALT — specifies that both the Activity Monitor and the emulator halt when the Activity Monitor recognizes the occurrence of the *event*.

RESTART	specifies that when the Activity Monitor recognizes the occurrence of the *event*, it starts over with the conditions specified in *user-state-0*. Typically, RESTART is used as an action for *user-state-1*, causing a transition back to *user-state-0*.
REPEAT	specifies that the Activity Monitor is to search for the preceding *event* repeatedly. The search starts over whenever an occurrence of the *event* is recognized.
ORIF	ORs the following *event* with the preceding event. The actions associated with the *event* that occurs first take precedence.
TRACE	specifies that trace collection is turned on when the Activity Monitor recognizes the occurrence of the *event*.
NOTRACE	specifies that trace collection is turned off when the Activity Monitor recognizes the occurrence of the *event*.
OSYNCL	causes the synchronous output level to go low on the first instruction boundary that occurs after the Activity Monitor recognizes the occurrence of the *event*.
OSYNCH	causes the synchronous output level to go high on the first instruction boundary that occurs after the Activity Monitor recognizes the occurrence of the *event*.
OSYNCT	causes the synchronous output level to toggle on the first instruction boundary that occurs after the Activity Monitor recognizes the occurrence of the *event*.

Discussion

The GO command controls the emulation session. With this command, you can establish the starting conditions for the emulator and Activity Monitor, specify the values for word recognition, and specify the actions to be taken as the result of word recognition.

There are three basic parameter groups associated with the GO command:

init-emu	initial emulation conditions.
init-AM	initial Activity Monitor conditions.
AM-programming	word recognition and action conditions.

You can use the default GO conditions by entering the GO command without parameters. The default conditions depend on when you enter the command.

If you do not specify parameters the first time you enter GO after activating the emulator, the default GO conditions are: GO FROM $ (current execution pointer) TRACE OSYNCH FOREVER. Once you have entered a GO command, the default *init-emu* condition is FROM $, and the default *init-AM* and *AM-programming* conditions are the same as those entered with the previous GO.

You can display the current *init-AM* and *AM-programming* conditions by entering the GO DISPLAY command.

The effect of the GO command on emulation and the Activity Monitor differs, depending on the operating mode of the emulator at the time GO is entered. If the emulator is in Halt mode and you enter the GO command, it starts both the emulator and the Activity Monitor. If the emulator is in Run mode and you enter the GO command without *init-emu* parameters, the command restarts the Activity Monitor, but does not affect the emulator. If you enter an *init-emu* parameter, the command also halts and restarts the emulator.

The remainder of this command entry discusses the various GO command parameters and their effects on emulation and the Activity Monitor. It also describes the effects of the QUERY and FASTBREAK variables on GO command operation.

Initial Emulator Conditions

The *init-emu* parameters determine when and where emulation starts. You can specify that emulation starts FROM a specified address and/or WHEN the ISYNC line goes high (ISYNCH).

If you enter the FROM *address* parameter when the emulator is in Halt mode, the emulator starts emulation at the specified address. If you enter the parameter when the emulator is in Run mode, the emulator halts and then restarts at the specified address.

If you omit the FROM *address* parameter when the emulator is in Halt mode, the emulator starts FROM $ (current execution pointer). If you omit the parameter when the emulator is in Run mode, emulation is not affected.

If you enter the WHEN ISYNCH parameter when the emulator is in Halt mode, the emulator starts emulation when the ISYNC line goes high. If you enter the parameter when the emulator is in Run mode, the emulator halts and then waits for ISYNC to go high before restarting. If you omit the WHEN ISYNCH parameter, the emulator disregards the level of ISYNC.

The usual application for WHEN ISYNCH is when you are attempting the simultaneous start of several emulators connected via the emulator ISYNC line. In this case, all emulators connected to the line wait for ISYNC to go high before starting.

You can stop emulation with a HALT action in AM-programming or by entering the HALT command.

The setting of ENI determines if interrupts are serviced when the emulator enters Halt mode. (Refer to the ENI entry for more information.)

Initial Activity Monitor Conditions

The *init-AM* parameters specify the starting conditions of the Activity Monitor. You can specify the starting Activity Monitor condition as TRACE or NOTRACE, combined with OSYNCH or OSYNCL.

If you specify TRACE, the Activity Monitor starts storing data immediately on invocation of GO. If you specify NOTRACE, the Activity Monitor does not store data even though a GO command has been invoked and emulation has started. It does not store data until it encounters a TRACE action from a new invocation of GO or from a TRACE action with *AM-programming*.

You can combine the OSYNCH/OSYNCL parameter with either TRACE or NOTRACE, or use it by itself. This parameter sets the value of the OSYNC output line to high (OSYNCH) or low (OSYNCL).

If you omit the TRACE/NOTRACE parameter when invoking GO, the Activity Monitor uses the trace condition specified in the last invocation of GO. If you have not entered a trace specification since activating the emulator, the default is TRACE.

If you omit the OSYNCH/OSYNCL parameter, the Activity Monitor uses the last entered OSYNC specification, either from *init-AM* or from the OSYNC tool variable. If you have not entered an OSYNC specification since activating the emulator, the default is OSYNCH.

You can stop the Activity Monitor by programming a STOP action in *AM-programming*, by entering the STOP command, or by entering the HALT command. The latter also halts the emulator.

Note that while the Activity Monitor is stopped, the emulator may continue to execute code, and some loss of bus information may occur. To avoid missing bus cycles when changing Activity Monitor programs, use the HALT command to stop emulation, and then enter the new GO command. Note also that the Activity Monitor must be inactive in order to display the trace buffer contents with the PRINT command.

Activity Monitor Programming

The *AM-programming* parameters establish how the Activity Monitor operates during the emulation session. The programming conditions can be FOREVER or TIL *user-state-0* [THEN *user-state-1*].

If you use the FOREVER parameter, the Activity Monitor proceeds as specified by *init-AM*. No events are detected and no actions are taken.

The TIL *user-state-0* [THEN *user-state-1*] parameters allow you to specify word recognition and actions. Each *user-state* allows two word recognizers and two actions, diagrammed as follows:

event [OCCURS *expr*] [*actions*] [ORIF *event* [*actions*]]
event [*actions*] [ORIF *event* [OCCURS *expr*] [*actions*]]
(*event* ORIF *event*) [OCCURS *expr*] [*actions*]

You can define *events* as execution addresses (EXECUTION), bus events (BUS), or I/O port addresses (IOPORT). Note, however, that a single user-state can contain only one range (address and data). In addition, an EXECUTION event cannot be combined with a BUS or IOPORT event within a single user-state.

The *action* parameter determines the action taken when an event is recognized. There are three basic action types:

- Activity Monitor and emulation control -- TRACE, NOTRACE, STOP, HALT, RESTART, REPEAT, and ORIF.
- OSYNC control -- OSYNCH, OSYNCL, and OSYNCT.
- Data retrieval -- MEM, REGS, and PCB.

The data retrieval *actions* are available only if the FASTBREAK tool variable is set to TRUE (non-zero).

Each *user-state* has an occurrence counter (OCCURS *expr*) that you can tie to one or both events. The counter counts the occurrences of the specified event and delays the action until the count is satisfied.

A bus event is composed of an address or range of addresses, an access type, a data type, and a data field.

Certain restrictions apply when using WORD data types and odd-aligned addresses. These restrictions include:

- FETCH accesses -- A WORD data type cannot be used when specifying a FETCH access to an odd address. If you want the Activity Monitor to recognize a FETCH to an odd address, use a BYTE data type.
- READ or WRITE accesses -- A WORD data type can be used when specifying a READ or WRITE at an explicit odd address. However, it cannot be used if the odd address is defined as a range or if the odd address contains an address-mask specifying don't care bits.

- All word-wide data for the 80188 is composed of successive byte accesses. Only odd-aligned word-wide data for the 80186 is composed of successive byte accesses. Events specifying word-wide data must include a specific address. For example, GO TIL BUS WRITE 1234 will not work properly; the correct syntax is GO TIL BUS *addr* WRITE 1234 (where *addr* cannot contain any don't-care values).
- Events containing word-wide data qualified by a FETCH are not guaranteed to be recognized by the 80188. (The 80186 does not have this restriction.) This is because the WORD FETCH can be split up by another bus cycle.

There are no special restrictions regarding the use of BYTE data types and odd addresses, nor the use of WORD and BYTE data types and even addresses; however, a byte data value that is not associated with an explicit odd address is assumed to be aligned with an even address.

The Effects of QUERY

The QUERY tool variable controls the behavior of the command-line prompt after emulation begins.

If QUERY is set to FALSE (zero) and the GO command is executed, the command-line prompt disappears from the screen. You cannot enter any commands at this time. The command-line prompt reappears when the Activity Monitor stops or the emulator halts. You can also display the command-line prompt by pressing <Ctrl><Break>. This returns you to the command line prompt, but does not affect emulation.

If QUERY is set to TRUE (non-zero) and you execute the GO command, a command-line prompt appears immediately after emulation begins. However, the status of the emulator is not automatically displayed on the command line. Status is displayed only when you execute the STATUS command or another command that communicates with the emulator.

For more information on the QUERY tool variable, refer to the appropriate entry in this chapter.

The Effects of FASTBREAK

The FASTBREAK tool variable determines when interference with emulation can occur while in Run mode. Setting FASTBREAK to FALSE (zero) prevents memory and register access requests during emulation. As such, the MEM *partition*, REGS, and PCB actions are not available when FASTBREAK is set to FALSE. The prompt in Run mode with this setting of FASTBREAK is "emu>" or "emu+>".

Setting FASTBREAK to TRUE (non-zero) allows memory and register access requests during emulation. Execution of these actions result in emulation breaks for data retrieval. The prompt in Run mode with this setting of FASTBREAK is "fst>" or "fst+>".

For more information on the FASTBREAK tool variable, refer to the appropriate entry in this chapter.

Differences Between ICE™-186/188 and I²ICE™ Software

The emulator GO command syntax and operation differs from the I²ICE GO command. I²ICE users should note of the following:

1. The emulator GO syntax requires that memory addresses be preceded by the keyword EXECUTION or BUS. There is no default address type.
2. The entry "GO TIL 0" is interpreted by the emulator as "GO TIL WORD 0", not "GO TIL 0", which is interpreted by I²ICE as "GO TIL an instruction at physical location 0 is executed". With the "GO TIL 0" entry, the emulator looks for a data word value of 0.

Examples

1. This example illustrates a successful emulator start-up and halt.

```
hlt> QUERY = 1
hlt> GO FROM 1100:4H FOREVER
   Use S (status) command to determine emulation status.
emu+> HALT
   Emulator Status:  halted (not servicing interrupts).
     EXECPOINTER is :MESSG.DELAY_A_LITTLE_BIT_#162 + aH (1000:0058H)
   Activity Monitor Status:  Stopped.
hlt>
```

2. This example illustrates the effects of setting QUERY to zero (FALSE). The command-line prompt disappears when the GO command is executed, then reappears when the emulator halts.

```
hlt> QUERY = 0
hlt> GO FROM 1100:4H TIL EXECUTION :messg.delay_a_\
hlt>> little_bit_ OCCURS 50T HALT
   Waiting...
   Waiting...
   Emulator Status:  halted (not servicing interrupts).
     EXECPOINTER is :MESSG.DELAY_A_LITTLE_BIT_#151 + 1H (1000:0003dH)
   Activity Monitor Status:  Stopped.
hlt>
```

3. This example shows the same GO conditions as those used in Example 2, except that QUERY is set to non-zero (TRUE). After the GO command is entered, a Run mode prompt appears and you can enter commands. However, emulator status is not immediately updated on the screen.

```
hlt> QUERY = 1
hlt> GO FROM 1100:4H TIL EXECUTION :messg.delay_a_\
hlt>> little_bit_ OCCURS 50T HALT
   Use S (status) command to determine emulation status.
emu+> S
   Emulator Status:  running user code.
   Activity Monitor Status:  Running
emu+> S
   Emulator Status:  running user code.
   Activity Monitor Status:  Running.
emu+> S
   Emulator Status:  halted (not servicing interrupts).
     EXECPOINTER is :MESSG.DELAY_A_LITTLE_BIT_#151 + 1H (1000:0003dH)
   Activity Monitor Status:  Stopped.
hlt>
```

4. In this example, the first execution of GO traces from point A (line #101) to point B (line #197) and then stops the Activity Monitor. The second execution of GO repeats the action, without interrupting emulation.

```
hlt> QUERY = 0
hlt> GO NOTRACE TIL EXECUTION #101 TRACE THEN \
hlt>> EXECUTION #197 STOP
```

```
   Emulator Status:  running user code.
   Activity Monitor Status:  Stopped.
emu> GO
Warning: NOTRACE is currently the initial TRACE condition.
   Emulator Status:  running user code.
   Activity Monitor Status:  Stopped.
emu> HALT
   Emulator Status:  halted (not servicing interrupts).
     EXECPOINTER is :MESSG.DELAY_A_LITTLE_BIT_#163 + 10H (1000:0005eH)
   Activity Monitor Status:  Stopped.
hlt>
```

5. This example traces from point A (:messg.delay_a_little_ bit_) to point B (:messg.rotate_message_) repeatedly, until the trace buffer is full.

```
hlt> QUERY = 0
hlt> GO NOTRACE TIL EXECUTION :messg.delay_a_little_\
hlt>> bit_ TRACE THEN EXECUTION :messg.rotate_\
hlt>> message_ NOTRACE RESTART ORIF FULLBUF HALT
   Emulator Status:  halted (not servicing interrupts).
     EXECPOINTER is :MESSG.DELAY_A_LITTLE_BIT_#129 +eH (1000:005cH)
   Activity Monitor Status:  Stopped.
hlt>
```

6. This example illustrates how you can use fastbreaks to access processor register contents during emulation. The Activity Monitor traces to point A (line #92), copies the register and flag contents to the trace buffer, then repeats the action until point B (line #226). At point B, the emulator and Activity Monitor halt.

```
hlt> QUERY = 1
hlt> FASTBREAK = 1
hlt> GO FROM 1100:4H TRACE TIL EXECUTION #92 REGS\
hlt>> REPEAT ORIF EXECUTION #226 HALT
   Use S (status) command to determine emulation status.
fst+> S
   Emulator Status:  halted (not servicing interrupts).
     EXECPOINTER is :MESSG.ROTATE_MESSAGE_#226 + 3H (1000:0095H)
   Activity Monitor Status:  Stopped.
hlt>
```

7. This example again illustrates the use of fastbreaks. The Activity Monitor stores a data structure repeatedly, until 255 fetches occur.

```
hlt> QUERY = 1
hlt> FASTBREAK = 1
hlt> GO TRACE TIL BUS :messg.display_buffer_ WRITE \
hlt>> MEM :messg.display_buffer_ LENGTH 2 REPEAT \
hlt>> ORIF FETCH OCCURS 255T HALT
   Emulator Status:  halted (not servicing interrupts).
     Execpointer is :MESSG.DELAY_A_LITTLE_BIT_#129 +3H (1000:0051H)
   Activity Monitor Status:  Stopped.
hlt>
```

Cross-References

ENI	OSYNC	QUERY
<expr>	Partition	REGS
FASTBREAK	PCB	STATUS
HALT	PRINT	STOP

HALT

Halts the emulator and stops the Activity Monitor

Syntax

HALT

Discussion

The HALT command halts the emulator and stops the Activity Monitor. You can enter this command at any time.

When MEMRDY/IORDY timeouts are disabled, and the target does not respond with a required signal, a HALT forces a RDY to terminate the hang, clearing the error conditions.

While the emulator is halted, it maintains the following processor functions:

- DMA transfers
- Refresh cycles
- Bus HOLD/HLDA

The extent of interrupt servicing during Halt mode is dependent on the setting of the ENI tool variable. The available settings include:

- ENI = 0 -- The emulator does not service any interrupts in Halt mode. Interrupts are marked as pending and are serviced once the emulator is re-started. The /RES signal from the prototype is ignored.
- ENI = 1 -- The emulator services internal processor interrupts in Halt mode, except during mapped I/O requests and synchronous emulator start-ups. User interrupts are marked as pending and are serviced once the emulator is re-started. The /RES signal from the prototype is ignored.
- ENI = 2 -- The emulator services both user and internal processor interrupts in Halt mode, except during mapped I/O requests and synchronous emulator start-ups.

Example

This example demonstrates the effect of the HALT command on the emulator, given two settings of ENI. When the HALT command is first invoked, the setting of ENI is 0; this HALT stops the emulator. With ENI set to 2, the subsequent invocation of HALT causes the emulator to service internal processor and user interrupts while it is halted.

```
hlt> ENI = 0
hlt> GO
  Use S (status) command to determine emulation status.
emu+> HALT
  Emulator Status:  halted (not servicing interrupts).
   Execpointer is :MESSG.DELAY_A_LITTLE_BIT_#129 + 3H (1000:0051H)
  Activity Monitor Status:  Stopped.
hlt> ENI = 2 1000:2FF0H
hlt> GO
  Use S (status) command to determine emulation status.
hlt> HALT
  Emulator Status:  halted (servicing internal CPU and user interrupts
                    at 1000:2ff0H).
   Execpointer is :MESSG.DELAY_A_LITTLE_BIT_#129 + aH (1000:0058H)
  Activity Monitor Status:  Stopped.
idi>
```

Cross-References

CAUSE
ENI
GO
STOP

HELP / Help Mode

Provides on-line information on keywords, topics, and error messages

Syntax

```
HELP [ ERR source_id error_number ]
     [ help_item                  ]
```

Where:

HELP	displays all of the emulator keywords and options as shown in the syntax menu.
ERR	specifies that additional information is requested about an error.
source_id	specifies the origination of the error. Enter one of the following *source_id* options to request more help information: BUILTIN_FUNCTION DBM KERNEL KERNEL_CMD ICE186 MATH_FUNCTION
error_number	specifies the error number. Enter the number with the T suffix.
help_item	specifies any topic or option shown by the command HELP <Enter>.

Discussion

Use the HELP command to reference a description of emulator commands, topics, or extended error messages. Help information displayed includes a description, examples, and cross-references. For syntax information regarding a specific command, refer to the appropriate entry in this chapter.

Help information for error messages includes a description and may describe what occurred to cause the error and what action to take. If an error message is followed by [*], the error has extended information available. When requesting help for an error, always add the T suffix to the error number if the BASE parameter is not set to decimal. If the T suffix is not used and the base is not decimal, the error number will not be interpreted as you expect. For example, a message will be displayed indicating that the error number was not found or information about an unrelated error will be displayed.

To find out what topics have help information available, enter the HELP command without options and note the entries listed.

In addition to the HELP command, a help mode feature exists that provides windowed help for keywords as they are entered. The help mode is activated and deactivated by toggling <Ctrl>B while the syntax menu is activated. Screens of text in the help window can be scrolled forward by pressing <Ctrl>R and backward by pressing <Ctrl>U. Lines of text in the help window can be scrolled forward by pressing <Ctrl>K and backward by pressing <Ctrl>O. For more information on the help mode, refer to Chapter 3, Section 3.2.2 and Section 3.2.3.

Examples

1. Enter the HELP command to see what commands, keywords, and topics have help information.

```
hlt> HELP

AAA       AAD          AAM
AAS       ACTIVATE     ADC
ADD       AFL          ALEMODE
AND       APPEND       ASM
BASE      BOUND        BREAK
.
.
.
```

2. Use HELP to learn how to use the HELP command.

```
hlt> HELP help
                              **>help<**

DESCRIPTION:

The HELP command displays information about a keyword, command, or error
message.  For screen handling, CTRL-S stops the screen from scrolling,
while any key resumes the screen scrolling.  In addition to an on-line
HELP command, there is a help mode that can be enabled/disabled by
toggling CTRL-B.  When in help mode, help information is displayed in a
window at the top of the screen for the various keywords as they are
entered.  The size of the window may be modified by using the HELPSIZE =
<number> directive in the configuration file at invocation time.

EXAMPLES:

1.  Request additional help for an error that has extended information.  A
[*] symbol following the error text indicates that extended information is
available.
hlt> 3++

[kernel] Error #27t
Illegal operation. [*]
hlt> HELP ERR kernel 27t
...error help text...
hlt>
```

```
2.  Request information about the keyword TO.
hlt> HELP TO
...help text...
hlt>

SEE ALSO:

    ERR

hlt>
```

Cross-References

Chapter 3, Section 3.2.3
ERROR

HOLDIO

Suspends I/O requests
to ICE-mapped ports

SYNTAX

```
HOLDIO
```

DISCUSSION

Use the HOLDIO command to suspend pending I/O input requests for ICE-mapped ports.

When I/O port addresses are mapped to ICE, the emulator simulates I/O accesses during emulation. It displays data values written from ports during I/O read cycles, and prompts you for data value inputs during I/O write cycles.

Instead of entering a data value for an I/O write cycle, you can enter the HOLDIO command. This suspends the input request and releases control to the command language. You can then enter any commands that do not access the 80C186 microprocessor.

While HOLDIO is active, the emulator ignores any commands that access the processor, such as PORT or WPORT. In addition, it ignores any interrupts, regardless of the setting of ENI.

You release HOLDIO by entering the RELEASEIO command. The emulator immediately displays the last suspended I/O input request.

Example

This example illustrates suspending I/O requests using the HOLDIO command.

```
hlt> GO FROM 0:0
  Port f000H requests byte input (enter value): HOLDIO
  Emulator Status:  hung pending mapped IO request.
Use RELEASEIO to service request.
  Activity Monitor Status:  Running.
IO?+> @type port.c

...command is executed...

IO?+> RELEASEIO
  Port f000H requests byte input (enter value): 12H
  Emulator Status:  running user code.
  Activity Monitor Status:  Running.
emu+>
```

Cross-References

MAPIO
Prompts
RELEASEIO

IF ... ELSE

Groups and conditionally executes commands

Syntax

```
IF (expr) {commands1} [ ELSE {commands2} ]
```

Where:

IF	conditionally executes commands based on the value of the IF condition as specified by *expr*.
(*expr*)	specifies a number or an expression which must evaluate as either TRUE (non-zero) or FALSE (zero). The parentheses are required punctuation.
{*commands1*}	specifies one or more emulator commands that are executed when *expr* evaluates to TRUE (non-zero). The braces are required punctuation when multiple commands are entered.
ELSE {*commands2*}	specifies one or more emulator commands (*commands*2) that are executed if *expr* evaluates to FALSE (zero). The braces are required punctuation when multiple commands are entered.

Discussion

The IF...ELSE statement tests the *expr* condition and if TRUE (non-zero) executes the command(s) in the *commands*1 specification. If the optional ELSE keyword is used, then any commands in the *commands*2 specification are executed if the *expr* condition evaluates to FALSE (zero).

IF blocks can be nested. When nested, the optional ELSE clause associates with the closest IF clause. The IF statement resembles the C language IF control construct.

The ELSE keyword must be on the same command line as the end of the {*commands*1} block. A continuation character (\) followed by <Enter> can be used at the end of the last line of the {*commands*1} block to move the ELSE keyword to the next line.

Examples

1. The following example shows how to use the IF...ELSE statement to test a condition. If the test condition (a > b) evaluates to true, then z takes the value of a. If the test condition evaluates to false, z takes the value of b. Assume a, b and z have been previously defined as INT2's.

```
hlt> IF (a > b)
hlt>> z = a \
hlt>> ELSE
hlt>> z = b
hlt>
```

2. The following example uses the IF...ELSE statement without the ELSE clause. If the test condition (AX > 0) is true, then increment the AX and BX registers; otherwise, do nothing.

```
hlt> IF ( AX > 0 )
hlt>> {
hlt>> AX += 1
hlt>> BX += 1
hlt>> }
hlt>
```

Cross-References

DO...WHILE
<expr>
IF_
WHILE

IF_

Groups and conditionally executes commands

Syntax

```
IF_  expr
  THEN
  commands1
  [ELSE commands2]
END[IF]
```

Where:

IF_	conditionally executes commands based on the value of the IF_ condition as specified by *expr*.
expr	specifies a number or an expression which must evaluate as either TRUE (non-zero) or FALSE (zero).
THEN *commands1*	specifies one or more emulator commands that are executed when *expr* evaluates to TRUE (non-zero).
ELSE *commands2*	if specified, *commands*2 is one or more emulator commands that are executed when *expr* evaluates to FALSE (zero).
END[IF]	specifies the end of the IF_ block command.

Discussion

The IF_ statement tests the *expr* condition and if TRUE (non-zero), executes any commands in the *commands*1 specification. If the optional ELSE keyword is used, then the commands in the *commands*2 specification are executed if the *expr* condition evaluates to FALSE (zero).

IF_ ... END blocks can be nested. When nested, the optional ELSE clause associates with the closest IF_ clause. The IF_ block command resembles the PL/M or Pascal IF...THEN...ELSE control construct.

NOTE

The following commands are not executable inside the IF_ command: WHILE, UNTIL, IF, FOR, DO...WHILE, SWITCH, and INCLUDE.

Examples

1. The following example shows how to use the IF_ statement to test a condition. If the test condition (a > b) evaluates to true, then z takes the value of a. If the test condition evaluates to false, then z takes the value of b.

```
hlt> DEFINE INT2 A=3
hlt> DEFINE INT2 B=4
hlt> DEFINE INT2 Z=0
hlt> IF_ a > b
hlt>> THEN z = a
hlt>> ELSE z = b
hlt>> ENDIF
hlt>
```

2. The following example shows how to use the IF_ statement without the ELSE option. If the test condition AX > 0 is true, increment the AX and BX registers; otherwise, do nothing.

```
hlt> IF_ AX > 0
hlt>> THEN
hlt>> AX+=1
hlt>> BX+=1
hlt>> ENDIF
hlt>
```

Cross-References

DO...WHILE
<expr>
IF .. ELSE
WHILE

INCLUDE

Executes emulator commands
defined in a disk file

Syntax

INCLUDE [NOLIST] *filename*

Where:

INCLUDE	executes the emulator commands in the specified file.
NOLIST	suppresses the echoing of file contents to the screen.
filename	is the name of a disk file that contains a sequence of emulator commands to be executed. It is created either by using the PUT or APPEND commands or by editing a text file.

Discussion

The INCLUDE command causes input to be taken from the named file until the end-of-file mark is reached. If the *filename* does not specify a directory, the emulator software searches the current directory. If the file is not found, it searches the directories specified by the host system's PATH environment parameter.

The *filename* text file can contain other INCLUDE commands, that is, nesting of INCLUDEs is permitted. Use this command only with text files that contain emulator commands.

The output of the INCLUDE command is the same as if the commands in the *filename* had been entered directly at the console. With the NOLIST option, command echoing is suppressed, but the responses appear on the console normally. Error messages are displayed if errors occur while processing a command in the file. If the error is severe, inclusion of the file is terminated.

NOTE

INCLUDE is not permitted in PROCs or block commands/control constructs.

Examples

1. The following example includes the contents of a file without echoing the file contents to the screen.

```
hlt> INCLUDE NOLIST setup.inc
```

2. The following example includes and views the contents of a file.

```
hlt> INCLUDE setup.inc
hlt> DEFINE ord1 i
hlt> DEFINE ord2 j
hlt> DEFINE int2 k
hlt> DEFINE ord4 s_factor
hlt> DEFINE ord4 r_factor
```

Cross-References

APPEND / PUT
Invocation

Invocation

Procedures for invoking the emulator software
and activating the emulator hardware

Discussion

The Invocation process involves two steps: invoking the emulator software and activating the emulator hardware. You can perform these steps automatically by using the RUN186.BAT file, or you can perform them explicitly by using the set of invocation commands and parameters.

If you invoke the emulator with the RUN186.BAT file, the file automatically invokes the software and activates the emulator. It reads parameters from the include file (ICE186.INC) and three configuration files (I186.CFG, ICE186.CFG, and V186.CFG). These parameters establish the emulator I/O and operating environment and define some commonly used LITERALLY abbreviations (e.g., EX for Execution, a GO command parameter).

You can modify the parameters of the include and configuration files prior to invocation, or you can override them by invoking the emulator explicitly using invocation options. Procedures for modifying the parameters prior to invocation are provided in the *ICE™-186/188 In-Circuit Emulator Installation Supplement.*

For more information on the configuration files refer to Chapter 3, Section 3.3.1. For more information on the invocation options refer to Chapter 3, Section 3.3.2.

Using the RUN186.BAT File

To use the RUN186.BAT file, enter "RUN186" at the DOS prompt:

```
C> RUN186
```

The RUN186.BAT file automatically invokes the emulator software and activates the emulator hardware.

Using Invocation Commands and Parameters

To invoke the software and activate the emulator explicitly, perform the following steps:

1. Invoke the emulator software from DOS:

   ```
   C> I186
   ```

   You can enter options following the keyword to customize your debug environment. (See Chapter 3, Section 3.3.2, Invocation Options, for more information.)

2. Define the emulator as the "tool" to be activated:

   ```
   hlt> DEFINE TOOL ICE186 = ICE186.CFG
   ```

3. Activate the emulator:

   ```
   hlt> ACTIVATE USING io-device ICE186
   ```

   The USING *io-device* specification overrides the I/O device defined in the ICE186.CFG configuration file. If you do not want to override the I/O device in the configuration file, enter "ACTIVATE ICE186" only. (See the ACTIVATE entry in this chapter.)

Note that the commonly used LITERALLY abbreviations, are not defined. If you want these abbreviations you must define them with the DEFINE command.

Cross-References

ACTIVATE
APPEND / PUT
Chapter 3, Sections 3.3.1 through 3.3.4
DEFINE
EXIT

I/O Functions

Built-in functions for
input and output

Discussion

The I/O functions read strings and single characters from the standard input device (keyboard), or write to the standard output device (screen). These functions are patterned after the C programming language functions of the same name. Some of these functions take a format string as an argument. Case is significant for the format string characters, which include the following:

b	binary
B	Boolean
d	decimal
o	octal
x	hexadecimal
u	unsigned decimal
c	single character
s	string
e	floating-point or double-precision floating-point: [-]m.nnnnE[-]xx
f	floating-point or double-precision floating-point: [-]mmm.nnn
g	floating-point or double-precision floating-point value using either f or e format, whichever takes less space without sacrificing full precision.

The input functions return EOF on end-of-file or on a read error. The output functions return EOF on a write error. The following I/O functions are discussed in this section:

GETCHAR	PUTS
GETS	SCANF
PRINTF	SPRINTF
PUTCHAR	SSCANF

GETCHAR Function

The GETCHAR function returns the next character from the keyboard. The syntax is as follows:

```
[obj-identifier =] GETCHAR ()
```

Where:

obj-identifier — specifies the debug object to which the function's return value (of type INT2) is assigned. If *obj-identifier* is not specified, the return value is displayed (in ASCII format) on the next line of the screen.

The following example illustrates the GETCHAR function:

```
hlt> DEFINE INT2 ans
hlt> DEFINE CHAR cvar
hlt> ans = GETCHAR ()
a                                                    /* user input */
hlt> ans
0061H
hlt> GETCHAR ()
'\n'
hlt> cvar = GETCHAR ()
a                                                    /* user input */
hlt> cvar
'a'
hlt>
```

GETS Function

The GETS function gets a string from the keyboard. The GETS function returns a null-terminated string from the keyboard to the address specified. The syntax is as follows:

```
GETS (addr-expr)
```

Where:

addr-expr — specifies an address (e.g., an NSTRING or LSTRING variable referenced with the & operator).

The following example illustrates the function GETS:

```
hlt> DEFINE NSTRING ns
hlt> GETS (&ns)
This is my string
hlt> ns
"This is my string"
hlt>
```

PRINTF Function

The PRINTF function displays formatted expressions on the screen. The PRINTF function formats any arguments in *expr* according to the format specified by *string-expr* and displays the results on the screen. The syntax is as follows:

```
PRINTF (string-expr [,expr [,...]] )
```

Where:

string-expr	specifies a quoted string of characters which specifies the format of the display. The format can contain two types of characters: ordinary characters and conversion specification characters.
expr	specifies one or more numbers or expressions corresponding to the conversion specification characters in *string-expr*.

Conversion specifications begin with the percent character (%), followed by an optional conversion modifier and end with a conversion character.

The conversion modifier can include:

- `-` A minus sign to signify left justification.
- `n` A number specifying a minimum field width.
- `.` A period that separates the field width from the next number specifying the maximum number of characters to be printed, or the number of digits to be printed to the right of a decimal point.
- `L` The letter "L" to signify a long integer.

In addition to these standard field template options, a maximum field width may be specified for all conversion types by a comma followed by the desired maximum width. If the maximum width is less than the minimum width, the maximum is assumed to be the same as the minimum. If the converted object is too large to fit in the maximum field width, the entire field is filled with asterisks. This operation always occurs except when fields generated with % signs are truncated to fit.

In addition to the standard C programming language conversion characters d, o, x, u, c, s, e, f, g, two other (b and B) are supported.

- `b` causes a corresponding argument to be displayed as binary notation.
- `B` causes a corresponding argument to be displayed as boolean (TRUE or FALSE).
- `c` causes a corresponding argument to be displayed as a single character.
- `d` causes a corresponding argument to be displayed as a decimal number.
- `e` causes a corresponding argument to be displayed in exponential notation; one digit is always displayed before the decimal point; the number of digits displayed after the decimal point defaults to six unless a precision field is specified.
- `f` causes a corresponding argument to be displayed in floating-point format; six digits are displayed after the decimal point unless a precision field is specified.

g	causes a corresponding argument to be displayed using either e or f format depending upon which takes less space without sacrificing full precision.
o	causes a corresponding argument to be displayed as an octal number.
s	causes a corresponding argument to be displayed as a character string; the argument must be terminated by a null character, unless a precision field is specified to indicate the number of characters to be displayed.
u	causes a corresponding argument to be displayed as an unsigned decimal number.
x	causes a corresponding argument to be displayed as a hexadecimal number.

The PRINTF function accepts the following escape characters.

\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	tab
\\	backslash
\"	double quote
nnn	character value; where *nnn* is a three-digit octal number that represents the ASCII value of the character. This value enables characters that may not be directly available from the keyboard to be inserted into a character string.
xnn	character value; where *nn* is a two-digit hexadecimal number that represents the ASCII value of the character. (The *x* indicates a hexadecimal number follows.) This value enables characters that may not be directly available from the keyboard to be inserted into a character string.

PRINTF Examples

1. The following example prints a simple message to the screen (\n is the escape character for a new line).

```
hlt> PRINTF("This is my message.\n")
This is my message.
hlt>
```

2. The following example uses character strings to print a date.

```
hlt> DEFINE NSTRING date = "Saturday"
hlt> DEFINE ORD1 day = 3
hlt> PRINTF ("Today is %s, the %drd of July.\n",date,day)
Today is Saturday, the 3rd of July.
hlt>
```

3. The following example prints a message with an audible beep, audible (007 octal is the ASCII code for the beep).

```
hlt> PRINTF("\007ATTENTION: Emulation has stopped\n")
(beep) ATTENTION: Emulation has stopped
hlt>
```

PUTCHAR Function

The PUTCHAR function displays a character on the screen. The PUTCHAR function appends the character specifier by *char-expr* to the standard output. The syntax is as follows:

PUTCHAR (*char-expr*)

Where:

string-expr specifies a quoted character or an expression which evaluates to a character.

The following example illustrates the function PUTCHAR (\n is the escape character for a new line):

```
hlt> DEFINE CHAR cvar = 'a'
hlt> PUTCHAR (cvar); PUTCHAR ('\n')
a
hlt>
```

PUTS Function

The PUTS function displays a string on the screen. The PUTS function appends the null-terminated string specified by *string-expr* to the standard output. The syntax is as follows:

```
PUTS (string-expr)
```

Where:

string-expr	specifies an NSTRING or LSTRING variable, string constant, or an expression which evaluates to a string.

The following example illustrates the function PUTS (\n is the escape character for a new line):

```
hlt> DEFINE NSTRING date = "6/2/53\n"
hlt> PUTS (date)
6/2/53
hlt> PUTS ("string constant \n")
string constant
hlt>
```

SCANF Function

The SCANF function reads data from the keyboard into any arguments designated by *addr-expr*, according to the format specifications in *string-expr*. The return value indicates the number of characters successfully read. The syntax is as follows:

```
[obj-identifier =] SCANF (string-expr, addr-expr [,...])
```

Where:

obj-identifier	specifies the debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.

string-expr specifies a quoted string of characters which specifies the format of the display. The format can contain two types of characters: ordinary characters and conversion specification characters.

addr-expr specifies one or more addresses (e.g., an NSTRING or LSTRING variable referenced with the & operator). These addresses, which correspond to the conversion specification characters in *string-expr*, specify the destination strings of the conversion input.

The following example illustrates the function SCANF:

```
hlt> DEFINE INT2 ans
hlt> DEFINE INT2 in2
hlt> DEFINE CHAR cvar
hlt> ans = SCANF ("%c", &cvar)
b                                                    /* operator input */
hlt> ans
0001H
hlt> cvar
'b'
hlt>
hlt> BASE = 10T
hlt> SCANF("%c %d", &cvar, &in2)
a 123                                                /* operator input */
2T
hlt> cvar
'a'
hlt> in2
123T
hlt>
```

SPRINTF Function

The SPRINTF function performs format conversions on data and stores the result in memory. The SPRINTF function formats the expressions in *expr* according to the format specifications in *string-expr* and stores the result in the string variable referenced by *addr-expr*. The syntax is as follows:

```
SPRINTF (addr-expr, string-expr, expr [,...] )
```

Where:

addr-expr	specifies an address (e.g., an NSTRING or LSTRING variable referenced with the & operator).
string-expr	specifies a quoted string of characters which specifies the format of the display. The format can contain two types of characters: ordinary characters and conversion specification characters.
expr	specifies one or more numbers or expressions corresponding to the conversion specification characters in *string-expr*.

The following example illustrates the function SPRINTF:

```
hlt> DEFINE NSTRING date
hlt> SPRINTF (&date, "%s", "1987")
hlt> date
"1987"
```

SSCANF Function

The SSCANF function reads data from the string variable referenced by *addr-expr1* into the destination string variable referenced by *addr-expr2* according to the format specification characters in *string-expr*. The syntax is as follows:

```
[obj-id =] SSCANF (addr-expr1, string-expr, addr-expr2 [,...])
```

Where:

obj-id	specifies the debug object to which the function's return value is assigned. If *obj-id* is not specified, the return value is displayed on the next line of the screen.
addr-expr1	specifies an address (e.g., an NSTRING or LSTRING variable referenced with the & operator).
string-expr	specifies a quoted string of characters which specifies the format of the display. The format can contain two types of characters: ordinary characters and conversion specification characters.
addr-expr2	specifies one or more addresses (e.g., an NSTRING or LSTRING variable referenced with the & operator). These addresses, which correspond to the conversion specification characters in *string-expr*, specify the destination strings of the conversion input.

The following example illustrates the function SSCANF:

```
hlt> DEFINE INT2 ans
hlt> DEFINE NSTRING sum = "6.325"
hlt> DEFINE CHAR cvar
hlt> ans = SSCANF (&sum, "%c", &cvar)
hlt> ans
0001H
hlt> cvar
'6'
```

Cross-References

C: A Reference Manual
DEFINE
<expr>
Functions
PROC

IORDY

Enables an emulator time-out when an I/O access takes more than one second

SYNTAX

```
IORDY  [= bool-expr]
```

Where:

IORDY	if entered without an optional expression, displays the current IORDY setting.
bool-expr	is any expression that evaluates to TRUE (non-zero) or FALSE (zero).

Default

TRUE

DISCUSSION

Use the IORDY tool variable to enable and disable I/O access time-outs during emulation. (These time-outs apply to I/O access cycles involving USER-mapped I/O ports only; they do not apply to ICE-mapped I/O ports.)

To enable I/O access time-outs, set IORDY to TRUE. The emulator will stop and display a break message if the processor READY line remains inactive for more than one second during an I/O access cycle to a USER-mapped I/O port.

To disable I/O access time-outs, set IORDY to FALSE.

You can display or change the IORDY setting at any time.

Example

This example displays the current setting of IORDY, then resets the variable.

```
hlt> IORDY
TRUE
hlt> IORDY=0
```

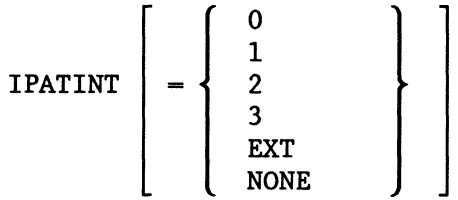
Cross-References

BUSACT
CAUSE
<expr>
MEMRDY
TOOLVAR

IPATINT

Selects the interrupt source for
iPAT interrupt latency measurements

Syntax



Where:

IPATINT	is the keyword. If entered without an optional parameter, it displays the current IPATINT setting.
0, 1, 2, or 3	specifies that the source for the interrupt is the processor INT0, INT1, INT2, or INT3 signal, respectively.
EXT	specifies that the interrupt is derived via an external signal applied to the INT pins on the emulator probe processor module.
NONE	specifies that no interrupt source is selected.

Default

NONE

Discussion

Use the IPATINT tool variable to specify the source of the interrupt used in iPAT interrupt latency measurements.

One of the features of the Intel iPAT is that it can be used to measure hardware-interrupt-to-software-event latency times. When using the emulator to generate the hardware interrupt, you have two choices. You can use one of the four processor INT signals (0, 1, 2, or 3), or you can use an external signal applied to the emulator probe processor module. (See the *ICE™-186/188 In-Circuit Emulator Installation Supplement* for instructions on applying the external signal.)

The emulator outputs tag words to the iPAT to indicate the occurrence of an interrupt event. An interrupt event occurs when there is a low-to-high transition on the selected input.

You can display the setting of IPATINT at any time, but can change it only when the emulator is in Halt mode.

For more information about iPAT interrupt latency measurements, refer to the Intel *iPAT™ Analyst User's Guide.*

Example

This example sets the source of the interrupt to processor signal INT0.

```
hlt> IPATINT = 0
hlt>
```

Cross-References

ICE™-186/188 In-Circuit Emulator Installation Supplement
iPAT™ Analyst User's Guide
TOOLVAR

ISTEP

Steps through a program by machine-language instructions

Syntax

```
ISTEP  [FROM address]  [step-size]
```

Where:

ISTEP	executes one or more machine-language instructions at a time.
FROM *address*	specifies a starting address where ISTEP is to begin. The default starting address is the current execution pointer.
step-size	is an unsigned integer expression specifying the number of instructions in a step. The default *step-size* is 1 instruction; the maximum is 255T instructions.

Discussion

Use the ISTEP command to single-step through the machine-language instructions of your program.

You can control the starting address and size of the step by using the FROM *address* and *step-size* parameters, respectively. The FROM *address* parameter specifies the starting address of the step. The *step-size* parameter specifies the size of the step in terms of instructions. You can specify from 1 to 255T instructions per step.

Note that the ISTEP command does not step through instructions that move data into a segment register (i.e., CS, DS, ES, or SS). The move instruction and its immediately following instruction are considered one instruction.

When you use the ISTEP command, the step breaks are the only breakpoints acknowledged by the emulator. The Activity Monitor trace is turned off, and the emulator ignores any breakpoints established by the GO command.

You can use the ISTEP command only when the emulator is in Halt mode.

Example

The following example uses the ISTEP command to step through five machine-language instructions beginning at 1100:4H. Note that instructions at 1100:000aH and 1100:0012H are missing due to the MOV and segment register restriction.

```
hlt> ISTEP FROM 1100:4H 5
1100:0004H    fa            CLI
1100:0005H    2e8e160000    MOV   SS,WORD PTR CS:0000H
1100:000dH    2e8e1e0200    MOV   DS,WORD PTR CS:0002H
:INTERFACE.PSTART_
1000:00a0H    e95dff        JMP   $-00a0H ;A=0000H NEAR
:MESSG.MAIN_#92
1000:0000H    56            PUSH  SI
hlt>
```

Cross-References

LSTEP
PSTEP
STEP

LIST

Records a debug session
to a file

Syntax

```
LIST  filename  [ APPEND    ]
                [ OVERWRITE ]
```

Where:

LIST	if entered without options, displays the current list filename.
filename	is the name (with optional path) of the disk file where the debug session log (or listing) is saved.
APPEND	causes log information from the current session to be added to the end of an existing file.
OVERWRITE	overwrites *filename* with the current log information.

Discussion

Use the LIST command to make a log of a debugging session. Log files (or list files) record all emulator command interaction with the terminal during a debug session and save it in the specified file. You can append data to an existing file, overwrite an existing file, or create a new log file by specifying a unique filename.

To create a new log file, enter the command "LIST *filename*". You should close a log file before opening a new one. If the log file is opened successfully, the emulator prompt will appear without any messages. If you enter EXIT without entering NOLIST, the log file is automatically closed.

To discontinue sending data to the log file, enter NOLIST. If you attempt to record data to a new log file, the currently open log file is automatically closed.

A log file can be opened at software invocation time by using the LOGFILE = *pathname* specification in the installation utility. If you enter the NOLIST command during the debug session, or if you enter EXIT without entering NOLIST, the file is closed.

NOTE

Output from a host operating system command (preceded by @) is not entered into the log file.

Example

The following example opens a list file "session.log" in the "tmp" directory.

```
hlt> LIST c:\tmp\session.log
hlt>
```

Cross-References

EXIT
NOLIST

LOAD

Loads an object file
into mapped memory

Syntax

```
                  [ NOSYMBOLS    ]
LOAD pathname     | NOCODE       |
                  | HEX          |
                  | APPEND       |
                  [ AT address   ]
```

Where:

LOAD	is the command keyword that loads the file. Intel 8086 absolute OMF, SAVE-formatted, and Intel 8086 hexadecimal files are supported.
pathname	is the fully qualified reference to the load file.
NOCODE	specifies that code and data are not to be loaded.
NOSYMBOLS	specifies that symbolic information (addresses and type definitions) is not to be loaded.
HEX	specifies that the object file is in Intel 8086 hexadecimal format. This file contains only code and data. No symbolic information is present.
APPEND	specifies that symbolic table files stay attached to the data base during loading.
AT *address*	is available only for files created with the SAVE command. It specifies the starting address in memory to load the file.

Discussion

The LOAD command loads code and data information into mapped memory and processes symbolic information if required.

The LOAD command supports three types of file formats:

- Intel 8086 absolute Object Module Format (OMF).

- SAVE format (files created by the SAVE command).
- Intel 8086 hexadecimal object file format.

Files in 8086 absolute OMF include symbolic information if the modules were compiled without optimization and with the compiler switches for debug information and symbols on. Files in the SAVE format contain no symbolic information.

The first time you load a file, a separate file containing symbolic information is created (unless the NOSYMBOLS options is used). This separate file is called the symbol table file. The symbol table file is "attached" to the emulator database and can be accessed via the emulator symbolic reference feature.

In subsequent load operations, the symbol table file is updated only if the loader detects that the creation date of the symbol table file is older than the creation date of the load file (indicating that the load file has been changed since the symbol table file was created). Symbolic information encountered in the load file is skipped if the symbol table file does not need updating, resulting in a faster load time.

You can force the creation of a new symbol table file by deleting the existing one. The next time the file is loaded, a new symbol table file will be created and "attached" to the emulator database.

A symbol table file is automatically removed from the emulator database when you load another file. You can explicitly remove a symbol table file from the database by using the REMOVE command, by LOADing with NOSYMBOLS, by executing the RESET ICE command, or by DEACTIVATEing the emulator.

To retain symbol table files attached to the emulator database, use the APPEND option. The existing symbol table files remain attached to the database and can be added to the current load. Multiple symbol table file lookup is then possible.

The symbol table file extension is ".sym". The root name of a given symbol table file is the same as the root name of the file being loaded (e.g., "sensor.sym" for load file "sensor.912"). If you are working with several programs (sensor.912, sensor.913, etc.), be sure the root names of the load files are unique to avoid the problem of having the same symbol table name for two unique files.

During the load operation, a period (.) is displayed on the screen for every load record processed. If starting address or register information is encountered, it is also displayed.

NOTE

It is recommended that files be copied from the floppy disk to the appropriate directory on the hard disk drive to minimize the risk of activating DOS error routines and to avoid performance degredation.

CAUTION

When attempting to load a file from a floppy disk drive, be sure to prevent any conditions that cause DOS error routines to be activated (such as incompatibility disk density errors or leaving the disk drive latch ajar). There are no software facilities to prevent loss of data if the emulator software is interrupted by DOS to handle error routines.

Examples

1. This example loads the file "c:\c186\tool" with all debug information.

```
hlt> LOAD c:\c186\tool
```

2. This example loads only the symbols for "c:\c186\tool".

```
hlt> LOAD c:\c186\tool NOCODE
```

3. These examples display the various symbol types in the symbol table file.

```
hlt> DIR FULL PUBLIC SYMBOLFILE//
...displays all global symbols in the symbol file...

hlt> DIR FULL MODSYMBOL SYMBOLFILE//
...displays all modules in the symbol file...

hlt> DIR FULL LINE SYMBOLFILE//
...displays all line information in the symbol file...
```

Cross-References

DIR
MAP
SAVE
SYMBOLIC
Symbolic References

LSTEP

Steps through a program by high-level language statements

Syntax

```
LSTEP [FROM address] [step-size] [ PROC  ] [ DISP  ] [NPROMPT]
                                 [ NPROC ] [ NDISP ]
```

Where:

LSTEP	steps through your program by high-level language statements. The line numbers for the statements are assigned to your program when it is compiled by an Intel compiler.
FROM *address*	specifies a starting address where LSTEP is to begin. The default starting address is the current execution pointer.
step-size	is an unsigned integer expression specifying the number of statements to be included within a step. The default *step-size* is 1 statement; the maximum is 255T statements.
PROC | NPROC	specifies how statements within a procedure or function call are to be counted for *step-size*. If PROC is chosen, all statements within a given procedure or function call are counted as a single statement. If NPROC is chosen, the statements are counted individually. NPROC is the default parameter.
DISP | NDISP	controls the display of the statement to be executed. The default is DISP, which allows display.
NPROMPT	terminates the command after one step.

Discussion

Use the LSTEP command to step through your program by high-level language statements.

You can control the starting address and size of the steps by using the FROM *address* and *step-size* parameters, respectively. The FROM *address* parameter specifies the starting address of the first step; ongoing steps start from where the previous step left off. The *step-size* parameter specifies the size of the steps in terms of statements. You can specify from 1 to 255T statements per step.

The NPROMPT parameter allows you to specify whether the Step mode is ongoing or complete after the initial LSTEP execution. If NPROMPT is not entered, the Step mode is ongoing. After each step is executed, a screen message asks if you want to step again or return to Halt mode. Press the <Enter> key to step again; press the E key to return to Halt mode.

When you use LSTEP, the step breaks are the only breakpoints acknowledged by the emulator. The Activity Monitor trace is turned off, and the emulator ignores any breakpoints established by the GO command.

You can execute the LSTEP command only when the emulator is in Halt mode.

LSTEP operates by executing repeated ISTEP instructions. After each ISTEP, the emulator exits emulation to evaluate whether the instruction corresponds to a line number. If it does not, another ISTEP is performed. Thus, LSTEP is not a full speed emulation command.

Exercise caution when using LSTEP with code that contains delay loops because they will take too long to execute for the LSTEP to be practical. In addition, LSTEP does not ignore interrupts, so using LSTEP with real-time applications is of limited use. When an interrupt occurs during the execution of an LSTEP command, the repeated ISTEPs will service the interrupt service routine. If this routine is lengthy, or does not contain line number information, the LSTEP will take an excessive amount of time to execute.

LSTEP (continued)

One way to ignore the interrupts is to set the Interrupt Flag (IFL) to zero before each LSTEP. This clears any pending interrupts. The following PROC will automatically clear pending interrupts:

```
hlt> DEFINE PROC clrint
hlt>> {
hlt>> IFL=0H
hlt>> LSTEP
hlt>> }
hlt>
```

Example

This example steps through two statements using the LSTEP command.

```
hlt> LSTEP FROM :messg.main_#92 2
:messg.main_#92     (1000:0000H)
:messg.main_#96     (1000:0005H)
Press E to exit, <Enter> to repeat.
?
```

At the ? prompt, you can press <Enter> to continue stepping or press E to exit the Step mode.

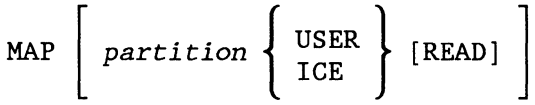
Cross-References

CALLSTACK
ISTEP
Prompts
PSTEP
STEP

MAP

Determines whether memory accesses are mapped to USER or ICE RAM

Syntax



Where:

MAP	if entered without optional parameters, displays the current memory map.
partition	is a single physical address, an expression that evaluates to a single physical address, or a range of physical addresses. The range is specified as either *address* TO *address* or *address* LENGTH *number-of-bytes*. The partition is mapped in 4K-byte blocks.
USER	specifies that *partition* is mapped to the prototype.
ICE	specifies that *partition* is mapped to the emulator.
READ	indicates that the *partition* is read-only. Emulation halts if a write to the *partition* occurs. Omitting READ allows read/write access to the *partition*.

Default

All memory mapped to USER with read/write access.

Discussion

The emulator provides 128K bytes of zero wait-state RAM. The MAP command allows you to specify whether a memory access uses USER (default) or ICE memory. It also allows you to specify whether the memory access is read/write (default) or read-only.

Memory can be mapped in 4K-byte blocks, up to a maximum of 128K bytes (32 4K-byte blocks). The starting addresses of the blocks fall on 4K-byte boundaries (e.g., 0000HP, 1000HP, 2000HP, etc.). If you specify a starting address that falls between two of the 4K-byte boundaries, the starting address is rounded down to the lower 4K-byte boundary (e.g., 1234HP becomes 1000HP). The ending address in a LENGTH specification is rounded up to the nearest 4K-byte boundary.

The partition parameter specifies the memory block to be mapped, while USER and ICE specify the location of the memory (prototype or emulator RAM, respectively). The READ parameter specifies that access to partition is read-only; otherwise, the access is read/write.

To use the READ (read-only) option for addresses mapped to ICE, you must program any chip-select register associated with the address partition to recognize external READY. You do this by setting the R2 bit of the particular chip-select register to 0.

There are five chip-select registers located in the processor Peripheral Control Block. Three of the registers (MMCS, LMCS, and UMCS) are always used to control memory. The other two registers (MPCS and PACS) can be used to control either memory or I/O devices.

You can access the chip-select registers by using the register access commands (see the Register Access entry in this chapter). The emulator keywords for the chip-select registers are CSCTRL5 (MPCS), CSCTRL4 (MMCS), CSCTRL3 (PACS), CSCTRL2 (LMCS), and CSCTRL1 (UMCS).

NOTE

The emulator halts and memory writes are blocked if a read-only violation occurs in memory mapped to ICE. The emulator halts and memory writes are not blocked if a read-only violation occurs in memory mapped to USER.

Consecutive MAP commands are cumulative. Use the RESET MAP command to clear the current mapping.

The current memory map can be displayed at any time by entering MAP without parameters. The map can be changed only when the emulator is in Halt mode.

Examples

1. This example displays the current memory map.

```
hlt> MAP
00000HP   length   100000H   USER    READ/WRITE
hlt>
```

2. This example maps 64K bytes starting at 0H to ICE memory, read only.

```
hlt> MAP 0HP LENGTH 64K ICE READ
00000HP   length   10000H   ICE     READ ONLY
10000HP   length   f0000H   USER    READ/WRITE
hlt>
```

3. This example illustrates the result of not specifying the "TO *address*" or the "LENGTH *number-of-bytes*" part of the *partition*. (Note that the MAP commands are cumulative. This example affects the memory map established in Example 2.)

```
hlt> MAP 0HP ICE
00000HP   length     1000H   ICE     READ/WRITE
01000HP   length     f000H   ICE     READ ONLY
10000HP   length    f0000H   USER    READ/WRITE
hlt>
```

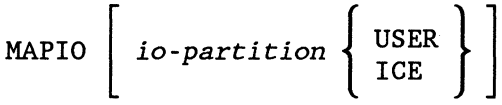
Cross-References

Memory Access	Register Access
Partition	RESET
PCB	VERIFY

MAPIO

Determines whether I/O ports are mapped to USER or ICE

Syntax



Where:

MAPIO	if entered without optional parameters, displays the current I/O map.
io-partition	an expression that evaluates to a range of I/O port addresses. The range is specified as either *port* TO *port* or *port* LENGTH *number-of-ports*. The partition is mapped in 4K-byte blocks.
USER	specifies that *io-partition* is mapped to the prototype.
ICE	specifies that *io-partition* is mapped to the emulator. I/O activity in this *io-partition* is simulated by the emulator.

Default

All ports mapped to USER.

Discussion

Use the MAPIO command to map I/O port addresses to the prototype (USER) or the emulator (ICE).

If you map I/O port addresses to ICE, you must program any chip-select register associated with the *io-partition* to recognize external READY. You do this by setting the R2 bit of the particular chip-select register to 0.

There are five chip-select registers located in the processor Peripheral Control Block. Two of these register, MPCS and PACS, can be used to control I/O devices.

You can access the chip-select registers by using the emulator register access commands (see the Register Access entry in this chapter). The emulator keywords for the two I/O chip-select registers are CSCTRL5 (MPCS) and CSCTRL3 (PACS).

When you map I/O port addresses to ICE, the emulator simulates I/O accesses during emulation. It displays data values written from ports during I/O read cycles, and prompts you for data value inputs during I/O write cycles (see I/O Simulation later in this discussion).

You can map I/O port addresses to ICE in 4K-byte blocks, up to a maximum of 64K bytes (16 4K-byte blocks). The starting addresses of the blocks fall on 4K-byte boundaries (e.g., 0000H, 1000H, 2000H, etc.). If you specify a starting address that falls between two of the 4K-byte boundaries, the starting address is rounded down to the lower 4K-byte boundary (e.g., 1234H becomes 0000H). The ending address in a LENGTH specification is rounded up to the nearest 4K-byte boundary.

The *io-partition* parameter specifies the block to be mapped, while USER and ICE specify the block location.

Consecutive MAPIO commands are cumulative. Use the RESET MAPIO command to clear the current I/O port map.

I/O Simulation

The emulator simulates I/O accesses to ICE-mapped I/O ports. It displays the data values written from ports during I/O read cycles, for example:

```
hlt> GO FROM 0:0
   34H written to IO Port f000H
```

It also prompts you for data value inputs during I/O write cycles, for example:

```
hlt> GO FROM 0:0
  Port f000H requests byte input (enter value):
```

When a request appears for data input, you have two choices. You can enter a value or enter the HOLDIO command. If you enter a value, the emulator resumes emulation. If you enter the HOLDIO command, the emulator suspends both the pending request and emulation. For example:

```
hlt> GO FROM 0:0
  Port f000H requests byte input (enter value): HOLDIO
  Emulator Status:  hung pending mapped IO request.
Use RELEASEIO to service request.
  Activity Monitor Status:   Running.
IO?+>
```

While the I/O request is pending, you can enter any commands that do not directly access the processor. The processor itself is hung, awaiting data input. In addition, the emulator does not service interrupts, regardless of the setting of ENI.

To release the pending I/O request, enter the RELEASEIO command. The emulator will immediately display the request, shown as follows:

```
IO?+> RELEASEIO
  Port f000H requests byte input (enter value): 12H
  Emulator Status:  running user code.
  Activity Monitor Status:  Running.
emu+>
```

CAUTION

All I/O outputs, whether through USER- or ICE-mapped I/O ports, appear on the pins of the user probe. Mapping to ICE only prevents the emulator from receiving the input values from the prototype system. Ensure that any I/O accesses during emulation cannot harm prototype hardware or software.

Example

This example maps an *io-partition* to 0F000 ICE.

```
hlt> MAPIO 0F000H ICE
0000H   length   f000H   USER
f000H   length   1000H   ICE
hlt>
```

Cross-References

HOLDIO
Partition
PCB
Prompts
RELEASEIO
Register Access
RESET

Math Functions

Built-in mathematical functions

Discussion

Three classes of math functions are available: ABS function, power functions, and trigonometric functions.

The ABS function returns the absolute value of its argument. The power functions are EXP, LOGE, LOG10, POW, SQRT. These have arguments of type REAL8 and return a value (*obj-identifier*) of type REAL8. The trigonometric functions are SIN, COS, TAN, ASIN, ACOS, ATAN, ATAN2. They have radian arguments of type REAL8 and return a value (*obj-identifier*) of type REAL8.. The magnitude of each argument in trigonometric functions should be checked by the caller to make sure the result is meaningful.

The following math functions are discussed in this section:

ABS	COS	SIN
ACOS	EXP	SQRT
ASIN	LOG10	TAN
ATAN	LOGE	
ATAN2	POW	

Absolute Value Function

ABS Function

The ABS function returns the absolute value of an expression. The syntax is as follows:

```
[obj-identifier =] ABS (expr)
```

Where:

obj-identifier	specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr	specifies a number or an expression.

Exponential and Logarithmic Functions

EXP Function

The EXP function returns the exponential function of an expression; that is, the number e raised to the *expr* power, where e is the base of the natural logarithms. EXP returns an invalid value when the correct return value would overflow. The syntax is as follows:

```
[obj-identifier =] EXP (expr)
```

Where:

obj-identifier	specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr	specifies a number or an expression.

LOGE Function

The LOGE function returns the natural logarithm of an expression . The syntax is as follows:

```
[obj-identifier =] LOGE (expr)
```

Where:

obj-identifier	specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr	specifies a number or an expression.

expr specifies a number or an expression.

LOG10 Function

The LOG10 function returns the base 10 logarithm of an expression . The syntax is as follows:

```
[obj-identifier =] LOG10 (expr)
```

Where:

obj-identifier specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.

expr specifies a number or an expression.

POW Function

The POW function raises the value specified by *expr* to the power specified by *pow-expr*. POW returns an invalid value when the correct value would overflow. The return value is of type REAL8. The syntax is as follows:

```
[obj-identifier =] POW (expr, pow-expr)
```

Where:

obj-identifier specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.

expr specifies a number or an expression that is to be raised to the power of *pow-expr*.

pow-expr specifies a number or an expression that is the power *expr* is to be raised to.

SQRT Function

The SQRT function returns the square root of an expression . SQRT returns 0 (zero) when *expr* is negative. The syntax is as follows:

```
[obj-identifier] = SQRT (expr)
```

Where:

obj-identifier	specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr	specifies a number or an expression.

Trigonometric Functions

SIN Function

The SIN function returns the sine of a radian expression . The syntax is as follows:

```
[obj-identifier] = SIN (expr)
```

Where:

obj-identifier	specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr	specifies a number or an expression evaluated in radians.

COS Function

The COS function returns the cosine of a radian expression . The syntax is as follows:

```
[obj-identifier =] COS (expr)
```

Where:

obj-identifier	specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr	specifies a number or an expression evaluated in radians.

TAN Function

The TAN function returns the tangent of a radian expression . The syntax is as follows:

```
[obj-identifier =] TAN (expr)
```

Where:

obj-identifier	specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr	specifies a number or an expression evaluated in radians.

ASIN Function

The ASIN function returns the arc sine of an expression in the range -pi/2 to pi/2. If the expression is greater than 1 or less than -1, ASIN returns the value 0 (zero). The syntax is as follows:

```
[obj-identifier =] ASIN (expr)
```

Where:

obj-identifier	specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr	specifies a number or an expression.

ACOS Function

The ACOS function returns the arc cosine of an expression in the range 0 to pi. If the expression is greater than 1, ACOS returns the value 0 (zero). The syntax is as follows:

```
[obj-identifier =] ACOS (expr)
```

Where:

obj-identifier	specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr	specifies a number or an expression.

ATAN Function

The ATAN function returns the arc tangent of an expression in the range -pi/2 to pi/2. The syntax is as follows:

```
[obj-identifier =] ATAN (expr)
```

Where:

obj-identifier	specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr	specifies a number or an expression.

ATAN2 Function

The ATAN2 function returns the second arc tangent of *expr2* divided by *expr1* in the range -pi to pi. The syntax is as follows:

```
[obj-identifier =] ATAN2 (expr1, expr2)
```

Where:

obj-identifier	specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr1 and expr2	specify numbers or expressions.

Cross-References

<expr>
Functions

Memory Access

Displays or writes to a location in memory

Syntax

```
data-type partition [= { expr [,...]
                       { data-type partition } ]
```

Where:

data-type specifies the data type used to access memory. (Refer to the <data type> entry in this chapter for more information.)

partition is a range of addresses. A *partition* can be any one of the following (see the Partition entry in this chapter).

```
{ addr
{ addr TO addr
{ addr LENGTH expr }
```

expr is an expression (see <expr> in this chapter). More than one expression can be entered by using a comma to separate them.

Discussion

Memory access commands allow you to read values from or write values to any memory partition by any emulator data type. You can also copy from one partition to another if the data types are the same. For memory read commands, the requested data is displayed in the current base, preceded by the address, using the specified *data-type* as a template. When multiple items are requested, as many items as will fit on a line are displayed, followed by a new line and address of the first item on the line, until all items have been displayed.

Memory Access (continued)

Addresses are always displayed in hexadecimal. If the *data-type* is ORD1 (or byte), the ASCII representation of the data is shown on the right side of the screen with non-printing characters displayed as a period (.). Memory access commands use CS as the default segment in segmented addresses.

If the length of a write is greater than the length of the data supplied (the number of expressions), memory is filled using the supplied data pattern until the write length has been reached. If source and destination addresses overlap, the write is done such that the source memory image is preserved in the destination. However, if the length of the data pattern exceeds the partition, an error results. The only exception to this occurs when the partition on the left side of the equation is a single address value. In this case, the values are written to memory beginning with the address given.

The setting of the FASTBREAK tool variable determines whether program memory can be accessed during emulation. If the program is being executed and FASTBREAK is TRUE (non-zero), the emulator pauses briefly while memory is accessed, and then resumes emulation. If FASTBREAK is FALSE (zero), memory accesses are not allowed during emulation.

Address syntax and addressable segment boundaries are described in detail under the Address Translation entry in this chapter. Partition syntax is described in detail under the Partition entry.

Examples

1. This example displays a 4-byte ordinal value at address 21:11H. It also displays a partition of 2-byte ordinal values starting at the current execution point.

```
hlt> BASE=16T
hlt> ORD4 21:11
0021:0011H    14a146d7
hlt> ORD2 $ LENGTH 10H
0021:0010H    46d7   14a1   e700   a148   0016   4ae7   1251   e700
0021:0018H    8842   5de5   8bc3   4dec   554d   0a9a   5900   9a00
```

2. This example writes the three ORD2 values 1234, 0abcd, and 3 to the memory location starting at 40:04H.

```
hlt> ORD2 40:04H = 1234,0ABCD,3
```

3. This example displays the 4-byte ordinal at offset address 0011H.

```
hlt> ORD4 11H
0011H     14a147d7
```

4. This example fills 128T memory locations (of type ORD1) starting at address 0HP (physical) with the value stored in location 100HP.

```
hlt> ORD1 0HP LENGTH 128T = ORD1 100HP
```

Cross-References

Address Translation
BTHRDY
<data type>
FASTBREAK
MAP
MEMRDY
Partition
PCB
Register Access
VERIFY

MEMRDY

Allows an emulator time-out
based on memory access time

Syntax

MEMRDY [= *bool-expr*]

Where:

MEMRDY	if entered without an optional expression, displays the current MEMRDY setting.
bool-expr	is any expression that evaluates to TRUE (non-zero) or FALSE (zero).

Default

TRUE

Discussion

Use the MEMRDY tool variable to enable and disable memory access time-outs. You can change the setting of the variable at any time.

Setting MEMRDY to TRUE enables memory access time-outs. The emulator will stop and display a break message whenever the processor READY line remains inactive for more than one second during a memory access cycle.

Setting MEMRDY to FALSE disables memory access time-outs.

Example

This example displays the current MEMRDY setting, then resets the variable.

```
hlt> MEMRDY
TRUE
hlt> MEMRDY=0
```

Cross-References

BTHRDY
CAUSE
<expr>
IORDY
Memory Access
TOOLVAR

Miscellaneous Functions

Built-in miscellaneous functions

Discussion

The miscellaneous functions consist of the following:

- Screen control functions: CLEAR, CLRTOBOT, CLRTOEOL, and MOVE.
- Date-and-time functions: TIME and CTIME.
- Random-number functions: SRAND and RAND.
- An execution pause function: SLEEP.

These functions are discussed in this section.

CLEAR Function

The CLEAR function clears the screen entirely and moves the cursor to the upper left corner of the screen. The syntax of the CLEAR function is as follows:

```
CLEAR ( )
```

CLRTOBOT Function

The CLRTOBOT function clears the screen from (but not including) the current line (the line where the CLRTOBOT command was entered) to the bottom of the window. The cursor goes to the line following the current line. The syntax of the CLRTOBOT function is as follows:

```
CLRTOBOT ( )
```

CLRTOEOL Function

The CLRTOEOL function clears the current line (the line from which the command was entered) from the cursor position to the right end of the line. The syntax of the CLRTOEOL function is as follows:

```
CLRTOEOL ( )
```

MOVE Function

The MOVE function moves the cursor to the designated row (counting from the top) and column (counting from the left) on the screen. The syntax of the MOVE function is as follows:

```
MOVE (row-expr, col-expr)
```

Where:

row-expr	specifies a number or an expression which specifies a row on the screen.
col-expr	specifies a number or an expression which specifies a column on the screen.

TIME Function

The TIME function gets the date and time in internal format (seconds since Greenwich Mean Time, January 1, 1970). TIME returns the value as a DWORD data type. The syntax is as follows:

```
[obj-identifier =] TIME ( )
```

CTIME Function

The CTIME function converts the output of the TIME function into a null-terminated ASCII string. The input expression (*expr*) is a value (such as one returned by TIME). The output string has the form "day month date hh:mm:ss year\n" (\n is the new-line escape character). The syntax is as follows:

```
[obj-identifier =] CTIME (expr)
```

Where:

obj-identifier	specifies a string data type to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.
expr	specifies a number or an expression.

The following example illustrates the function CTIME:

```
hlt> DEFINE DWORD now
hlt> DEFINE NSTRING today
hlt> now = TIME ()
hlt> CTIME (now)
"Wed Feb 04 16:26:07 1987"
hlt> today = CTIME (now)
hlt> today
"Wed Feb 04 16:26:17 1987"
hlt> CTIME (TIME())
"Wed Feb 04 16:26:27 1987"
```

SRAND Function

The SRAND function sets the starting point for generating a pseudo-random number using the RAND function. The RAND function can be set to a random starting point by calling SRAND with a *seed-expr* other than 1. Using 1 initializes a predictable response. The syntax is as follows:

SRAND (*seed-expr*)

Where:

seed-expr	specifies a number or an expression.

RAND Function

The RAND function returns a pseudo-random number of INT4 data type. If the SRAND function has been previously executed, the RAND function uses the output of the SRAND function as its seed expression. If the SRAND function has not been previously executed, the RAND function generates a less-random number. The syntax is as follows:

```
[ obj-identifier = ] RAND ( )
```

Where:

obj-identifier specifies a debug object to which the function's return value is assigned. If *obj-identifier* is not specified, the return value is displayed on the next line of the screen.

The following example illustrates the SRAND and RAND functions:

```
hlt> BASE = 10T
hlt> DEFINE INT4 card
hlt> SRAND (3)
hlt> card = RAND ()
hlt> card
16838T
hlt> RAND ()
5758T
```

SLEEP Function

The SLEEP function suspends execution of the current process for the approximate number of seconds specified by *expr*. The syntax of the SLEEP function is as follows:

```
SLEEP (expr)
```

Where:

expr specifies a number or an expression.

In the following example, the SLEEP function is used to suspend execution for approximately 5 seconds.

```
hlt> SLEEP (5)
hlt>
```

Cross-Reference

Functions

NAMEFRAME

Displays stack frame

Syntax

```
NAMEFRAME [= expr]
```

Where:

NAMEFRAME	if entered without an option, displays the current stack frame number.
expr	specifies a number or an expression which specifies a stack frame number.

Discussion

The NAMEFRAME parameter contains a value that represents the current stack frame. The content of a dynamic variable is stored in its procedure's activation frame on the stack. When making a transition from emulation to interrogation mode, NAMEFRAME is set to the most recent stack frame (frame number = 0).

The command CALLSTACK must be entered before the NAMEFRAME parameter can be interactively modified. By changing the value of NAMEFRAME, active dynamic variables can be referenced. When NAMEFRAME is modified, NAMESCOPE is automatically updated, thus changing the current symbolic scoping to the stack frame corresponding to *expr*.

NAMEFRAME (continued)

Example

1. Assume the following lexical scoping:

```
modx
  proca
    var1
  procb        <--NAMESCOPE
    var2
```

Assume the following stack frames. (The numbers represent stack frame numbers.):

```
procb      0 <--most recent
procb      1
proca      2
procb      3
```

With NAMEFRAME=1, *var2* references *var2* in procb's frame 1.

With NAMEFRAME=3, *var2* references *var2* in procb's frame 3.

Cross-References

$ (execution pointer)
CALLSTACK
<expr>
NAMESCOPE
Parameters

NAMEPATH

Determines search order for emulator objects

Syntax

NAMEPATH [= *expr*]

Where:

NAMEPATH	if entered without an option, displays the current namepath.
expr	specifies a string or an expression which specifies a specific quoted string of directory names.

Discussion

Use the NAMEPATH parameter to change the search order for objects. The default path is:

```
KEYWORD// PARAMETER// DYNAMIC// DEBUG// TOOLVAR// TOOLPTR// SYMBOL// PUBLIC//
```

The NAMEPATH parameter controls where the emulator software searches for object names. When an object name is specified without a directory, the NAMEPATH parameter is consulted to determine the order of the object directories to search. Override the default NAMEPATH search order by specifying an object name with a directory prefix:

```
DEBUG//ax                  /* where ax is a user-defined debug variable */

TOOLVAR//ax                  /* where ax is an emulator tool variable. */
```

Change the search order of directories by specifying a search path. The specification must be in quotes. Always include KEYWORD//, PARAMETER//, and TOOLVAR// in the path. KEYWORD// is the directory for the emulator software commands. PARAMETER// is the path for software parameters such as NAMEPATH. TOOLVAR// is the directory for tool-specific commands.

WARNING

If the KEYWORD//, PARAMETER//, and TOOLVAR// portions of the NAMEPATH parameter are inadvertently lost, then all object names must have prefixed directory names. Enter KEYWORD//EXIT to exit and re-invoke the emulator software.

Example

Change the emulator's search order. Spaces are the delimiters between directories. Use a backslash to continue commands over more than one line.

```
hlt> NAMEPATH = "DEBUG// KEYWORD// PARAMETER// DYNAMIC// TOOLVAR// \
hlt> TOOLPTR// SYMBOL// PUBLIC//"
hlt>
```

Cross-References

DIR
Object Hierarchy
Parameters

NAMESCOPE

Shortens symbolic referencing

Syntax

```
NAMESCOPE [= expr]
```

Where:

NAMESCOPE	without expr, displays the address and symbolic reference of the current symbolic scope.
expr	is an address or symbolic reference.

Discussion

The NAMESCOPE parameter is used to simplify accessing program symbolics by setting the scope so that a symbol can be accessed without using a fully qualified symbol name.

In general, to display or modify the contents of a program symbol using symbolics, it is necessary to access the symbol with a fully or partially qualified symbol name. The format of a fully qualified symbol is:

```
:main_module.procedure1.subprocedure.variable
```

If the variable is global, or within the current symbolic scoping as determined by NAMESCOPE, only the variable name is required to access the variable contents.

When the emulator parameter UPDATE is set to TRUE, NAMESCOPE is automatically updated to match the execution pointer ($) when the value of $ changes. Changing NAMESCOPE does not change the value of $. The value of NAMESCOPE is also changed when the emulator parameter NAMEFRAME is modified.

The DIR SYMBOL command display defaults to the program symbols in the symbolic scope as defined by NAMESCOPE unless overridden by DIR command options.

NAMESCOPE (continued)

When the emulator is searching for a program symbol, the emulator parameter NAMEPATH is consulted to determine the order of the search. To shorten search time, precede the symbol with a NAMEPATH prefix such as SYMBOL// or PUBLIC//.

Examples

1. Assume that a user program with two modules, mod1 and mod2 has been LOADed. View and access the symbols in mod1 without using full qualification then switch to mod2.

```
hlt> NAMESCOPE = :mod1
hlt> DIR MODSYMBOL
TOOL//
   MEMORY          array [0..] of BYTE
   VAL1            BYTE
   VAL2            BYTE
    .
    .
    .
hlt> val2 = 55H
hlt> NAMESCOPE = :mod2
hlt> DIR MODSYMBOL
TOOL//
   MEMORY          array [0..] of BYTE
   VAL3            BYTE
   VAL4            BYTE
hlt> val3
088H
hlt>
```

2. Display the current address and symbolic value of NAMESCOPE.

```
hlt> NAMESCOPE
0500:00C4H
:MOD2.PROC1#3"
hlt>
```

Cross References

$ (execution pointer)
<addr>
DIR
<expr>
NAMEFRAME
Object Hierarchy
Parameters
Symbolic References
UPDATE

NOLIST

Closes a list file

Syntax

```
NOLIST
```

Discussion

This command closes a list (log) file. It has no effect unless the command LIST *filename* has been previously entered or the LOGFILE = *pathname* option was used in the operating environment configuration file (see Chapter 3, Section 3.3.1).

The EXIT command also closes any LIST file that is open when the EXIT command is executed.

Cross-References

EXIT
LIST

Object Hierarchy

Illustrates software organization

Discussion

The object hierarchy refers to the emulator software directory structure. This structure includes several directories (or paths), to commands, keywords, and built-in functions as well as debug objects created by the user during a debugging session. User-defined debug objects included in the debug environment are debug variables, LITERALLYs, PROCs, and program variables defined in a program that has been loaded into memory. Objects contained in user programs include module names, line numbers, user program symbol names and other symbolics.

All objects are maintained in one of the four main directories and their associated sub-directories, which are automatically created by the emulator software. The four main directories are: TOOL//, LOCAL//, GLOBAL//, and SYMBOLGROUP//. You cannot access these directories directly (For example, you cannot enter the command DIR LOCAL//.)

The following chart identifies all the main directory names and a sample of the sub-directories under each main directory. Note this is a nested structure. A complete identifier may consist of the concatenation of one name from each level, but only with all names chosen from the same main directory.

Object Hierarchy (continued)

Directory Name	Examples of Contents
LOCAL//	
KEYWORDGROUP//	
BUILTIN_FUNC//	PRINTF(), SLEEP()
KEYWORD//	ACTIVATE, DEFINE
PARAMETER//	NAMEPATH, BASE
DEBUG//	
LITERALLY//	User-defined
DEBUGVAR//	User-defined
PROC//	User-defined
TOOL//	
KEYWORD//	LOAD, MAP, GO
PARAMETER//	NAMESCOPE
DEBUG//	
LITERALLY//	User-defined
DEBUGVAR//	User-defined
PROC//	User-defined
TOOLVAR//	AX, FLAGS, IP
TOOLPTR//	OFFSET, PHY
SYMBOL//	User-defined
MODSYMBOL//	User-defined
SYMBOLGROUP//	
SYMBOLFILE//	User-defined
PUBLIC//	User-defined
MODULE//	User-defined
LINE//	User-defined
GLOBAL//	
DEBUG//	
LITERALLY//	User-defined
DEBUGVAR//	User-defined
PROC//	User-defined

LOCAL//	contains most objects which support the debug environment, such as keywords, parameters, and debug objects.
KEYWORDGROUP//	contains the group of keywords for the emulator software.
KEYWORD//	contains all of the emulator tool keywords.
BUILTIN_FUNC//	contains string, I/O, and other miscellaneous functions.
PARAMETER//	contains all emulator-defined parameters that control the state of the software. Emulator-defined parameters have reserved word names and a value that can be changed within a defined range (see Parameter in this chapter).
DEBUG//	contains objects defined by a user to enhance the debug process. Types of debug objects are aliases (LITERALLY), variables (<data type>), and procedures (PROC).
LITERALLY//	contains user-defined LITERALLY definitions.
DEBUGVAR//	contains user-defined debug variables.
PROC//	contains user-defined procedures (PROCs) or functions.
TOOL//	contains keywords, command language parameters, debug objects, control variables, tool pointers, namescope symbols, and symbolgroup link lists for the emulator. This directory is created when the emulator is DEFINEd as a TOOL.

Object Hierarchy (continued)

TOOLVAR//	contains a set of control variables that control the state and data related to the hardware resources as viewed by the emulator software (for example, AX, FLAGS, IP, PE).
TOOLPTR//	contains the types of pointers used by the emulator software.
SYMBOL//	contains procedures, functions, labels, variables, and line numbers in a user program's symbol file.
MODSYMBOL//	contains symbols residing within a module.
SYMBOLFILE//	contains program symbol subdirectories.
PUBLIC//	contains public program symbols.
MODULE//	contains program symbols and subdirectories that contain symbols.
LINE//	contains a line numbers of a high-level language program.
SYMBOLGROUP//	contains user program symbolics. This directory is created when program symbolics are loaded.
GLOBAL//	contains globally defined debug objects.

Examples

1. Display the BUILTIN_FUNCs' keywords.

```
hlt> DIR KEYWORD LOCAL//KEYWORDGROUP//BUILTIN_FUNC
```

2. List the parameters associated with the emulator software.

```
hlt> DIR PARAMETER
```

3. List the line numbers in all modules in the loaded user program.

```
hlt> DIR LINE SYMBOLGROUP//MODULE//
```

Cross-References

DIR
Parameters

OSYNC

Sets the synchronous line output

Syntax

```
OSYNC  [= expr]
```

Where:

OSYNC	if entered without an optional expression, displays the current setting of the synchronous line output.
expr	is a 0 (low) or 1 (high).

Default

1 (high)

Discussion

The OSYNC tool variable allows you to set the level of the synchronous output line to either a TTL high (OSYNC = 1) or a TTL low (OSYNC = 0) voltage. The OSYNC level changes with the next invocation of the GO command.

OSYNCH corresponds to the TTL high setting (OSYNC = 1); OSYNCL corresponds to the TTL low setting (OSYNC = 0).

Once set, the synchronous output line remains at the specified TTL voltage until changed by another invocation of GO. The default setting is TTL high (OSYNC = 1).

You can display the current level of OSYNC at any time.

Example

This example sets OSYNC to TTL high, then checks the setting. OSYNCH will be output with the next invocation of GO.

```
hlt> OSYNC=1
hlt> OSYNC
Next GO command will use:  OSYNCH
Current OSYNC level at BNC:  0H
hlt>
```

Cross-References

<expr>
GO
TOOLVAR

Parameters

Define and control the state of the emulator software

Discussion

There are two types of parameters: local and tool. These parameters define and control the state of the emulator software. These parameters are maintained in the directory PARAMETER//.

Local Parameters

BASE	specifies the number base for console input and output. The default is 16T (hexadecimal).
DISPLAYFLAG	controls whether or not the value resulting after an assignment operation is displayed. If DISPLAYFLAG is TRUE (non-zero), the value is displayed. The default is FALSE (zero).
EDITOR	specifies which editor is invoked when the keyword EDIT is entered. The default is "EDLIN".
ERROR	specifies the display format for error messages. ERROR can be set to 0 or 1. In both cases, an error number associated with the error condition is displayed. Additionally, ERROR=1 causes the standard length error message to be displayed, while ERROR=0 causes the abbreviated error message to be displayed. The default is 1.
NAMEPATH	specifies the search path used by the emulator command language when looking for an object. The default is as follows (left to right): KEYWORD// PARAMETER// DYNAMIC// DEBUG// TOOLVAR// TOOLPTR// SYMBOL// PUBLIC//
SYMBOLIC	specifies whether addresses in certain commands should be converted to their equivalent symbolic names.

In addition to these local parameters, the DIR PARAMETER command displays other parameters (for example, CASE, CURRENTTOOL, LASTTOOL) that are recognized internally and should not be modified.

Tool Parameters

$	contains the current execution pointer as a sequential address and its symbolic equivalent. The current execution address may be displayed or modified in a single command using $. It can be assigned an address or any symbolic name which evaluates to a pointer.
NAMESCOPE	contains the address and symbolic equivalent of the current symbolic scope. It can be assigned an address or any symbolic name which evaluates to a pointer.
NAMEFRAME	is used to access different procedure activation frames on the stack. The default is 0.
UPDATE	indicates whether the value of $ should be used to update NAMESCOPE. The default is TRUE.

Partition

An address or a
range of addresses

Syntax

```
{ addr                  }
{ addr TO addr          }
{ addr LENGTH expr      }
```

Where:

addr	specifies a memory location address.
TO *addr*	specifies the ending address used in conjunction with the command entered.
LENGTH *expr*	specifies a number (or an expression that evaluates to a positive integer) that is used to indicate the number of items desired.

Discussion

A partition is a single address or a range of addresses. When a range of addresses is required, specify the range with either the TO or the LENGTH keyword.

The TO keyword assumes byte addresses. Note that in the form *addr* TO *addr*, the two addresses must both be either segmented addresses (i.e., *aa:bb*) or physical address (i.e., *nnn*P). If they are both physical addresses, the first address must be lower (numerically smaller) than the second. If they are both segmented addresses, the segment values for both addresses must be equal.

The result of using the syntax *addr* LENGTH *expr* varies according to the application. For example, with memory access commands, it refers to the number of data type objects that will be affected by the command. With ASM commands, it determines the number of instructions that will be displayed or assembled.

Expressions and symbolic references are valid in partitions.

Examples

1. This example illustrates a physical address range.

```
hlt> ASM OP TO 25P
/* ... display ... */
hlt>
```

2. These examples illustrate virtual address ranges.

```
hlt> ASM 0 TO 4
/* ... display ... */
hlt>
hlt> ASM 4:0 TO 4:38H
/* ... display ... */
hlt>
hlt> BYTE 4:0 LENGTH 5
/* ... display ... */
hlt>
```

3. This example uses a physical address with the keyword LENGTH.

```
hlt> MAP OP LENGTH 128K ICE
hlt>
```

4. These examples illustrate ranges defined with symbolic references.

```
hlt> ASM :main_#3 TO :main_#8 +3
/* ... display ... */
hlt>
hlt> BYTE &buff LENGTH 20T
/* ... display ... */
hlt>
```

5. This example uses the data type keyword BYTE to access an array.

```
hlt> BYTE &sarray[1] TO &sarray[10] = var3
/* ... display ... */
hlt>
```

Partition (continued)
Cross-References

<addr>
ASM
<expr>
MAP
Memory Access
Print
SAVE

PCB

Displays the register contents of the Peripheral Control Block

Syntax

PCB

Discussion

Use the PCB command to display the contents of the registers in the Peripheral Control Block of the 80C186 microprocessor.

You can enter the PCB command at any time when the emulator is halted. If the emulator is running, you can enter the command only if FASTBREAK is set to TRUE (non-zero).

Unlike the 80C186 microprocessor, the emulator microprocessor predefines the LMCS, PACS, MMCS, and MPCS chip-select registers upon initialization. It predefines these registers in order to avoid user accesses to inactive registers, which would result in undefined or illegal register values. The emulator microprocessor predefines the registers as follows:

- LMCS (keyword CSCTRL2) -- 0038H; memory block size 1K; 0 wait states; external READY used.
- PACS (keyword CSCTRL3) -- 0CF8H; memory base address of 0CC00H; 0 wait states; external READY used.
- MMCS (keyword CSCTRL4) -- 41F8H; base address 40000H; 0 wait states; external READY used.
- MPCS (keyword CSCTRL5) -- 81B8H; total memory block size 8K; individual select size 2K; peripherals mapped into I/O space; 7 /PCS lines; A1 and A2 not provided.

You can change the predefined values of the chip-select registers by using the emulator register access commands (see the Register Access entry in this chapter). The new values remain in effect until you change them or re-initialize the emulator through the ACTIVATE or RESET ICE command. Re-initialization of the emulator resets the registers to the predefined values. The registers are also reset to the predefined values whenever power is cycled to the prototype.

PCB (continued)
Example

This example displays the registers in the Peripheral Control Block.

```
hlt> PCB
Relocation  | RELREG=ffffH
 Register   |  Bit 15: 1 (ESC Trap)
            |  Bit 14: 1 (Slave Mode)
            |  Bit 12: 1 (Located in Memory Space)
-------------------------------------------------------------------------------
 Refresh    | Enable Refresh Ctrl  EDRAM=ffffH
 Control    | Clock Pre-scaler     CDRAM=ffffH
            | Memory Partition     MDRAM=ffffH
Enhanced    | Power-Save Control   PDCON=ffffH
-------------------------------------------------------------------------------
Chip Select | CSCTRL3  (PACS)=0cf8H  CSCTRL5  (MPCS)=81b8H
 Registers  | CSCTRL2  (LMCS)=0038H  CSCTRL4  (MMCS)=41f8H  CSCTRL1  (UMCS)=fffbH
-------------------------------------------------------------------------------
   DMA      | Control Word       DMA06=f730H      DMA16=b730H
 Channel    | Transfer Count     DMA05=0000H      DMA15=0000H
Descriptors | Dest. Ptr  (High)  DMA04=fff0H      DMA14=fff0H
            | Dest. Ptr   (Low)  DMA03=05e6H      DMA13=1fffH
            | Src Ptr    (High)  DMA02=fff1H      DMA12=fff0H
            | Src Ptr     (Low)  DMA01=0000H      DMA11=337dH
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
  Timer     | Mode/Ctl Word      TIMER04=2020H  TIMER14=2020H  TIMER24=ffffH
 Control    | Maximum Count B    TIMER03=653dH  TIMER13=001fH  (unused)
Registers   | Maximum Count A    TIMER02=0733H  TIMER12=0733H  TIMER22=0019H
            | Count Register     TIMER01=0000H  TIMER11=0000H  TIMER21=0001H
-------------------------------------------------------------------------------
Current Mode:  Slave (iRMX).
-------------------------------------------------------------------------------
Interrupt   | Unused         INTRPT10=000fH | Intrpt Request  INTRPT8= 0000H
Controller  | Unused         INTRPTF= 000fH | In-Service      INTRPT7= 0000H
Registers   | Level 5 Ctrl   INTRPTE= 000fH | Priority Mask   INTRPT6= 0007H
            | Level 4 Ctrl   INTRPTD= 000fH | Mask            INTRPT5= 00fdH
            | Level 3 Ctrl   INTRPTC= 000fH | Unused          INTRPT4= 0018H
            | Level 2 Ctrl   INTRPTB= 000fH | Unused          INTRPT3= 0018H
            | Level 0 Ctrl   INTRPTA= 000fH | Specific EOI    INTRPT2= 00ffH
            | Intrpt Status  INTRPT9= 0000H | Intrpt Vector   INTRPT1= 0000H
hlt>
```

Cross-References

FASTBREAK
GO
Register Access

PHYSICAL

Converts a segmented address
to a physical address

Syntax

PHYSICAL(*address*)

Where:

(*address*)	is an address in the format *segment:offset*, *offset* or a symbolic address. The parentheses are required punctuation.

Discussion

Use the PHYSICAL command to convert a segmented address to a physical address. The address is converted according to the current conversion rules as discussed in the Address Translation entry.

If the specified address provides only an offset value, the default *segment* is the current code segment (CS). If you enter a physical address, the input address is simply returned with no error.

The setting of the FASTBREAK tool variable affects the PHYSICAL command. If the PHYSICAL command requires access to the processor during emulation, FASTBREAK must be set to TRUE (non-zero).

Example

This example displays the current value of CS, then converts a segmented address to a physical address.

```
hlt> CS
0c000H
hlt> PHYSICAL(0C000:1234H)
0c1234HP
```

Cross-References

Address Translation
FASTBREAK

POINTER <pointer-type>

Determines which pointer data-type is used to access memory

Syntax

```
POINTER  pointer-type  partition  [= expr [,...]]
```

Where:

POINTER — specifies that the *pointer-type* that follows defines how memory is displayed or modified.

partition — is a range of addresses. A partition can be any one of the following.

```
{ addr                  }
{ addr TO addr          }
{ addr LENGTH expr      }
```

expr — specifies a number or an expression. More than one expression can be entered by using a comma to separate them.

pointer-type — is the name of an emulator pointer data type. Supported pointer-types are:

OFFSET	accesses memory as a 16-bit offset address (e.g., 1234)
PHY	accesses memory as a 20-bit physical address, e.g., 12345P.
LPOINTER	accesses memory as a 32-bit full pointer (e.g., 1234:5678) (16-bit segment:16-bit offset).

Discussion

The POINTER *pointer-type* pair is an emulator data-type that specifies a particular format for accessing memory. The *pointer-types* can only be used for memory access; they cannot be used for accessing DEBUGVARs of type POINTER, (see the DEFINE entry).

NOTE

The POINTER *pointer-type* pair cannot be used in expression operations. The following example generates a syntax error:

```
hlt> DEFINE DWORD addr = POINTER PHY 0HP          /*erroneous command */
```

Examples

1. Display 4 bytes starting at memory location 0P (physical) as an OFFSET pointer-type.

```
hlt> POINTER OFFSET 0P LENGTH 2
00000HP    ffffH    ffffH
hlt>
```

2. Display 4 bytes starting at memory location 0P (physical) as an LPOINTER pointer-type.

```
hlt> POINTER LPOINTER 0P
00000HP    ffff:ffffH
hlt>
```

3. Write to memory at location 0P (physical).

```
hlt> POINTER PHY 0P LENGTH 3 = 1234P
hlt> POINTER PHY 0P LENGTH 3
00000HP  01234HP  01234HP  01234HP
hlt>
```

Cross-References

<data type>
DEFINE
<expr>
Memory Access
Partition
REDEFINE

PORT

Displays or modifies the contents of byte-wide, USER-mapped I/O ports

Syntax

```
PORT  io-address  [= data]
```

Where:

PORT *io-address*	displays the contents of the I/O port at the specified io-address. The io-address is a 16-bit expression or number in the current base specifying the physical I/O address. The range of available io-addresses is 0 to 0FFFFH.
data	is any byte of data entered in the current base. Using this parameter writes the data to the specified I/O port.

NOTE

If a port does not have a latched mechanism, a value written to the port may be different from the value read from the port.

Discussion

Use the PORT command to display a value from a byte-wide port or write a value to a byte-wide port. You can access up to 64K byte-wide ports.

The I/O ports must be mapped to USER; the emulator displays an error message if the io-address you attempt to access is mapped to ICE.

Note that Intel Corporation reserves port locations 0F8H-0FFH for coprocessor communication.

The FASTBREAK tool variable controls port access during emulation. Setting FASTBREAK to TRUE (non-zero) allows you to make asynchronous port accesses while the emulator is executing your program. Setting FASTBREAK to FALSE (zero) prevents port accesses during emulation.

The PORT command is disabled if there are any outstanding mapped-I/O port accesses.

Example

The following example displays and changes the contents of the I/O port at address 10H.

```
hlt> PORT 10H
0010H    ffH
hlt> PORT 10H = 20H
```

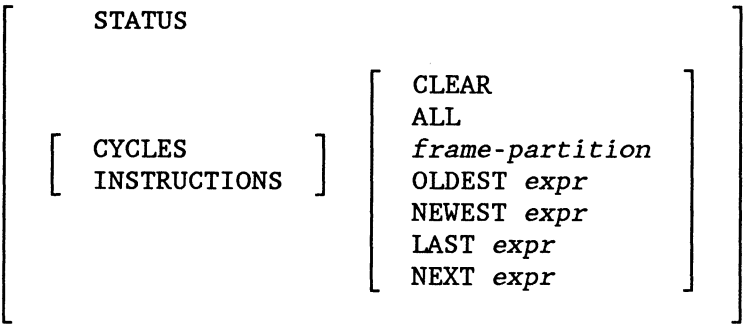
Cross-References

FASTBREAK
IORDY
WPORT

PRINT

Formats and displays the contents of the trace buffer

Syntax



Where:

PRINT	is the command keyword. If entered without parameters, it displays the next item in the trace buffer using the current format (CYCLES or INSTRUCTIONS). An item is either a bus cycle or a disassembled instruction.
STATUS	displays the number of frames in the trace buffer, the current display format (CYCLES or INSTRUCTIONS), and the next frame to be printed. Frames are based on bus cycles, not instructions. Therefore, the number of frames always refers to the number of CYCLES-format frames. The trace buffer contains a variable number of frames. The maximum possible number of CYCLES-format frames is 3072.
CYCLES	displays the trace buffer in bus cycle format. Once set, the CYCLES display format remains in effect until PRINT INSTRUCTIONS is entered.
INSTRUCTIONS	displays the trace buffer as execution instructions. INSTRUCTIONS is the default display format. It remains in effect until PRINT CYCLES is entered.

CLEAR	erases the trace buffer. Issuing a GO command with new break or trace information also clears the trace buffer.
ALL	displays the entire contents of the trace buffer in the current format (CYCLES or INSTRUCTIONS).
frame-partition	specifies a single frame or a range of items to be printed. The range is specified as either *start-frame* TO *end-frame* or *start-frame* LENGTH *number-of-items*.
OLDEST *expr*	displays the specified number of trace buffer items, starting with the oldest frame, 1. Because frames are based on bus cycles, the number of trace items and the number of frames displayed are not necessarily the same; e.g., "PRINT CYCLES OLDEST 5" displays the oldest five bus cycles; thus, the oldest five frames (frames 1 - 5). "PRINT INSTRUCTIONS OLDEST 5" displays the oldest five instructions, and this may require more than five frames.
NEWEST *expr*	displays the specified number of items, relative to the end of the trace buffer. For example, given a trace buffer with 256 entries, the NEWEST 5 frames are frames 252 through 256.
LAST | NEXT *expr*	displays the last or next *expr* item, relative to the current trace buffer pointer. The trace buffer pointer always points to the next frame to be displayed. For example, after displaying the first 100 frames of the trace buffer (frames 1 - 100), the LAST 5 frames are frames 96 through 100, and the NEXT 5 frames are frames 101 through 105.

Trace information can be copied to a file by using the redirection command discussed in section 3.2.9. For example, the command "PRINT INSTRUCTIONS ALL '> TRACE.DOC" redirects the output to a file called TRACE.DOC. This allows you to edit the trace information. In addition, if the editor has the search (or find) feature, you can use that feature to easily locate importorant portions of code.

Discussion

The emulator stores bus and execution data and optional register and memory data in the trace buffer during emulation. The GO command (discussed in this chapter) controls trace collection. The PRINT command displays this trace information. You can view the contents of the trace buffer only when the Activity Monitor is stopped.

During trace collection, the trace buffer fills with frame numbers, addresses, processor status, and time stamps. The bus and execution information is reconstructed from the actual trace data in the hardware.

The trace buffer hardware is 4096 acquisitions deep. More than one acquisition is required to form a CYCLES-format frame. A frame displays either an execution or a bus event. The number of acquired frames is a function of the following:

A. Current microprocessor activity characteristics (i.e., the number of wait states per bus cycle, whether the task is compute-bound or bus-bound, etc.). Less frames are available if wait states are used.

B. The Activity Monitor programming in the GO command. Turning the trace collection on and off frequently reduces the number of available frames.

Trace information is displayed in one of two formats: INSTRUCTIONS or CYCLES.

The INSTRUCTIONS Format

The INSTRUCTIONS format displays execution address and machine-language instructions. Bus cycle data (e.g., memory and I/O reads and writes) is interleaved with the instructions. Memory and register data that resulted from specifying certain *actions* (MEM, REGS, and PCB) with the GO command is also interleaved with the instructions.

The column headings for the INSTRUCTIONS format are the following: FRAME, ADDRESS, DATA, and MNEMONICS.

The FRAME column contains the frame number of the acquisition. Usually, a one-to-one correspondence of frame numbers to acquisitions exists. However, since instruction rebuilding may consume several frames, frame numbers may not be sequential.

The ADDRESS column contains either the start address of an executed instruction or the bus address at which a bus cycle (other than a FETCH) occurred. When the ADDRESS column contains a bus cycle address, the column also contains the status code(s) and data in the following format: address-code-data.

The code portion of the bus cycle data consists of one or more letters indicating the cycle type, as shown below:

Cycle Type	Code
Interrupt Acknowledge	A
I/O Port Input	I
I/O Port Output	O
Halt	H
Fetch Instruction	F
Read Memory	R
Write Memory	W
Locked Instruction	L
DMA0	0
DMA1	1

The DATA column contains data bytes associated with the executed instruction. The MNEMONICS column contains the instruction mnemonics and the operands.

Occasionally, a question mark (?) may appear in the MNEMONICS column. This question mark indicates that no equivalent mnemonic for the contents of the specified location exists or that all of the opcode data necessary for disassembly was not stored. Memory accesses are not attempted in an effort to rebuild mnemonics.

Symbolic Support in INSTRUCTIONS Format

The INSTRUCTIONS display format may or may not include symbolic references, depending on the operating mode of the emulator and the setting of the SYMBOLIC parameter.

If SYMBOLIC is set to TRUE (non-zero) and the emulator is in Halt mode, the INSTRUCTIONS format includes symbolic references. The emulator converts the physical addresses in the trace buffer into segmented addresses, and then displays a symbolic reference for every address that has an exact symbolic match in the symbol table. You can improve the display response time by setting SYMBOLIC to FALSE (zero).

If the emulator is in Run mode, no symbolic information appears in the INSTRUCTIONS format display, regardless of the setting of the SYMBOLIC variable. While running, the emulator cannot convert the physical addresses in the trace buffer into segmented addresses.

The CYCLES Format

The CYCLES format displays trace information in bus-cycle or execution-address format and in the order that it was collected: frame number, execution and bus address, bus data, processor status, user state, and time-stamp (clock) information. The CYCLES format also displays register and memory accesses that were stored using the GO command.

The column headings for the CYCLES format are the following: FRAME, EXEC ADDR, BUS ADDR, DATA, STATUS, USTATE, and CLOCKS.

The FRAME column contains the frame number of the acquisition.

The EXEC ADDR column contains the address of the first byte of the instruction that was executed. (Technically, an instruction may consist of one or more bytes, all of which were executed.) Note that data, status, and time-stamp information is not available with execution addresses.

The BUS ADDR column contains the address associated with any kind of bus activity.

The DATA column contains up to 16 bits of data, shown in low-byte/high-byte order. If only the high byte is valid, the data is right-justified. If only the low byte is valid, the data is left-justified. With 80186 applications, the DATA column displays two bytes of data

per frame; with 80188 applications, the DATA column displays one byte per frame. Thus, for 80188 applications, the length of cycles trace will double.

The STATUS column contains a variable-length hexadecimal status code followed by a one-character access code. This access code indicates the origin of the trace data. Refer to Table 6-10 for definitions of these access codes.

Table 6-10 CYCLES Format Access Codes

Bits	Function	Code
xxxx 0000	Interrupt Acknowledge	A
xxxx 0001	I/O Port Input	I
xxxx 0010	I/O Port Output	O
xxxx 0011	Halt	H
xxxx 0100	Fetch Instruction	F
xxxx 0101	Read Memory	R
xxxx 0110	Write Memory	W
xxx0 xxxx	Locked Instruction	L
xx1x xxxx	DMA0	0
x1xx xxxx	DMA1	1

The USTATE column contains either 0 or 1. This number indicates the *user-state* that the GO command was in during the execution of this part of the program. According to the GO syntax, you can specify up to two *user-states*.

The following example illustrates *user-states* in a sample GO command entry:

```
hlt> GO TIL BUS 1618AHP WRITE THEN EXECUTION \
hlt>> 1000DHP RESTART ORIF EXECUTION 1FF4EHP \
hlt>> HALT
```

The statement "BUS 1618AHP WRITE" is *user-state-0* (USTATE 0). The statement "THEN EXECUTION 1000DHP RESTART ORIF EXECUTION 1FF4EHP HALT" is *user-state-1* (USTATE 1).

PRINT (continued)

The CLOCKS column contains time-stamp information. This information is associated with stored bus cycles only. The time stamp counts prototype clocks. The format of this field depends on the value of the CLK command.

If CLK is set to 0, the CLOCKS column contains the number of clock ticks that elapsed between bus cycles. The time stamp can count up to 257 ticks. If the number of ticks exceeded the 257 maximum, the CLOCKS column for that entry will be ">257".

If CLK is non-zero, the column heading will be TIME. (The value of CLK is assumed to be the frequency, in MHz, of the CLKOUT processor clock signal in the prototype.) The TIME column contains the elapsed time between bus cycles in nanoseconds.

Examples

1. This example displays the trace buffer in the INSTRUCTIONS format. The data in the trace buffer was acquired using "GO FROM 1100:4H TRACE TIL EXECUTION 10009HP MEM 1301CHP LENGTH 16T HALT".

```
hlt> PRINT INSTRUCTIONS ALL
FRAME--ADDRESS------DATA-----------MNEMONICS------------------------
   2   11004HP       fa             CLI
   4   11005HP       2e8e160000     MOV   SS,WORD PTR CS:0000H
   8   11000HP-R-0016
   9   1100aHP       bc9001         MOV   SP,0190H
  12   1100dHP       2e8e1e0200     MOV   DS,WORD PTR CS:0002H
  16   11002HP-R-0013
  17   11012HP       eaa0000010     JMP   1000H:00a0H (PSTART_)
:INTERFACE.PSTART_
  22   1000:00a0H    e95dff         JMP   $-00a0H  ;A=010000 (#85)
:MESSG2.MAIN_#85
  26   1000:0000H    56             PUSH  SI
  29   1618eHP-W-bced
  30   1000:0001H    57             PUSH  DI
  32   1618cHP-W-0000
  33   1000:0002H    55             PUSH  BP
  34   1618aHP-W-0000
  35   1000:0003H    8bec           MOV   BP,SP
```

```
:MESSG2.MAIN_#89
  36   1000:0005H   b81a00          MOV   AX,001aH
  38   1000:0008H   50              PUSH  AX
  41   16188HP-W-1a00
  42   1000:0009H   ff361800        PUSH  WORD PTR 0018H
  43   13018HP-R-0000
  45   16186HP-W-0000

       -----------------------------------------
  46   |      Memory trace at CS:IP 1000:000dH       |
       -----------------------------------------
       01301cHP  00 00 00 00 00 00 00 00 00 00 00 00
       013028HP  00 00 00 00
At end of trace buffer.
hlt>
```

2. This example displays the same trace buffer in the CYCLES format.

```
hlt> PRINT CYCLES ALL
FRAME--EXEC ADDR--BUS ADDR---DATA-----STATUS--USTATE----CLOCKS
   1              11004HP    fa2e     14 F       0         4
   2   11004HP
   3              11006HP    8e16     14 F       0         4
   4   11005HP
   5   11006HP
   6              11008HP    0000     14 F       0         4
   7              1100aHP    bc90     14 F       0         4
   8              11000HP    0016     15 R       0         6
   9   1100aHP
  10              1100cHP    012e     14 F       0         4
  11              1100eHP    8e1e     14 F       0         4
  12   1100dHP
  13   1100eHP
  14              11010HP    0200     14 F       0         4
  15              11012HP    eaa0     14 F       0         4
  16              11002HP    0013     15 R       0         6
  17   11012HP
  18              11014HP    0000     14 F       0         4
  19              11016HP    10bf     14 F       0         4
  20              11018HP    4d3f     14 F       0         4
  21              100a0HP    e95d     14 F       0         6
  22   100a0HP
  23              100a2HP    ff3f     14 F       0         4
```

PRINT (continued)

```
FRAME--EXEC ADDR--BUS ADDR---DATA-----STATUS--USTATE----CLOCKS
  24             100a4HP    d95f     14 F       0         4
  25             10000HP    5657     14 F       0         7
  26   10000HP
  27             10002HP    558b     14 F       0         4
  28             10004HP    ecb8     14 F       0         4
  29             1618eHP    bced     16 W       0         4
  30   10001HP
  31             10006HP    1a00     14 F       0         5
  32             1618cHP    0000     16 W       0         4
  33   10002HP
  34             1618aHP    0000     16 W       0         9
  35   10003HP
  36   10005HP
  37             10008HP    50ff     14 F       0         5
  38   10008HP
  39             1000aH     3618     14 F       0         4
  40             1000cHP    00e8     14 F       0         4
  41             16188HP    1a00     16 W       0         4
  42   10009HP
  43             13018HP    0000     15 R       0         7
  44             1000eHP    0f00     14 F       0         4
  45             16186HP    0000     16 W       0         5
       ------------------------------------------
  44   |      Memory trace at CS:IP 1000:000dH       |
       ------------------------------------------
       01301cHP  00 00 00 00 00 00 00 00 00 00 00 00
       013028HP  00 00 00 00
At end of trace buffer.
hlt>
```

Cross-References

CLK
GO
SYMBOLIC

Syntax

```
[RE]DEFINE  PROC  [proc-data-type]  proc-identifier
    [ (arg-name [,...])]  [DEFINE parameter-type arg-name][...]
    {
    [ commands                ]
    [ [forward-spec] [...]    ] [...]
    [ RETURN (arg-name)       ]
    }
```

forward-spec::=

```
  FORWARD { PROC  [proc-data-type]  proc-identifier  }
          { POINTER  [dir-name]  obj-identifier      }
          { data-type  [dir-name]  obj-identifier    }
```

Where:

DEFINE	signals creation of a user-defined procedure or procedure argument.
REDEFINE	signals creation or re-creation of a user-defined procedure. Use REDEFINE to change the definition of an existing PROC.
PROC	specifies that a user-defined procedure (or function) is to follow.
proc-data-type	specifies the data type to be returned (see <data type> in this chapter).
proc-identifier	specifies the name of the PROC.
(*arg-name*)	specifies the name of a debug object that is used as a parameter in the procedure. Names of parameters are separated by commas and the parentheses are required punctuation.
parameter-type	specifies the data type of the parameter (see <data-type> in this chapter).

PROC (continued)

{*commands*}	are any emulator commands (except for INCLUDE), including the ARGVECTOR and ARGCOUNT predefined local variables. The braces are required punctuation.
FORWARD	specifies a forward referenced or recursive debug object. Forward reference any debug objects that have not been defined.
POINTER	specifies a memory POINTER data type (see POINTER in this chapter)
dir-name	specifies an emulator object directory name such as: LOCAL// or GLOBAL// (see Object Hierarchy in this chapter).
obj-identifier	specifies a user-defined name for the debug object or POINTER that is being forward referenced (not defined yet).
data-type	specifies an emulator data type (see <data type> in this chapter).
RETURN	specifies an argument to be returned upon completion of PROC execution.

Discussion

A PROC is created with the DEFINE (or REDEFINE) command. It is best to use the EDIT command to initially create and to edit a PROC. A PROC is executed when it is called by name, just as a built-in function is executed. PROCs can be defined to be stand-alone or to accept parameters. If the argument data type is not specified, the caller's data type is used as the default. When executing a PROC, an error message is displayed if the PROC requires arguments which have not been passed to it.

You can also DEFINE PROCs which accept a variable number of arguments. There are two predefined local variables, ARGVECTOR and ARGCOUNT. The ARGCOUNT variable tracks the number of arguments supplied when the function is called. The ARGVECTOR variable (array) stores the actual arguments passed when a function is called.

Nested PROCs and recursive or reentrant PROCs are supported to the extent of available host memory. A nested PROC cannot be executed outside the PROC where it resides. PROCs can also call other PROCs that have been previously defined. Use the keyword FORWARD to reference debug objects (including other PROCs) that have not yet been defined. To DEFINE recursive PROCs, the keyword FORWARD must be used (see example 7). Redefining or editing a PROC that contains FORWARD references results in the removal of those debug objects that are forward referenced.

Debug variables DEFINEd inside the PROC are local to the PROC unless they are declared as STATIC. Debug variables inside the PROC that are not declared STATIC are automatically removed after the execution of the PROC. Do not use REDEFINE inside a PROC. REDEFINEing a debug object that was DEFINEd outside a PROC removes the debug object outside the PROC.

PROCs can return values using the RETURN command. If the RETURN command is not used or executed, the PROC returns a NULL value. The RETURN keyword is only used inside procedure definitions; it is not a separate command. If the return data type does not match the calling data type, then an explicit data-type conversion occurs. The default data type return value is QWORD (ORD8).

NOTE

When storing PROC definitions on disk using the PUT/APPEND command, LITERALLY definitions used within the PROC are not expanded. When restoring the PROCs at a later time using the INCLUDE command, if the LITERALLY definitions do not exist, an error results.

PROC (continued)
Examples

1. Define a PROC to execute a series of commands to display the FLAGS bits in binary.

```
hlt> DEFINE PROC myproc
hlt>> {
hlt>> BASE = 2T
hlt>> FLAGS
hlt>> BASE = 16T
hlt>> }
hlt>
```

2. The following procedure accepts 3 parameters and returns their average:

```
hlt> DEFINE PROC avg(a,b,c)
hlt>> {
hlt>> RETURN((a + b + c) /3)
hlt>> }
hlt>
```

3. To execute the procedure, enter the following:

```
hlt> BASE = 10T
hlt> avg(4,6,3)
4T
hlt>
```

4. The next example is an averaging PROC using the ARGVECTOR and ARGCOUNT variables. Compare the following example of the avg PROC to the previous example. Also note since this PROC has a RETURN statement, the PROC can be used anywhere an expression is valid, for instance, in a DEFINE statement.

```
hlt> REDEFINE PROC avg()
hlt>> {
hlt>> DEFINE INT4 sum = 0                                    /*  accumulator   */
hlt>> DEFINE INT2 i                                          /*  loop counter  */
hlt>> FOR (i = 0; i < ARGCOUNT; sum += ARGVECTOR[i++]);
hlt>> RETURN((ARGCOUNT) ? (sum/ARGCOUNT) : (0))
hlt>> }
hlt>
hlt> DEFINE INT2 runavg = avg(4,6,3)
hlt> runavg
4T
hlt>
```

5. The following procedure shows the effect of changing the PROC's return data type (*proc-type*).

```
hlt> DEFINE CHAR cvar='a'
hlt> DEFINE PROC INT4 cproc
hlt>> {
hlt>> RETURN cvar
hlt>> }
hlt>
hlt> cproc
00000061H
hlt>
hlt> REDEFINE PROC CHAR cproc
hlt>> {RETURN cvar}
hlt> cproc
'a'
hlt>
```

6. Use the FORWARD keyword to refer to undefined procs.

```
hlt> DEFINE PROC INT8 calc (a,b,c)
hlt>> DEFINE INT8 a
hlt>> DEFINE INT8 b
hlt>> DEFINE INT8 c
hlt>> {
hlt>> FORWARD PROC INT8 min
hlt>> FORWARD PROC INT8 max
hlt>> IF ((a > 0) && (b > 0))
hlt>> RETURN (max(a,b) * c) \
hlt>> ELSE IF ((a < 0) && ( b < 0))
hlt>> RETURN (min(a,b) * c) \
hlt>> ELSE RETURN (0)
hlt>> }
hlt>
hlt> DEFINE PROC INT8 min (x,y)
hlt>> {
hlt>> RETURN ((x < y) ? (x) : (y))
hlt>> }
hlt>
hlt> DEFINE  PROC INT8 max (x,y)
hlt>> {
hlt>> RETURN ((x > y) ? (x) : (y))
hlt>> }
hlt>
```

7. Use the FORWARD definition in a recursive procedure.

```
hlt> DEFINE PROC QWORD factorial (n)
hlt>> DEFINE QWORD n
hlt>> {
hlt>> FORWARD PROC QWORD factorial                    /* recursive proc */
hlt>> IF (n == 0)
hlt>> RETURN (1)
hlt>> ELSE RETURN (n * factorial(n - 1))
hlt>> }
hlt>
hlt> BASE = 10T
hlt> factorial (4)
24T
hlt>
```

Cross-References

DEFINE (for more PROC examples)
DIR
EDIT
<expr>
Functions
REDEFINE
SHOW

Prompts

The command-line prompts that indicate the emulator mode.

Discussion

The emulator has five operating modes: Halt, Run, Step, Pause, and Arrested. The command-line prompt tells you which mode is current.

The five operating modes and their command-line prompts are described in this entry and are summarized in Table 6-11.

Halt Mode

In Halt mode, the emulator is halted. The command-line prompt is either ""hlt>" or "idi>", depending on the setting of ENI and the status of interrupt servicing. The "hlt>" prompt appears when ENI is set to 0. The emulator is halted and will not service any interrupts. The "idi>" (Interrupts During Interrogation) prompt appears when ENI is set to 1 or 2. The emulator is halted, but will service interrupts.

Changing ENI from 1 or 2 to 0 affects the emulator immediately, while changing it from 0 to 1 or 2 affects the emulator after the next halt. These ENI changes can be made only while the emulator is in Halt mode.

Run Mode

In Run mode, the emulator is executing program code. The command-line prompt is either "emu>", "emu+>", "fst>", or "fst+>". The "+" indicates that the Activity Monitor is active.

The setting of FASTBREAK determines which command-line prompt is displayed after you enter the GO command (either "emu>" or "fst>").

The "emu>" or "emu+>" prompt appears when FASTBREAK is set to FALSE (zero). The emulator is executing code, and you cannot make any user-directed memory or register access requests. Once the emulator is halted, however, you can make register and memory accesses.

The "fst>" or "fst+>" prompt appears when FASTBREAK is set to TRUE (non-zero). With this setting, you can make user-directed memory and register access requests during emulation.

Note that the emulator may or may not display the Run mode prompts, depending on the setting of QUERY. (See the QUERY entry in this chapter for more details.)

Step Mode

The emulator enters Step mode when you enter the LSTEP or STEP command. The command-line prompt is "?". You can enter Step mode only when the emulator is in Halt mode.

Once the emulator is in Step mode, you may continue stepping by pressing the <Enter> key. To exit Step mode (and return to Halt mode), press the E key.

The Activity Monitor is not active during Step mode.

Pause Mode

In Pause mode, the emulator pauses because of one of the following:

IO?>	The emulator is paused as the result of a HOLDIO command. A mapped I/O request will remain pending until you issue the RELEASEIO command.
isync?>	The emulator is paused, waiting for ISYNC to go high. (Refer to the discussion in Chapter 5 on the synchronization of multiple emulators for more information.)
reset?>	The emulator is paused because the prototype is being held reset.
busact?>	The emulator is paused because a BUSACT time-out has occurred.
memrdy?>	The emulator is paused because a MEMRDY time-out has occurred.
iordy?>	The emulator is paused because an IORDY time-out has occurred.

Prompts (continued)

A "+" is appended to the Pause mode prompts when the Activity Monitor is active.

For more information on emulator time-outs, refer to the BUSACT, MEMRDY, and IORDY entries in this chapter.

While in Pause mode, the emulator does not respond to interrupts or hold requests. Therefore, use this mode with extreme care. For example, it might be inappropriate to use this mode if your prototype employs a real-time operating system, or if memory is mapped to prototype dynamic RAM.

Arrested Mode

In Arrested mode, the emulator locks up because of one of the following:

`pwr?>` The emulator cannot detect power in the prototype. Turn on power to the prototype, then proceed.

`clk?>` The emulator cannot detect a prototype clock. Fix the prototype clock, then proceed.

`???>` The emulator is locked up due to a serious, unrecoverable problem. DEACTIVATE, then re-ACTIVATE the emulator.

`hwfail?>` The emulator is locked up due to a hardware failure. Contact your Intel service representative for assistance (see the inside-back cover of this manual).

In Arrested or Pause mode, you can enter any commands that do not require access to the 80C186 microprocessor.

Table 6-11 ICE™-186/188 Emulator Modes

Mode	Prompt*	Description
Halt	hlt>	Emulator is halted with interrupts disabled. ENI=0.
Halt	idi>	Emulator is halted with interrupts enabled. ENI=1 or ENI=2.

* A "+" is appended to the prompt if the Activity Monitor is active.

Table 6-11 ICE™-186/188 Emulator Modes (continued)

Mode	Prompt*	Description
Step	?	Stepping commands are active. Step mode must be entered from and will return to Halt mode.
Run	emu>	Emulator is executing the program; no fastbreaks are allowed. FASTBREAK=FALSE.
Run	fst>	Emulator is executing the program, fastbreaks are allowed. FASTBREAK=TRUE.
Pause	IO?>	Emulator is paused because a mapped I/O request is pending.
Pause	isync?>	Emulator is waiting for the synchronous line input to go high.
Pause	reset?>	Prototype is being held reset.
Pause	busact?>	BUSACT time-out has occurred.
Pause	memrdy?>	MEMRDY time-out has occurred.
Pause	iordy?>	IORDY time-out has occurred.
Arrested	pwr?>	Emulator is locked up because it cannot detect prototype power. Turn on power to the prototype, then proceed.

* A "+" is appended to the prompt if the Activity Monitor is active.

Prompts (continued)

Table 6-11 ICE™-186/188 Emulator Modes (continued)

Mode	Prompt*	Description
Arrested	clk?>	Emulator is locked up because it is not receiving a prototype clock. Fix the prototype clock, then proceed.
Arrested	???>	Emulator is locked up due to a serious, unrecoverable problem.
Arrested	hwfail?>	Emulator is locked up due to a hardware failure.

* A "+" is appended to the prompt if the Activity Monitor is active.

Cross-References

BUSACT
ENI
FASTBREAK
GO
HALT
HOLDIO
IORDY
LSTEP
MAPIO
MEMRDY
QUERY
STEP

Syntax

```
PSTEP  [FROM address ]  [step-size]
```

Where:

PSTEP	steps through your program by high-level language statements. The line numbers for the statements are assigned to your program when it is compiled by an Intel compiler.
FROM *address*	specifies a starting address where PSTEP is to begin. The default starting address is the current execution pointer.
step-size	is an unsigned integer expression specifying the number of statements within the step. The default *step-size* is 1 statement; the maximum is 255T statements.

Discussion

Use the PSTEP command to step through your program by high-level language statements.

When PSTEP is executed, it displays the statements in the specified step, then returns the emulator to Halt mode. If a statement is a direct call to a procedure, the PSTEP command counts the procedure call and the statements within the called procedure as a single statement.

You can use the current execution pointer as the starting address for PSTEP, or you can specify a starting address by using the FROM *address* parameter. In either case, the starting address must be the start of a statement.

The *step-size* parameter specifies the number of statements to step through, from 1 to 255T statements.

PSTEP (continued)

While the PSTEP command is active, the step breaks are the only breakpoints acknowledged by the emulator. The Activity Monitor trace is turned off, and the emulator ignores any breakpoints established by the GO command.

You can use the PSTEP command only when the emulator is in Halt mode.

Example

This example executes four instructions using the PSTEP command.

```
hlt> PSTEP FROM :messg.main_#121 4
:MESSG.INITIALIZE_BUFFER_#121     (1000:001fH)
:MESSG.INITIALIZE_BUFFER_#135     (1000:0024H)
:MESSG.MAIN_#103     (1000:0013H)
:MESSG.MAIN_#104     (1000:0016H)
hlt>
```

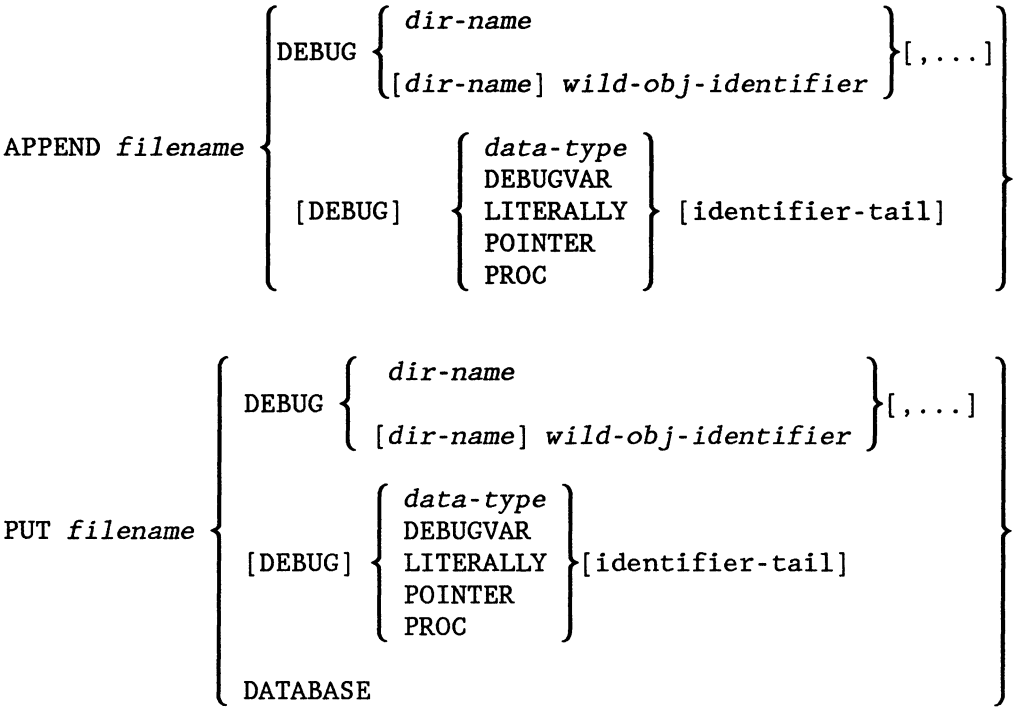
Cross-References

CALLSTACK
ISTEP
LSTEP
STEP

PUT / APPEND

Stores emulator data base information to a file

Syntax



identifier-tail::= [*dir-name*] [*wild-obj-identifier*] [,...]

Where:

APPEND — causes debug information to be stored in the specified file by appending to an existing file. The specified file must have write permission. If the file does not exist, it is created.

PUT	causes debug information to be stored in the specified file. If the file already exists, a query to overwrite the existing file is displayed. If the PUT command is executed from within an INCLUDE file (see the INCLUDE entry), and the file already exists, the command fails.
filename	specifies the path and file name where information is to be stored.
DEBUG	if entered without options, specifies that all debug definitions are stored.
data-type	specifies that all debug variables of that data type are stored. (see the <data type> entry.)
DEBUGVAR	specifies that all debug variable definitions are stored in the file.
LITERALLY	specifies that all LITERALLY definitions are stored in the file.
POINTER	specifies that all POINTER definitions are stored in the file.
PROC	specifies that all PROC definitions are stored in the file.
dir-name	specifies an emulator object directory such as GLOBAL// or LOCAL//.
wild-obj-identifier	specifies the name of a debug object. The * and ? can be used as wild-card characters.
DATABASE	stores a binary representation (interpreted by the emulator software) of all currently defined debug objects and the emulator parameters to the file specified.

Discussion

Use the APPEND or PUT commands to store objects created during a debug session in a disk file. This information can be stored in two forms: a text file or an emulator data base file. Objects stored in the text file format are stored using the REDEFINE command. The text file can then be included (see the INCLUDE entry) during later debug sessions. A data base file can be restored by using the -r invocation option when invoking the emulator software.

When the emulator data base is stored, full path names are stored for the supporting files unless they reside in the current working directory. When a data base is restored, if the full path names are no longer valid, the data base is not properly restored.

Also, when a debug procedure (PROC) is defined, a temporary file is used to store information about the PROC. The directory where this temporary file is stored depends on the designation of the host operating system TMP or TEMP environment parameters. The full path name for the temporary file is recorded in the data base. When a binary representation of the data base is stored with the EXIT or PUT command, this full path name is also stored. If the data base is restored on a system that does not contain this path, any attempt to execute or read the contents of the PROC results in an error.

When storing PROC definitions on disk using the APPEND or PUT command, LITERALLY definitions used within the PROC are not expanded. When restoring the PROCs at a later time using the INCLUDE command, if the LITERALLY definitions do not exist, an error results.

PUT / APPEND (continued)

Examples

1. The following example shows how to check the listing of the current DEBUG definitions and use the PUT command to store all the current LITERALLY definitions to a file.

```
hlt> DIR DEBUG
LOCAL//
    def         literally
    lit         literally
      i           ord1
      j           ord2
      k           int2
hlt> PUT C:myfile LITERALLY
hlt>
```

2. The following example shows how to use the APPEND command to store all the PROC definitions to an existing file.

```
hlt> APPEND C:myfile PROC
hlt>
```

3. The following example shows how to store a binary representation of the data base to a file.

```
hlt> PUT C:\tmp\mybase DATABASE
hlt>
```

Cross-References

DEFINE
INCLUDE
PROC
REDEFINE
SAVE

Syntax

QSTAT [= *bool-expr*]

Where:

QSTAT	if entered without an optional expression, displays the current QSTAT setting.
bool-expr	is any expression that evaluates to TRUE (non-zero) or FALSE (zero).

Defalut

FALSE

Discussion

Use the QSTAT tool variable to specify whether the emulator supports queue status or normal mode. The mode setting determines the signal definitions of processor pins 61, 62, and 63.

If QSTAT is set to FALSE, the processor is programmed for normal mode and the pins are defined as:

Pin 61 = ALE
Pin 62 = /RD
Pin 63 = /WR

If QSTAT is set to TRUE, the processor is programmed for queue status mode and the pins are defined as:

Pin 61 = QS0
Pin 62 = /QSMD
Pin 63 = QS1

NOTE

A slash (/) preceding a signal name indicates that the signal is active low.

You can view the setting of the QSTAT variable at any time, but can change it only when the emulator is halted.

Note that you can also program the QSTAT variable via the communications configuration file (see Chapter 3, Section 3.3.1).

NOTE

The 80C186 does not support the 8087 instruction set. In compatibility mode, the 80C186 will always TRAP (type 7) on the execution of the ESC opcode and return the address of the ESC instruction. This places the 80C186 in an endless loop of exception handling.

Example

This example displays the current setting of QSTAT and then changes the setting.

```
hlt> QSTAT
TRUE
hlt> QSTAT = 0
hlt>
```

Cross-References

Chapter 3, Section 3.3.1
<expr>
TOOLVAR

QUERY

Determines whether commands can be entered while the emulator is in Run mode

Syntax

QUERY [= *bool-expr*]

Where:

QUERY	if entered without an optional expression, displays the current QUERY setting.
bool-expr	is any expression that evaluates to TRUE (non-zero) or FALSE (zero).

Default

TRUE

Discussion

The QUERY tool variable determines whether you can enter commands while the emulator is running.

If QUERY is set to FALSE and you enter the GO command, the command-line prompt disappears from the display and a "Waiting..." message appears periodically on the screen. You cannot enter any commands at this time. The prompt reappears if the Activity Monitor stops or the emulator halts. You can exit this mode by pressing <Ctrl><Break>.

If QUERY is set to TRUE and you enter the GO command, a Run mode prompt appears on the display. This means that you can enter any commands that do not violate the FASTBREAK variable setting (see FASTBREAK in this chapter).

You can enter the STATUS (S) command to obtain emulator status.

For a complete definition of command-line prompts, refer to the Prompts entry in this chapter.

Examples

1. In this example, QUERY is set to zero (FALSE) and the GO command is executed. The QUERY mode is then exited by a <Ctrl><Break>.

```
hlt> QUERY = 0
hlt> GO
     Waiting...
     Waiting...
     Waiting...          ( <Ctrl><Break> is entered )
^C

*** break ***
emu+>
```

2. In this example, QUERY is set to non-zero (TRUE) and the GO command is executed. The STATUS command is then used to obtain the emulation status.

```
hlt> QUERY = 1
hlt> GO
  Use S (status) command to determine emulation status.
emu+> S
  Emulator Status:  running user code.
  Activity Monitor Status:  Running.
emu+>
```

Cross-References

GO
Prompts
STATUS / S

R187

Displays the 80C187 numeric processor extension unit register contents

Syntax

R187

Discussion

The R187 command displays the 80C187 numeric processor extension unit register contents in the current base.

The setting of the COPROC187 command affects the operation of R187 If COPROC187 is set to FALSE, 80C187 register accesses are prevented at all times.

The R187 command can only be used when the emulator is halted or servicing interrupts.

Display of the 80C187 stack registers (ST0 - ST7) is supported only if you have an 80287 coprocessor installed on your IBM PC AT system board.

Examples

1. The following example illustrates a display of the 80C187.

```
hlt> R187
FCW = 037fH
  Exceptions Mask:   PM  UM  OM  ZM  DM  IM
                      1   1   1   1   1   1
  Precision Control: 64-bit Significand
  Rounding Control: Round to nearest or even
FSW = 7901H  Top of stack: Reg7
FTW = 38f0H
FIA =0004001fH
FIO =    04c3H
FDA =000cd340H
ST0:   VALID    00000000000000c0ff3f   1.50000000000000000e+0
ST1:   EMPTY    00000000000000a00040
ST2:   ZERO     0000000000000000000    0
ST3:   VALID    00000000000000a00040   2.50000000000000000e+0
ST4:   VALID    00989999999999d90040   3.40000000000000000e+0
ST5:   EMPTY    00989999999999d90040
ST6:   SPECIAL  00989999999999d90040   Infinity
ST7:   VALID    00989999999999d90040   3.40000000000000000e+0
```

Cross-References

COPROC187
Register Access 80C187

Syntax

```
                ┌ LITERALLY [dir-name] obj-id = "string"                          ┐
                │                                                                 │
                │ PROC (see PROC entry in this chapter)                           │
REDEFINE        │                                                                 │
                │                  ┌ data-type ┐                                  │
                │ [STATIC]         │           │ [dir-name] obj-id [= expr]       │
                │                  └ POINTER   ┘                                  │
                └                                                                 ┘
```

Where:

REDEFINE	signals the creation of a debug object. No error message results whether the debug object has or has not previously been defined.
PROC	specifies a user-defined function or procedure. (See PROC in this chapter.)
LITERALLY	is used to create an alias.
dir-name	specifies an emulator directory name, for example, LOCAL// or GLOBAL//. See Object Hierarchy and NAMEPATH in this chapter.
obj-id	specifies a unique, user-defined name for the object being defined.
"string"	(with LITERALLY) is the string of characters replaced by *obj-identifier*. The quotation marks are required punctuation to delimit *string*.
STATIC	indicates that a debug variable will be globally recognized. Data types are static unless they are defined inside a block (e.g., DO...END).
data-type	specifies an emulator data type (see <data type> in this chapter).
POINTER	indicates that a debug pointer will be defined (see <data type> in this chapter).
expr	specifies a number, address, or expression.

Discussion

Use the REDEFINE command to create or recreate debug objects. A debug object can be REDEFINED without first being REMOVEd. Debug objects are LITERALLY definitions, debug variables, and PROCs used to customize the debug environment. Reference a debug object by entering its name.

NOTE

Do not use REDEFINE inside a PROC. REDEFINEing a debug object that was previously DEFINEd outside the PROC removes the debug object outside the PROC (see example 4).

Examples

1. After DEFINEing a BYTE debug variable, change its type to INT2 and assign it a new value.

```
hlt> DEFINE BYTE var1 = 0
hlt> REDEFINE INT2 var1 = 400
```

2. Alias the number 1000 as the debug LITERALLY named max.

```
hlt> REDEFINE LITERALLY max = "1000"
hlt>
```

3. The following example REDEFINEs a procedure.

```
hlt> REDEFINE PROC power(arg1,arg2)
hlt>> DEFINE INT1 arg1
hlt>> DEFINE INT1 arg2
hlt>> {
hlt>> DEFINE INT1 index
hlt>> DEFINE DWORD result = 1
hlt>> FOR (index = 1; index <= arg2; ++index)
hlt>> result=result * arg1
hlt>> RETURN(result)
hlt>> }
hlt>
```

4. Avoid using REDEFINE inside a PROC unless you want to remove a debug object.

```
hlt> DEFINE INT4 var2
hlt> DIR DEBUG
LOCAL//
    var2 int4
hlt> DEFINE PROC p1
hlt>> {
hlt> REDEFINE INT4 var2
hlt>> }
hlt> DIR DEBUG
LOCAL//
   p1  proc                                  /* var2 has been removed */
hlt>
```

Cross-References

<data type>	INCLUDE	PUT/APPEND
DEFINE	NAMEPATH	PROC
DIR	Object	REMOVE
EDIT	Hierarchy	SHOW
<expr>	POINTER	

Register Access

Displays or modifies the contents of microprocessor registers

Syntax

register [=*expr*]

Where:

register	displays the current value of the microprocessor registers or flag and is one of the keywords listed in Table 6-12 or 6-13.
expr	is a value (of the correct data type) used to set the contents of an microprocessor register.

Table 6-12 Microprocessor Registers

Keyword	Description	Memory Type
AX	Accumulator register word	ORD2
AL	Accumulator word low byte	ORD1
AH	Accumulator word high byte	ORD1
BX	B register word	ORD2
BL	B word low byte	ORD1
BH	B word high byte	ORD1
CX	C register word	ORD2
CL	C word low byte	ORD1
CH	C word high byte	ORD1
DX	D register word	ORD2
DL	D word low byte	ORD1
DH	D word high byte	ORD1

Table 6-12 Microprocessor Registers (continued)

Keyword	Description	Memory Type	
SI	Source Index	ORD2	
DI	Destination Index	ORD2	
BP	Base Pointer	ORD2	
SP	Stack Pointer	ORD2	
CS	Code Segment	ORD2	
DS	Data Segment	ORD2	
ES	Extra Segment	ORD2	
SS	Stack Segment	ORD2	
IP	Instruction Pointer	ORD2	
FLAGS	All Flags	ORD2	
FH	High byte of flag word	ORD1	
FL	Low byte of flag word	ORD1	
OFL	Overflow flag	BIT1	(bit11)
DFL	Direction flag	BIT1	(bit10)
IFL	Interrupt flag	BIT1	(bit9)
TFL	Trap flag	BIT1	(bit8)
SFL	Sign flag	BIT1	(bit7)
ZFL	Zero flag	BIT1	(bit6)
AFL	Auxiliary flag	BIT1	(bit4)
PFL	Parity flag	BIT1	(bit2)
CFL	Carry flag	BIT1	(bit0)

Table 6-13 Peripheral Control Block (PCB) Registers

Keyword	Register Name	Offset from Base
RELREG	Relocation Register	0FEH ORD2
PDCON*	Power-Down Mode Control Register	0F0H ORD2
EDRAM*	Enable DRAM Refresh	0E4H ORD2
MDRAM*	Memory Address for DRAM Refresh	0E0H ORD2
CDRAM*	Clock Register for DRAM Refresh	0E2H ORD2
	DMA Descriptor Channel 1	
DMA16	Control Word	0DAH ORD2
DMA15	Transfer Count	0D8H ORD2
DMA14	Destination Pointer (High)	0D6H ORD2
DMA13	Destination Pointer (Low)	0D4H ORD2
DMA12	Source Pointer (High)	0D2H ORD2
DMA11	Source Pointer (Low)	0D0H ORD2
	DMA Descriptor Channel 0	
DMA06	Control Word	0CAH ORD2
DMA05	Transfer Count	0C8H ORD2
DMA04	Destination Pointer (High)	0C6H ORD2
DMA03	Destination Pointer (Low)	0C4H ORD2
DMA02	Source Pointer (High)	0C2H ORD2
DMA01	Source Pointer (Low)	0C0H ORD2
	Chip-Select Control Registers	
CSCTRL5	MPCS Register	0A8H ORD2
CSCTRL4	MMCS Register	0A6H ORD2
CSCTRL3	PACS Register	0A4H ORD2
CSCTRL2	LMCS Register	0A2H ORD2
CSCTRL1	UMCS Register	0A0H ORD2

*These registers are meaningful only when the processor is in Enhanced operation mode.

Table 6-13 Peripheral Control Block (PCB) Registers (continued)

Keyword	Register Name	Offset from Base	
	Timer 2 Control Registers		
TIMER24	Mode/Control Word	66H	ORD2
TIMER23	Unused	64H	
TIMER22	Maximum Count A	62H	ORD2
TIMER21	Count Register	60H	ORD2
	Timer 1 Control Registers		
TIMER14	Mode/Control Word	5EH	ORD2
TIMER13	Maximum Count B	5CH	ORD2
TIMER12	Maximum Count A	5AH	ORD2
TIMER11	Count Register	58H	ORD2
	Timer 0 Control Registers		
TIMER04	Mode/Control Word	56H	ORD2
TIMER03	Maximum Count B	54H	ORD2
TIMER02	Maximum Count A	52H	ORD2
TIMER01	Count Register	50H	ORD2
	Slave Mode (iRMX™)		
INTRPT10	Unused	3EH	
INTRPTF	Unused	3CH	
INTRPTE	Level 5 Control Register	3AH	ORD2
INTRPTD	Level 4 Control Register	38H	ORD2
INTRPTC	Level 3 Control Register	36H	ORD2
INTRPTB	Level 2 Control Register	34H	ORD2
INTRPTA	Level 0 Control Register	32H	ORD2
INTRPT9	Interrupt Controller Status Register	30H	ORD2
INTRPT8	Interrupt Request Register	2EH	ORD2
INTRPT7	In-Service Register	2CH	ORD2

Table 6-13 Peripheral Control Block (PCB) Registers (continued)

Keyword	Register Name	Offset from Base	
	Slave Mode (iRMX™)		
INTRPT6	Priority-Level Mask Register	2AH	ORD2
INTRPT5	Mask Register	28H	ORD2
INTRPT4	Unused	26H	
INTRPT3	Unused	24H	
INTRPT2	Specific EOI Register	22H	ORD2 (write only)
INTRPT1	Interrupt Vector Register	20H	ORD2
	Master Mode (Non-iRMX™)		
INTRPT10	Interrupt 3 Control Register	3EH	ORD2
INTRPTF	Interrupt 2 Control Register	3CH	ORD2
INTRPTE	Interrupt 1 Control Register	3AH	ORD2
INTRPTD	Interrupt 0 Control Register	38H	ORD2
INTRPTC	DMA 1 Control Register	36H	ORD2
INTRPTB	DMA 0 Control Register	34H	ORD2
INTRPTA	Timer Control Register	32H	ORD2
INTRPT9	Interrupt Controller Status Register	30H	ORD2
INTRPT8	Interrupt Request Register	2EH	ORD2
INTRPT7	In-Service Register	2CH	ORD2
INTRPT6	Priority Mask Register	2AH	ORD2
INTRPT5	Mask Register	28H	ORD2
INTRPT4	Poll Status Register	26H	ORD2
INTRPT3	Poll Register	24H	ORD2
INTRPT2	EOI Register	22H	ORD2 (write only)
INTRPT1	Unused	20H	

Discussion

You can access the microprocessor registers at any time when the emulator is halted. If the emulator is running, you can access the registers only if FASTBREAK is set to TRUE.

Use the keywords in Tables 6-12 and 6-13 to display or modify a particular register value. Entering a keyword without an optional expression displays the current value of that register.

Two commands are also provided for displaying registers in groups: REGS and PCB. Refer to the appropriate entries in this chapter for more information.

Unlike the 80C186/80C188 microprocessor, the ICE-186/188 emulator predefines the LMCS, PACS, MMCS, and MPCS chip-select registers upon initialization. It predefines these registers in order to avoid user accesses to inactive registers, which would result in undefined or illegal register values. The ICE-186/188 emulator predefines the registers as follows:

- LMCS (keyword CSCTRL2) -- 0038H; memory block size 1K; 0 wait states; external READY used.
- PACS (keyword CSCTRL3) -- 0CF8H; memory base address of 0CC00H; 0 wait states; external READY used.
- MMCS (keyword CSCTRL4) -- 41F8H; base address 40000H; 0 wait states; external READY used.
- MPCS (keyword CSCTRL5) -- 81B8H; total memory block size 8K; individual select size 2K; peripherals mapped into I/O space; 7 /PCS lines; A1 and A2 not provided.

You can change the predefined values of the chip-select registers by using the appropriate register keywords. The new values remain in effect until you change them or re-initialize the emulator through the ACTIVATE or RESET ICE command. Re-initialization of the emulator resets the registers to the predefined values. The registers are also reset to the predefined values whenever power is cycled to the prototype.

Register Access (continued)

Example

This example displays the RELREG value, then modifies it.

```
hlt> RELREG
20FFH
hlt> RELREG = 30FFH
```

Cross-References

FASTBREAK
Flags
PCB
REGS

Register Access 80C187

Displays and modifies
80C187 numeric processor
extension unit registers

Syntax

187-register [= *expr*]

Where:

187-register displays the current value of the 80C187 numeric processor extension unit register and is one of the keywords listed in Table 6-15.

expr is a value (of the correct data type) used to set the contents of an 80C187 numeric processor extension unit register.

Table 6-15 80C187 Unit Registers

Keyword	Description	Data Type
ST0*	Internal Stack Register 0 (top of stack)	REAL10
ST1*	Internal Stack Register 1	REAL10
ST2*	Internal Stack Register 2	REAL10
ST3*	Internal Stack Register 3	REAL10
ST4*	Internal Stack Register 4	REAL10
ST5*	Internal Stack Register 5	REAL10
ST6*	Internal Stack Register 6	REAL10
ST7*	Internal Stack Register 7	REAL10
FSW	Status Word	ORD2
FCW	Control Word	ORD2
FTW	Tag Word	ORD2
FIA	Instruction Address	ORD4
FDA	Data Address	ORD4
FIO	Instruction	ORD2

*These registers can be displayed only if an 80287 coprocessor is installed on your IBM PC AT system board.

Discussion

The 80C187 numeric processor extension unit is designed for use with the 80C186 microprocessor. It provides floating-point, extended-integer, and binary-coded decimal data types and adds over 50 mnemonics to the microprocessor instruction set.

To display or modify 80C187 registers, you must set the COPROC187 tool variable to TRUE. If COPROC187 is set to FALSE, the emulator assumes there is no unit in the prototype and prevents 80C187 register access.

To display the stack registers (ST0 - ST7), you must have an 80287 coprocessor installed on your IBM PC AT system board. This restriction applies only to the display of the stack registers; you can modify the registers regardless.

The register accesses can only be used when the emulator is halted or servicing interrupts.

Cross-References

COPROC187
<expr>
R187

REGS

Displays the contents of microprocessor registers and flags register

Syntax

REGS

Discussion

Use the REGS command to display the contents of the microprocessor registers and flag.

The setting of the FASTBREAK tool variable affects the operation of REGS. If you want to use the REGS command while the emulator is running, set FASTBREAK to TRUE. The REGS command is not allowed during emulation if FASTBREAK is FALSE.

The REGS command displays the general-purpose registers, the stack and base pointers, the instruction pointer, the flags register, and the segment registers.

Example

This example illustrates the display of microprocessor registers.

```
hlt> REGS
AX=3e51H    CS=1000H    IP=0051H
BX=94faH    SS=1600H    SP=0176H
CX=cefaH    ES=0000H    DI=0000H
DX=0001H    DS=1300H    SI=edbcH
BP=017aH
RELREG=     30ffH
FLAGS=      f006H  PFL
```

Cross-References

FASTBREAK
Flags
GO
Register Access

RELEASEIO

Allows pending I/O requests to be serviced

Syntax

```
RELEASEIO
```

Discussion

I/O requests are pending whenever a HOLDIO command is entered or when a mapped I/O read or write has occurred in Run mode. These pending requests disable interrupt servicing, regardless of the setting of ENI. The RELEASEIO command allows you to service pending I/O requests so that the emulator can resume execution.

If the HOLDIO command caused suspension of the I/O request, enter RELEASEIO to resume the execution of the last command entered before HOLDIO. You will be prompted for an I/O input.

If the pending I/O request was caused by a mapped I/O read or write during Run mode, enter RELEASEIO to resume execution of the GO command. The Activity Monitor programming will remain unaltered.

Example

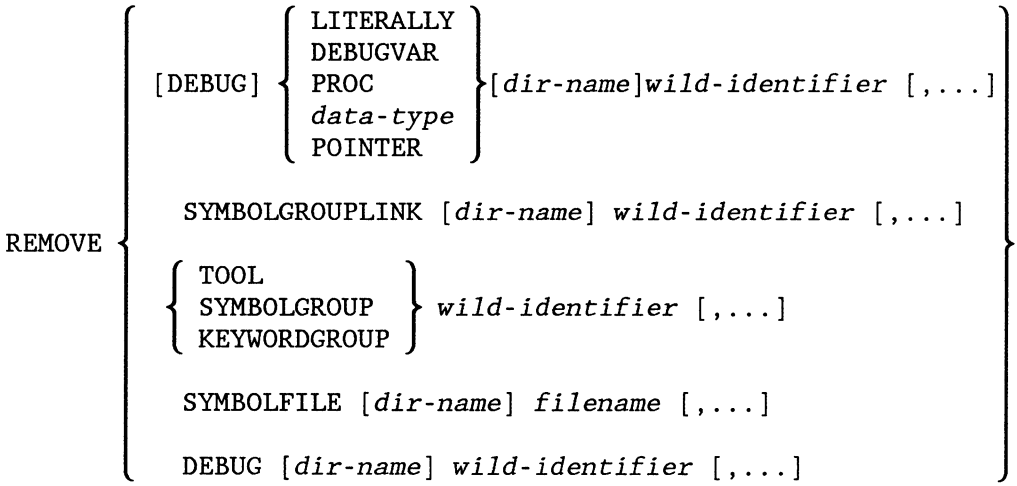
This example stops the temporary suspension of I/O requests and displays the prompt for input data.

```
IO?> RELEASEIO
Port 0002H requests word input (enter value):
```

Cross-References

GO
HOLDIO
MAPIO

Syntax



Where:

REMOVE	deletes an object from the emulator database that is no longer required.
DEBUG	specifies the debug object to be removed. If a debug type (LITERALLY, DEBUGVAR, or PROC) is specified, only debug objects of that type are removed.
LITERALLY	specifies that only LITERALLYs are to be removed.
DEBUGVAR	specifies that only debug variables are to be removed.
PROC	specifies that only user-defined procedures are to be removed.
data-type	specifies that only debug objects of the named *data-type* are removed (see <data type> in this chapter).
POINTER	specifies that only a memory pointer type is to be removed.

REMOVE (continued)

dir-name	specifies an emulator object directory such as LOCAL// or GLOBAL//.
wild-obj-identifier	specifies the name of a debug object. The * and ? can be used as wild characters.
filename	specifies the path and filename of a host development system file.
TOOL	specifies that the tool, (which has been previously defined and deactivated), is to be removed.
SYMBOLGROUP	specifies that a symbolgroup is to be removed (see Object Hierarchy in this chapter).
KEYWORDGROUP	specifies that a keywordgroup is to be removed (see Object Hierarchy in this chapter).
SYMBOLFILE	specifies that a symbolfile that was previously defined or attached to the emulator database with a LOAD command is to be removed (see Object Hierarchy in this chapter).

Discussion

The REMOVE command is used to remove debug objects that were defined with the DEFINE or REDEFINE command. A warning message is displayed if the *wild-obj-identifier* specified does not exist. If more than one *wild-obj-identifier* is specified, no warning is given if one or more, but not all, *wild-obj-identifiers* do not exist. The DIR command can be used to verify removal of the specified objects.

Examples

1. Remove a previously defined LITERALLY named len.

```
hlt> REMOVE LITERALLY len
hlt>
```

2. Remove all DEBUGVARs starting with the letters num (use the asterisk (*) wild character).

```
hlt> REMOVE DEBUGVAR num*
hlt>
```

Cross-References

<data type>
DEACTIVATE
DEFINE
DIR
Object Hierarchy
REDEFINE
SHOW

REPEAT

Groups and executes commands forever
or until an exit condition is met

Syntax

```
REPEAT
   [ WHILE expr ]
   [ UNTIL expr ]
   [ commands   ]
END[REPEAT]
```

Where:

REPEAT	specifies a loop statement which executes commands in blocks.
WHILE *expr*	continues to execute when *expr* is TRUE (non-zero). The loop terminates when *expr* is FALSE (zero). This form of WHILE differs from the WHILE command (see WHILE in this chapter).
UNTIL *expr*	continues to execute when *expr* is FALSE (zero). The loop terminates when *expr* is TRUE (non-zero).
commands	are one or more emulator commands. The following commands are not executable inside the REPEAT command: DO, IF, FOR, SWITCH, and INCLUDE.
END[REPEAT]	specifies the end of the REPEAT block and starts execution. The optional REPEAT keyword is used to label the block type.

Discussion

The REPEAT statement is executed immediately after the END statement is entered. A REPEAT block not containing WHILE or UNTIL clauses is executed forever or until it is aborted with a <Ctrl>C or <Ctrl><Break>. Multiple WHILE and UNTIL statements can be entered in a single REPEAT block. A REPEAT block containing WHILE or UNTIL exits the block (at the END statement) when any of the test conditions are satisfied.

Example

The following example uses the REPEAT command to single step 15 times while incrementing an integer value.

```
hlt> DEFINE INT2 var = 0
hlt> REPEAT
.hlt>> WHILE var < 15
.hlt>> ISTEP
.hlt>> var++
.hlt>> END
/* display */
hlt>
```

Cross References

COUNT
DO...WHILE
<expr>
FOR
WHILE

RESET

Re-initializes specified functions of the emulator

Syntax

```
        { MAP   }
        { MAPIO }
RESET   { ICE   }
        { REGS  }
        { UNIT  }
```

Where:

RESET	resets the specified parameter.
MAP	resets the memory map to its default setting: all memory mapped to USER with read/write access. This parameter can be executed only when the emulator is in Halt mode.
MAPIO	resets the I/O map to its default setting: all ports mapped to USER. This parameter can be executed only when the emulator is in Halt mode.
ICE	resets the emulator controller unit to its original condition, and resets all functions supported by the emulator to the default settings. USER-mapped memory contents are not affected; however, ICE-mapped memory contents may be lost. This parameter can be executed at any time. Note that this parameter resets the processor chip-select registers to values predefined by the emulator. For a description of these values, refer to the Register Access entry in this chapter.

REGS — resets the processor registers without activating the processor /RES input line. The reset values are:

CS = 0FFFFH
IP = 0
FLAGS = 0F002H
Relocation Register = 20FFH

All other registers, except for the Peripheral Control Block registers, are set to 0. The Peripheral Control Block registers are not affected by this parameter.

This parameter can be executed only when the emulator is in Halt mode.

UNIT — resets the processor by toggling the processor /RES input line. This parameter can be executed at any time, but it only takes effect when the emulator is in Run mode or in Halt mode with ENI=2 or ENI=1.

Discussion

Use the RESET command to re-initialize specific emulator functions. You can list one or more reset parameters with the command.

Example

This example resets the memory and I/O maps to the default settings.

```
hlt> RESET MAP MAPIO
hlt>
```

Cross-References

MAP
MAPIO
PCB
Register Access
REGS

RSTEN

Enables and disables external reset of the processor during emulation

Syntax

RSTEN [= *bool-expr*]

Where:

RSTEN	if entered without an optional expression, displays the current RSTEN setting.
bool-expr	is any value that evaluates to TRUE (non-zero) or FALSE (zero).

Default

TRUE

Discussion

Use the RSTEN tool variable to enable and disable external processor resets during emulation.

If you set RSTEN to TRUE, external resets are enabled and the processor acknowledges the /RES signal from the prototype. If you set RSTEN to FALSE, /RES is ignored by the processor.

(Note that /RES is always ignored in Halt mode when ENI is set to 0 or 1.)

You can display the setting of the RSTEN variable at any time, but can change it only in Halt mode.

Example

The following example displays the current RSTEN setting, then sets the variable to FALSE.

```
hlt> RSTEN
TRUE
hlt> RSTEN = 0
```

Cross-References

<expr>
TOOLVAR

SAVE

Saves contents of memory
in an object file on disk

Syntax

SAVE *pathname partition*

Where:

SAVE	saves the specified memory partition in an object file on disk.
pathname	is the fully qualified reference to the object file.
partition	identifies the location of the memory to be saved. It is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-bytes*.

Discussion

Use the SAVE command to transfer memory contents to a file on the PC disk.

No assumptions are made about the format of the memory. SAVE formats the data in the saved file in Intel 8086 absolute Object Module Format with no debug or type definition information included. Once the data is saved on disk, you can load it into mapped memory by using the LOAD command (see the LOAD entry in this chapter).

If the file specified by *pathname* already exists, the emulator displays the following message:

```
pathname already exists, overwrite[y/n]?
```

If you respond with "y", the emulator overwrites the current file.

Example

This example saves the contents of the specified memory section to a file called "myprog" in directory "workdir".

```
hlt> SAVE c:\workdir\myprog 200:100H LENGTH 300
hlt>
```

Cross-References

LOAD
Partition

SELFTEST

Runs the customer confidence tests

Syntax

```
SELFTEST
```

Discussion

Use the SELFTEST command to run the customer confidence tests. These tests verify the operation of the emulator hardware and detect any catastrophic failures.

You can use the SELFTEST command at any time after the emulator software has been invoked. Be aware, however, that the SELFTEST command resets the emulator hardware and firmware to the power-up states. It erases any user-entered environment settings, such as memory and I/O maps.

SELFTEST requires that the emulator user probe be connected to the crystal power accessory (CPA). This way, the emulator operates in a stand-alone mode, where the CPA serves as the target.

For detailed instructions on how to install the CPA and run SELFTEST, refer to the *ICE™-186/188 In-Circuit Emulator Installation Supplement.*

Example

This example illustrates how SELFTEST works. As each confidence test is run, the test name is displayed on the screen along with a "Passed" or "Failed" designation.

```
hlt> SELFTEST
Stand by, downloading User Confidence Test software:.......
     ICE-186 Confidence Test:
     Crystal Power Accessory Required
     TEST#          Title                                  Status
      1             BASIC FUNCTIONALITY TEST               Passed.
      2             WORD RECOGNIZER RAM TEST               Passed.
                              .
                              .
                              .
     18             EXTENDED TRACE TEST                    Passed.
WARNING...emulator has been reset to default activation state
hlt>
```

You can exit SELFTEST at any time by pressing <Ctrl><Break>.

If you encounter a test failure, contact your Intel service representative. Service information is provided on the inside-back cover of this manual.

Cross-Reference

ICE™-186/188 In-Circuit Emulator Installation Supplement

SHOW

Displays the definition of debug objects

Syntax

```
                 [ data-type  ]
                 | DEBUGVAR   |
SHOW [DEBUG]     | LITERALLY  |  [dir-name] [wild-obj-id [,...]]
                 | POINTER    |
                 [ PROC       ]
```

Where:

SHOW	If entered without options, displays all the DEFINEd debug objects.
DEBUG	entered without a specified debug object type, is the same as SHOW <Enter>.
data-type	displays the debug variables and the current value of the specified *data-type* objects (see <data type> in this chapter).
DEBUGVAR	displays all the definitions and current values of debug variables.
LITERALLY	displays all the LITERALLY definitions.
POINTER	displays all POINTER definitions.
PROC	displays all the procedure definitions.
dir-name	specifies an emulator object directory such as LOCAL// or GLOBAL//.
wild-obj-id	specifies the name of a user-defined debug object. The * and ? can be used as wild characters.

Discussion

The SHOW command is used to display the definition of debug objects currently in the debug environment. For alias definitions (LITERALLYs), the LITERALLY name and definition are displayed.

For debug variables, the type, variable name, and current value are displayed. For debug PROCS, the PROC name, PROC type, argument list, and text are displayed.

Examples

1. Show the debug LITERALLY that was DEFINEd as len.

```
-> SHOW LITERALLY len
redefine literally len = "length"
->
```

2. Show the debug variables that are DEFINEd.

```
-> SHOW DEBUGVAR
redefine int2 abc = 0T
redefine bool1 value = TRUE
->
```

3. Show a debug PROC called power.

```
-> SHOW PROC power
redefine proc power(arg1, arg2)
define int1 arg1
define int1 arg2
{
define int1 index
define ord4 result = 1
for (index = 1; index <= arg2; ++index)
result=result * arg1
return(result)
}
->
```

Cross-References

DEFINE
DIR
REDEFINE

STATUS / S

Displays the current emulator status

Syntax

STATUS
S

Discussion

Use the STATUS or S command to display the current status of the emulator and Activity Monitor. You can enter the command at any time.

Example

This example queries for status information after the invocation of a GO command. Status information is automatically displayed after the STOP and HALT command entries.

```
hlt> QUERY = 1
hlt> GO FOREVER
   Use S (status) command to determine emulation status.
emu+> STATUS
   Emulator Status:  running user code.
   Activity Monitor Status:  Running.
emu+> STOP
   Emulator Status:  running user code.
   Activity Monitor Status:  Stopped.
emu> HALT
   Emulator Status:  halted (not servicing interrupts).
     Execpointer is :MESSG2.DELAY_A_LITTLE_BIT_#162  (1000:004eH)
   Activity Monitor Status:  Stopped.
hlt>
```

Cross-References

CAUSE
GO
QUERY

STEP

Steps through a program by machine-language instructions

Syntax

```
STEP [FROM address] [step-size] [ CALL  ] [ DASM  ] [NPROMPT]
                                [ NCALL ] [ NDASM ]
```

Where:

STEP	executes one or more machine-language instructions.
FROM *address*	specifies a starting address where STEP is to begin. The default starting address is the current execution pointer.
step-size	is an unsigned integer expression specifying the number of instructions to be included in a step. The default *step-size* is 1 instruction; the maximum is 255T instructions.
CALL | NCALL	specifies how call instructions are to be counted for *step-size*. If CALL is chosen, all statements and calls within a call are counted as a single instruction. If NCALL is chosen, the stepping is one statement at a time. NCALL is the default setting.
DASM | NDASM	controls the disassembly of instructions. DASM causes disassembly, while NDASM suppresses disassembly. DASM is the default setting.
NPROMPT	terminates the command after one step.

Discussion

Use the STEP command to single-step through the machine-language instructions of your program.

You can control the starting address and size of the steps by using the FROM *address* and *step-size* parameters, respectively. The FROM *address* parameter specifies the starting address of the first step;

ongoing steps start from where the previous step left off. The *step-size* parameter specifies the size of the steps in terms of instructions. You can specify from 1 to 255T instructions per step.

The NPROMPT parameter allows you to specify whether tne Step mode is ongoing or complete after the initial STEP execution. If NPROMPT is not entered, the Step mode is ongoing. After each step is executed, a screen message asks if you want to step again or return to Halt mode. Press the <Enter> key to step again; press the E key to return to Halt mode.

Note that the STEP command does not step through instructions that move data into a segment register (i.e., CS, DS, ES, or SS). The move instruction and its immediately following instruction are stepped as one.

When you use STEP, the step breaks are the only breakpoints acknowledged by the emulator. The Activity Monitor trace is turned off, and the emulator ignores any breakpoints established by the GO command.

You can execute the STEP command only when the emulator is in Halt mode.

Example

This example steps through five instructions and then terminates the command.

```
hlt> STEP FROM :messg.main_#121 5 NPROMPT
:MESSG.INITIALIZE_BUFFER_#121
1000:001fH   56          PUSH   SI
1000:0020H   57          PUSH   DI
1000:0021H   55          PUSH   BP
1000:0022H   8bec        MOV    BP,SP
:MESSG.INITIALIZE_BUFFER_#135
1000:0024H   8b7e0a      MOV    DI,WORD PTR [BP]+0aH
hlt>
```

Cross-References

ISTEP
LSTEP
Prompts
PSTEP

STOP

Stops the Activity Monitor
without stopping emulation

Syntax

STOP

Discussion

Use the STOP command to stop Activity Monitor functions without halting emulation. You can enter the command at any time.

The default state of the Activity Monitor is to trace continuously. Use STOP in this case to stop the acquisition prior to viewing the trace buffer. STOP is also useful when the Activity Monitor does not break as expected due to breakpoint events not occurring.

Examples

1. In this example, the emulator is started using the default Activity Monitor programming (TRACE FOREVER). The prototype hangs up. STOP is used to stop the Activity Monitor (and acquisition) so that the trace buffer can be PRINTed.

```
hlt> QUERY = 1
hlt> GO
   Use S (status) command to determine emulation status.
emu+> STOP
   Emulator Status:  running user code.
   Activity Monitor Status:  Stopped.
emu> PRINT ALL

...trace buffer is displayed...

emu>
```

2. In this example, the STOP command is invoked after searching for an event (ISYNCT) that never occurred.

```
hlt> QUERY = 1
hlt> GO TIL ISYNCT
   Use S (status) command to determine emulation status.
emu+> S
   Emulator Status:  running user code.
   Activity Monitor Status:  Running.
emu+> S
   Emulator Status:  running user code.
   Activity Monitor Status:  Running.
emu+> STOP
   Emulator Status:  running user code.
   Activity Monitor Status:  Stopped.
emu>
```

Cross-References

CAUSE
GO
HALT
STATUS / S

String Functions

Built-in functions for
string manipulation

Discussion

The following string manipulation functions are discussed in this section:

STRNCMP	STRNCPY	STRNCAT	STRLEN
STRCMP	STRCPY	STRCAT	

Six of the functions are related in pairs: STRCMP and STRNCMP, STRCPY and STRNCPY, and STRCAT and STRNCAT. The function names with "N" as a middle letter (STRNCMP, STRNCPY, STRNCAT) operate on a sub-string whose length is defined by the third argument. The other functions (STRCMP, STRCPY, STRCAT) operate on the entire source string.

STRLEN Function

The STRLEN function returns the length of an ASCII string, excluding the NULL terminating character (if any). The syntax is as follows:

```
[obj-identifier =] STRLEN (string-expr)
```

Where:

obj-identifier	specifies the debug object to which the function's return value is assigned. If obj-identifier is not specified, the return value is displayed on the next line of the screen..
string-expr	specifies an NSTRING or LSTRING variable, a quoted string constant, or an address of a string.

The following example illustrate the function STRLEN:

```
-> BASE = 10T
-> DEFINE NSTRING month = "October"
-> DEFINE NSTRING year = "1986"
-> DEFINE INT4 ans
-> ? STRLEN (month)
7T
-> ans = STRLEN (year)
-> ans
4T
```

STRCMP Function

The STRCMP function compares two ASCII strings, character by character. The comparison stops when a mismatch is found or when a NULL is encountered in one of the strings.

The return value is the difference between the hexadecimal values of the characters at the stopping position. The character from the second string (*string-expr2*) is subtracted from the character from the first string (*string-expr1*). This value is less than zero, zero, or greater than zero, giving the lexicographical difference between the two strings. The syntax is as follows:

[*obj-identifier* =] STRCMP (*string-expr1*, *string-expr2*)

Where:

obj-identifier	specifies the debug object to which the function's return value is assigned. If obj-identifier is not specified, the return value is displayed on the next line of the screen.
string-expr1	specifies an NSTRING or LSTRING variable, a quoted string constant, or an address of a string.
string-expr2	specifies an NSTRING or LSTRING variable, a quoted string constant, or an address of a string.

The following example illustrates the function STRCMP:

```
-> BASE = 10T
-> DEFINE NSTRING name1 = "Rosenberg"
-> DEFINE NSTRING name2 = "Rosenbaum"
-> DEFINE INT4 order = STRCMP (name1, name2)
-> order
4T
-> STRCMP (name2, name1)
-41
```

STRNCMP Function

The STRNCMP function compares a specified maximum number of characters (*expr*) in two ASCII character strings. The comparison stops when a mismatch is found, when a NULL is encountered in one of the strings, or when the specified number of character positions have been compared.

The return value is the difference between the hexadecimal values of the characters at the stopping position. The character from the second string is subtracted from the character from the first string. This value is less than zero, zero, or greater than zero, giving the lexicographical difference between the strings. The syntax is as follows:

[*obj-id* =] STRNCMP (*string-expr1*, *string-expr2*, *expr*)

Where:

obj-id	specifies the debug object to which the function's return value is assigned. If obj-id is not specified, the return value is displayed on the next line of the screen..
string-expr1	specifies an NSTRING or LSTRING variable, a quoted string constant, or an address of a string.
string-expr2	specifies an NSTRING or LSTRING variable, a quoted string constant, or an address of a string.
expr	specifies a number or an expression which specifies the maximum number of characters to compare.

The following example illustrates the function STRNCMP:

```
-> BASE = 10T
-> DEFINE NSTRING name1 = "Rosenbaum"
-> DEFINE NSTRING name2 = "Rosenberg"
-> DEFINE NSTRING name3 = "Rosen"
-> STRNCMP (name1, name2, 7)
-4T
-> STRNCMP (name1, name3, 9)
98T
-> STRNCMP (name1, name3, 3)
0T
```

STRCPY Function

The STRCPY function copies the second string (*string-expr*) into the first string (*addr-expr*), until the second string's terminating NULL character is copied. It overwrites any data that may be in the first string. The second string remains unchanged. The syntax is as follows:

STRCPY (*addr-expr*, *string-expr*)

Where:

addr-expr	specifies an address (e.g., an NSTRING or LSTRING variable referenced with the & operator).
string-expr	specifies an NSTRING or LSTRING variable, a quoted string constant, or an address of a string.

The following example illustrates the function STRCPY:

```
-> BASE = 10T
-> DEFINE NSTRING month = "October"
-> DEFINE NSTRING year = "1986"
-> DEFINE NSTRING date
-> STRCPY (&date,  month)
-> date
"October"
-> STRCPY (&date, year)
-> date
"1986"
```

STRNCPY Function

The STRNCPY function copies the specified maximum number of characters (*expr*) from the second string (*string-expr*) to the first string (*addr-expr*). Copying stops when a NULL terminating character is copied or when the number of characters specified (*expr*) have been copied. If *expr* is greater than the length of *string-expr*, the *addr-expr* resulting from the copy is *string-expr*. The syntax is as follows:

```
STRNCPY (addr-expr, string-expr, expr)
```

Where:

addr-expr	specifies an address (e.g., an NSTRING or LSTRING variable referenced with the & operator).
string-expr	specifies an NSTRING or LSTRING variable, a quoted string constant, or an address of a string.
expr	specifies a number or an expression which specifies the maximum number of characters to copy.

The following example illustrates the function STRNCPY:

```
-> BASE = 10T
-> DEFINE NSTRING month = "October"
-> DEFINE NSTRING year = "1986"
-> DEFINE NSTRING date
-> STRNCPY (&date, month, 3)
-> date
"Oct"
-> STRNCPY (&date, year, 7)
-> date
"1986"
```

STRCAT Function

The STRCAT function appends the second string (*string-expr*) to the right end of the first string (*addr-expr*). Copying of the second string starts from the left end of the string and continues until a NULL terminating character is copied. The second string is left unchanged. The syntax is as follows:

STRCAT (*addr-expr*, *string-expr*)

Where:

addr-expr	specifies an address (e.g., an NSTRING or LSTRING variable referenced with the & operator).
string-expr	specifies a pointer expression (address), an NSTRING or LSTRING variable, or a string constant.

The following example illustrates the function STRCAT:

```
-> BASE = 10T
-> DEFINE NSTRING month = "10"
-> DEFINE NSTRING date = "22."
-> STRCAT (&date, month)
-> date
"22.10"
```

STRNCAT Function

The STRNCAT function appends the specified number of characters (*expr*) from the second string (*string-expr*) to the right end of the first string (*addr-expr*). Copying of the second string starts from the left end of the string and continues until a NULL terminating character is copied or the specified number of characters have been copied. The second string is left unchanged. The syntax is as follows:

STRNCAT (*addr-expr*, *string-expr*, *expr*)

Where:

addr-expr	specifies an address (e.g., an NSTRING or LSTRING variable referenced with the & operator).
string-expr	specifies a pointer expression (address), an NSTRING or LSTRING variable, or a string constant.
expr	specifies a number or an expression which specifies the maximum number of characters to concatenate.

The following example illustrates the function STRNCAT:

```
-> BASE = 10T
-> DEFINE NSTRING month = "10.86"
-> DEFINE NSTRING date = "22."
-> STRNCAT (&date, month, 2)
-> date
"22.10"
->
```

Cross-References

C: A Reference Manual	Functions
DEFINE	PROC
<expr>	REDEFINE

SWITCH

Causes execution to branch to one of several CASE statements or the DEFAULT statement

Syntax

```
SWITCH (expr)
  {
  CASE label-expr: [commands; ] [...]
  [ DEFAULT: commands; ]
  }
```

Where:

SWITCH	begins the SWITCH control block and causes the evaluation of *expr*.
expr	specifies a number or an expression. The value of *expr* is compared to the value of the label in each CASE statement.
{ }	Braces are used to group all of the CASE statements and one optional DEFAULT statement.
CASE	indicates the beginning of one case statement.
label-expr:	specifies a number or an expression whose value is compared to *expr*. The colon (:) is required punctuation.
commands	are any emulator commands, including BREAK (which causes an immediate exit from the SWITCH construct). The INCLUDE command cannot be used.
DEFAULT:	specifies the statement that is executed if none of the CASE statement *label-expr*: values match that of *expr*. The colon (:) is required punctuation.

Discussion

The SWITCH statement transfers execution control to the command(s) following the CASE label-expr: statement whose *label-expr*: value matches the value of the SWITCH expression. If no CASE *label-expr*: matches, no commands are executed unless there is a DEFAULT statement. There can be only one DEFAULT statement. Once command execution begins at a CASE *label-expr*:, it continues through all remaining CASE commands unless the BREAK command is encountered.

CASE statements can occur in any order, but the DEFAULT statement, if any, must be last.

Examples

1. The following example shows the format of a SWITCH statement.

```
-> SWITCH (entry)
->> {
->> CASE 1: BASE = 2T; BREAK
->> CASE 2: BASE = 10T; BREAK
->> CASE 3: BASE = 16T; BREAK
->> DEFAULT: BASE
->> }
->
```

2. This example shows SWITCH used in a PROC. The PROC gets an integer (INT4) from the keyboard using SCANF and compares it to the CASEs. When there is a match, another PROC is called. When there is no match, DEFAULT is executed; assume proc1, proc2, proc3, and proc4 have been previously defined.

```
-> DEFINE INT4 response
-> REDEFINE PROC menux ()
-> {
->> PRINTF ("Enter a number: ")
->> SCANF("%d",&response)
->> SWITCH (response)
```

```
->> {
->> CASE 1:   proc1
->> BREAK
->> CASE 2:   proc2
->> BREAK
->> CASE 3:   proc3
->> BREAK
->> CASE 4:   proc4
->> BREAK
->> DEFAULT:  MOVE (17,5)
->> PRINTF("Not a valid choice \n")
->> BREAK
->> }
->> }
->
```

Cross-References

BREAK	<expr>
CONTINUE	PROC
DEFINE	REDEFINE

SYMBOLIC

Enables/disables symbolic display

Syntax

```
SYMBOLIC  [= expr]
```

Where:

SYMBOLIC	is a parameter that controls the display of symbolic information. If entered without an optional expression, it displays the current SYMBOLIC setting.
expr	is any expression that evaluates to TRUE (non-zero) or FALSE (zero). Default is TRUE.

Discussion

SYMBOLIC is a parameter that determines whether symbolic information will be displayed in the ASM, CAUSE, GO, HALT, ISTEP, LSTEP, PRINT, PSTEP, STATUS, and STEP commands.

When SYMBOLIC is TRUE, available symbolic information is displayed. When FALSE, no symbolic information is displayed.

Cross-References

ASM
CAUSE
GO
HALT
ISTEP
LSTEP
PRINT
PSTEP
STATUS / S
STEP

Syntax

There are nine symbolic reference forms:

1. References to program modules:

   ```
   :module
   ```

. References to program labels:

   ```
   [:module.][procedure.][...] label
   ```

3. References to procedures:

   ```
   [:module.][procedure.][...] procedure
   ```

4. References to line numbers:

   ```
   [:module] line-number
   ```

5. References to variables:

   ```
   [:module.][procedure.][...] variable
   ```

6. References to array variables:

   ```
   [:module.][procedure.][...]variable [array-expr][...]
   ```

7. References to fields in a record or structure:

   ```
   [:module.][procedure.][...] record.field [.field]
   ```

8. Changing the value of a variable:

   ```
   variable-reference = expr
   ```

9. Indirect referencing through a pointer:

   ```
   *[:module.][procedure.][...] ptr-variable
   ```

Symbolic References (continued)

Where:

module *procedure* *label* *variable* *ptr-variable* *record* *field*	are names of program objects that follow the rules for identifiers.
line-number	is one or more decimal digits preceded by a #.
[*array-expr*[,...]]	is a list of one or more expressions identifying an element in an array. The list is enclosed in square brackets (the required pair of brackets is underlined to distinguish them from the inner brackets indicating the optional part of the reference).
variable-reference	is a reference to a variable, an array variable, or a field in a record or structure.
expr	converts, if necessary, to the type of the variable in the *variable-reference*.

Discussion

A program symbol table file contains the names of all objects in the program, including the type and (for some objects) the length of each object. A symbolic reference identifies an object by name. When a symbolic reference is used in a command or expression, the value corresponding to the object is returned. The value returned depends on the type of the object. This section reviews the kinds of symbolic references and the values these represent. This section also discusses a special operator, the address of operation (indicated by the ampersand (&) or the period (.)), which is used with symbolic references.

The emulator supports C-86, PL/M-86, Pascal-86, Fortran-86, and ASM86 Intel languages. The handling of symbolic references is compiler-dependent; some compilers may prefix or append characters to the symbol names in a user program. Use the DIR command to view the symbol file, and review how symbols are referenced for your language. This discussion uses PL/M as the high-level language in its examples.

Symbol Table

The LOAD command reads information about the program symbols from the object file named in the LOAD command. Then, unless you suppress symbolic information by using the NOSYMBOLS option, the LOAD command stores this information in a symbol file called "loadfile.SYM". This file will reside in the same directory as "loadfile".

NOTE

If a flexible disk containing the symbol table file is removed, a fatal error will occur when a symbolic reference requires access to the symbol table file.

To utilize the full symbolic capabilities of the emulator, modules must be compiled with the optimization switches turned off and the compiler switches for debug information turned on. (Refer to the compiler or assembler manual for details.) Compilers also generate line numbers. Line number information is LOADed with the program when this information is available.

The symbol file preserves scope and type information (unless the NOTYPE option is used) specified in the program file. When the type of a variable cannot be determined from the file, the emulator software assigns it a NULL type. Operations that require type information are invalid with NULL types.

Names

All symbolic references except line numbers involve the names of objects. The character sets which identify objects are referred to as names. Command keywords, debug object names, and program symbols are all names. The NAMEFRAME, NAMEPATH, and NAMESCOPE parameters are available for making name usage easier.

References to Program Addresses

References to modules, procedures, labels, and line numbers represent addresses in the program. The value returned is of a pointer type. In each case, the value represents the address of the first executable instruction in the module, in the procedure, at the label, or at the line number.

Modules. A module is identified with a leading colon (:). For example, if "tca" is the name of a module, the reference is entered as follows:

```
hlt> :tca
1ca7:0000H
```

The pointer value returned is the first executable address in the module.

Line Numbers. A line number reference is the line number preceded by a number sign (#). The module name may be required as qualification. The following examples illustrate line number references:

```
hlt> #14
1ca7:001aH

hlt> :tcainv#35
1ca7:04d9H
```

The NAMESCOPE parameter determines when the line number reference requires a module name as qualification. If the address stored in the NAMESCOPE variable is within the desired module, you can omit the module name. After a load and after emulation breaks, NAMESCOPE is updated to the current execution address if the UPDATE parameter is TRUE. (Refer to the NAMESCOPE and UPDATE entries in this chapter for more information.)

Procedures. A procedure reference is the name of the procedure. If necessary, the reference can include the module name and the names of any procedures that enclose the given procedure.

The NAMESCOPE parameter determines when the procedure reference requires qualification (module name and outer procedure names). If the address stored in the NAMESCOPE parameter is within the desired module, you can omit the module name. If the NAMESCOPE value is within any enclosing procedures, you can omit these qualifiers as well. For example:

```
hlt> errch
1ca7:010aH

hlt> :tca.main
1ca7:005bH

hlt> :mod1.outer_procedure.inner_procedure
0100:0200H
```

Labels. Labels identify statements within procedures. Typically, labels are the objects of GOTO statements. They can also be used to control transfer statements within a procedure. The NAMESCOPE parameter determines the qualifiers required to identify a label. For example:

```
hlt> start_over
0100:0234H
hlt> :mod1.first_procedure.start_over
0100:0234H
```

The first example is valid only if the value in NAMESCOPE is within the scope of the procedure "first_procedure" in module "mod1".

References to Program Variables

You can use a reference to a program variable to display or change the contents of the variable, or to use the contents of the variable as an operand in an expression. A reference to a program variable is valid if the variable is active at the current execution point and if it has a type that fits the context of the reference. The following sections describe the kinds of variables that programs can contain.

Symbolic References (continued)

Static Variables. The simplest kind of variable is a static variable representing a single scalar quantity. Static variables are always activ and can always be accessed. The NAMESCOPE parameter determine the kind of qualification required to identify a variable, as discussed previously in the Procedures and Labels sections of this entry. The value returned by the reference depends on the type of the variable.

For example, assume the current program has the following symbols in the SYMBOLFILE// directory (displayed with DIR MODSYMBOL SYMBOLFILE//).

MEMORY	array [0..] of ord1
DENOMINATOR	ord2
I	ord2
PI	real4
SGN	int2
TERMS	ord2
DONE	label

After halting the emulator within this module, references to any of the variables are valid. For example:

```
hlt> pi
3.14159

hlt> terms
5000
```

Dynamic Variables. Dynamic variables are based variables or stack-resident variables. The operating system memory manager allocates space for based dynamic variables as required during program execution. Stack-resident variables are allocated to the stack instead of to fixed memory locations. Examples of stack-resident variables are parameters in procedures, local variables in PL/M REENTRANT procedures, and all local variables in Pascal procedures or C functions

The form of a reference to a dynamic variable is exactly the same as for a static variable. The difference is that if the execution point is not within the procedure that defines the variable, the variable is not active. By contrast, static variables are always active. An error results if you try to access a variable that is not active.

When compiling a procedure, a compiler inserts a sequence of code called the prelude at the beginning of the procedure. The prelude reserves space for stack-resident variables required by the procedure. For stack-based variables to be fully active following a break within a procedure, the break must occur after the prelude.

For example, suppose you load a program named "messg2", and the program has a procedure named "rotate_message_" with a stack-based variable named "index_ptr_". The execution pointer must be within the "rotate_message_" procedure (and past the prelude) before the variable is fully active, as shown below:

```
hlt> /* Emulate until rotate_message_ is reached. */
hlt> GO TIL EXECUTION :messg2.rotate_message_
   Use S (status) command to determine emulation status.
emu+> S
   Emulator Status:  halted (not servicing interrupts).
     Execpointer is :MESSG2.ROTATE_MESSAGE_#146 + 1H  (1000:0067H)
   Activity Monitor Status:  Stopped.
hlt> /* Display the procedure symbols currently active. */
hlt> DIR
TOOL//
    INDEX_PTR_          dynamic int2
    TEMP_               dynamic int1
hlt> /* Try to view the index_ptr_ variable.*/
hlt> index_ptr_

[kernel] Error #26t
Database server error.
[dbm] Error #-210t
No callstack frames exist for tool
hlt> /* LSTEP past the procedure prelude. */
hlt> LSTEP NPROMPT
:MESSG2.ROTATE_MESSAGE_#150   (1000:006eH)
hlt> /* Now view the index_ptr_ variable. */
hlt> index_ptr_
0001H
```

Array Variables. An array consists of elements of a given type. To access an individual element, the reference specifies the index or indexes of the element. For example, suppose the array named "date has four elements, and it is defined (in PL/M) as an array of bytes. You can access any element with a reference such as the following:

```
hlt> date[0]
3H
```

Suppose that element "date[0]" contains the length of the data portion of the array. In other words, the data elements are numbered 1 through the value of "date[0]" (in this case 3). The following debug procedure displays all the elements as ORD1s:

```
hlt> DEFINE PROC dsply {
hlt>> DEFINE ORD1 index = 1
hlt>> COUNT date[0]
.hlt>> date [index]
.hlt>> index = index + 1
.hlt>> END
hlt>> }
hlt> dsply
48H
54H
47H
hlt>
```

To continue one step further, suppose "date" is really an array of ASCII characters. The next example displays these character values, and it also illustrates the use of the & (ampersand) operator to specify the address of a variable. (More details on the & operator are provided later in this entry.)

```
hlt> CHAR &date[1] LENGTH date[0]
```

NOTE

The emulator software does not check array indexes for boundary errors. If an index is outside the bounds of an array, the result is meaningless.

Pascal and C arrays can have more than one dimension. The reference to an element in such an array must have the required number of indexes. For example, if array variable "big_pascal_array" has four dimensions, a reference might be as follows:

```
hlt> big_pascal_array[10,2,34,80]
126H
```

Structure Variables. In C and PL/M, variables with compound elements are called structures. (In Pascal these variables are called records.) The individual elements of a structure are called fields. A reference to a field variable first lists the name of the structure, and then the name of the field. (A dot (.) is used to separate the structure name and the field names.) The NAMESCOPE parameter determines the amount of additional qualification required, just as for other variables.

For example, suppose a PL/M program has a structure declared as follows:

```
DECLARE house STRUCTURE (stories BYTE;
                         rooms BYTE;
                         bathrooms BYTE);
```

After the program is executed so that the variable fields have taken on values, you can access the fields with references such as:

```
hlt> house.stories
2T
hlt> house.rooms
10T
hlt> house.bathrooms
4T
```

(This example assumes that the NAMESCOPE parameter is inside the module and procedure containing the structure.)

Compound Variables. The program can contain compound forms such as arrays of arrays, arrays of structures, and structures of arrays. The rules for references to these compound forms are a combination of the previously discussed rules for variables.

As an example, suppose a PL/M program contains a structure definec as follows:

```
DECLARE table(9) STRUCTURE(
      option(10) BYTE) data(
            8,    'CONTROLS'    ,
            7,    'MODULES'     ,
            5,    'LINES'       ,
            5,    'PROCS'       ,
            5,    'COUNT'       ,
            7,    'NOCOUNT'     ,
            4,    'LIST'        ,
            4,    'SAVE'        ,
            5,    'MERGE'
                                    );
```

References to this structure have forms such as:

```
hlt> table[0].option[0]
8T
```

To display an entire option in this structure, note that the first element in each row gives the number of ASCII characters in the option. For example:

```
hlt> CHAR &table[0].option[1] LENGTH table[0].option[0]
1def:010fH          C O N T R O L S
```

The following debug procedure displays the entire structure:

```
hlt> DEFINE PROC showtable {
hlt>> DEFINE ORD1 index = 0
hlt>> COUNT 9T                          /* Length of the table in decimal */
.hlt>> CHAR &table[index].option[1] LENGTH
.hlt>> table[index].option[0]
.hlt>> index=index + 1
.hlt>> END
hlt>> }
hlt> showtable
1dfe:010fH          C  O  N  T  R  O  L  S
1dfe:0119H             M  O  D  U  L  E  S
1dfe:0123H                   L  I  N  E  S
1dfe:012dH                   P  R  O  C  S
1dfe:0137H                   C  O  U  N  T
1dfe:0141H             N  O  C  O  U  N  T
1dfe:014bH                      L  I  S  T
1dfe:0155H                      S  A  V  E
1dfe:015fH                   M  E  R  G  E
```

Based Variables and Pointer Variables. A based variable is referenced through another variable (called a pointer variable). The pointer variable contains the address of the based variable. An example of a PL/M definition of a pointer is as follows:

```
DECLARE optionptr POINTER;
DECLARE option BASED optionptr BYTE;
```

The equivalent definition of a pointer in the C language is as follows:

```
UNSIGNED CHAR *optionptr;
UNSIGNED CHAR option
```

The executable part of the program then assigns a value to the based variable by setting the pointer to the desired address, and then assigning the value with PL/M statements, for example:

```
optionptr = old_option;
option = 42;
```

You can also assign and display with C statements. The * operator can precede any reference to a program pointer variable. The effect is to obtain the value of the object that the pointer points to.

```
optionptr = &option;
*optionptr = 42;
```

After this code is executed, the value of the pointer is the address of the variable. Assume the following address and contents for the two variables:

Variable	Address of of Variable	Contents of Variable
optionptr	1d00:0420H	1d00:0874H
option	1d00:0874H	42

The following examples illustrate references to these variables. Note that the emulator allows you to refer to the based variable directly.

```
hlt> option
42T
hlt> optionptr
1d00:0874H
hlt> *optionptr
42T
```

Note that for PL/M, the symbol table entry describing the pointer variable does not contain any information about the type of the based variable. To display the contents of the based variable, specify the type with a keyword such as ORD1.

```
hlt> ORD1 optionptr
1d00:0874H                    42
hlt> ORD1 &option
1d00:0874H                    42
hlt> *optionptr
42T
```

(The & operator causes the emulator to return the address of the variable rather than its contents.)

For the C language, pointers and structure references are identical to the syntax required in C programs.

Changing the Value of a Variable

To make debugging easier, you can change the value of a program variable using the emulator. The value assigned is converted to the type of the variable. For example:

```
hlt> :calculatepi.terms = 10000T
hlt>
```

The variable must be active to receive a value. You cannot change the address corresponding to a module, procedure, label, or line number.

Keywords, debug object names, and program symbols are all names. The emulator will let you define a debug object with the same name as a keyword or symbol. The NAMEPATH parameter contains the search order for objects specified without an object type prefix.

In case of a duplication, keywords and debug object names have precedence over symbols (unless the NAMEPATH default has been changed). Thus, if your program has a procedure named "exit", the keyword EXIT masks out that symbol. The following command produces a syntax error:

```
hlt> GO TIL exit
```

To avoid conflict, the symbol must be a fully qualified reference (`:module.exit` in the previous example) or use the prefix SYMBOL// (i.e., GO TIL SYMBOL//exit).

"Address Of" Operator (& and .)

The & or . operator can precede any of the references to program variables. The effect is to return the address of the variable instead of the value of the variable. The & or . operator is used when an address is required, such as when using a data type keyword to override the variable type or when setting a breakpoint on a data address.

Thus, if your program contains an ORD1 variable called "temp_variable" in procedure "getchar", the following reference returns the contents of the variable:

```
hlt> getchar.temp_variable
19T
```

However, the following reference uses the & operator to return the location of the variable:

```
hlt> &getchar.temp_variable
011a:0056H
hlt> .getchar.temp_variable
0110:0056H
```

The & operator should precede the outermost qualifier, including the module name if it is used. For example:

```
hlt> &:tca.start.i
1def:0125H
```

Preludes and Prologues

This section describes preludes and prologues and their effect on symbolic references.

Preludes

Some symbolic references rely on the current register contents to formulate an address that is used in subsequent calculations. For example, stack-based local symbols require that the execution pointer ($) be within the procedure in which the local symbol is declared.

High-level languages manipulate registers upon entry to procedure calls. This entry code is called the "prelude". CALLSTACK and local symbol references will fail if the $ is within the prelude code. You can use PSTEP or LSTEP to step through this prelude code.

Prologues

Some 8086 OMF programs (those built with INITCODE options for LOC86 and LINK86) have a section of starting code that initializes CS, ES, DS, SP, and SS registers. This starting code is called the "prologue".

You must execute the prologue code before making any partially qualified symbol references. The LOAD command gives you the starting address of the prologue. To execute the prologue, enter "GO TIL EXECUTION prgstart", where prgstart is the starting address of code outside the prologue.

Cross-References

Address Translation	NAMEFRAME
DIR	NAMEPATH
<expr>	NAMESCOPE

TOOLVAR

Displays the settings of the tool variables

Syntax

```
TOOLVAR
```

Discussion

Use the TOOLVAR command to display the settings of the following tool variables:

ALEMODE	ALE timing control.
BTHRDY	Source of the READY signal.
BUSACT	Bus activity time-out control.
ENI	Interrupt enable control.
FASTBREAK	Fastbreak emulation control.
IORDY	I/O time-out control.
IPATINT	Source of iPAT interrupt signal.
MEMRDY	Memory time-out control.
OSYNC	Synchronous output level for the next invocation of GO.
QSTAT	Queue status control.
QUERY	Status information control.
RSTEN	Reset control.
VERIFY	Memory write verification.

Example

This example displays the tool variable settings by invoking TOOLVAR.

```
hlt>  TOOLVAR
ALEMODE =START  BTHRDY    =FALSE  BUSACT =TRUE
ENI     =0      FASTBREAK =FALSE
IORDY   =TRUE   IPATINT   =NONE   MEMRDY =TRUE  OSYNC   =TRUE
QSTAT   =TRUE   QUERY     =TRUE   RSTEN  =TRUE  VERIFY  =TRUE
```

Cross-References

ALEMODE
BUSACT
BTHRDY
ENI
FASTBREAK
IORDY
IPATINT
MEMRDY
OSYNC
QSTAT
QUERY
RSTEN
VERIFY

UPDATE

Updates NAMESCOPE when $ is updated

Syntax

```
UPDATE  [= bool-expr]
```

Where:

UPDATE	if entered without an optional expression, displays the current setting.
bool-expr	is a number or an expression that evaluates to TRUE (non-zero) or FALSE (zero). Default is TRUE.

Discussion

The UPDATE variable controls whether the value of NAMESCOPE is updated to match the value of the execution pointer ($) whenever $ changes. If UPDATE is set to TRUE, NAMESCOPE is updated whenever $ is updated. If UPDATE is set to FALSE, NAMESCOPE is not automatically updated. However, NAMESCOPE can still be manually changed using the NAMESCOPE command.

Cross-References

$
<expr>
NAMESCOPE
Parameters

Syntax

```
VERIFY [ = expr]
```

Where:

VERIFY	if entered without an option, displays the current setting.
expr	specifies a number or an expression that evaluates to TRUE (non-zero) or FALSE (zero).

Discussion

Use the VERIFY control variable to specify whether read-back checks (read after write verification) are performed on commands that write to memory (LOAD, ASM, and Memory Access). If VERIFY is set to TRUE, read-back checks do occur. If it is set to FALSE, read-back checks do not occur.

Read-back checks are useful in preventing errors when writes to unwritable memory are attempted, such as memory mapped to hardware registers or EPROM. If an error is detected, the address where the verification failed is reported.

NOTE

Read-back checks increase the time used by the emulator in performing memory writes.

Examples

1. Check the current value of VERIFY.

```
-> VERIFY
TRUE
->
```

VERIFY (continued)

2. Change the value of VERIFY to cause read after memory writes to not occur.

```
-> VERIFY = 0                                    /* or VERIFY = FALSE */
->
```

Cross-References

ASM / SASM
<expr>
LOAD
Memory Access
TOOLVAR

VERSION

Displays the version numbers of the emulator software

Syntax

```
VERSION
```

Discussion

Use the VERSION command to determine what version of the emulator software resides in the host development system and what version of the emulator software resides in the emulator controller unit.

WHILE

Executes a loop while
a condition is true

Syntax

```
WHILE (expr)  { commands [;...] }
```

Where:

WHILE	specifies the beginning of a loop.
(*expr*)	specifies a number or an expression that is evaluated and tested. The loop is repeated until the expression evaluates to FALSE (zero). The parentheses are required punctuation.
{*commands*}	are any emulator commands, except INCLUDE. When there is more than one command, they must be enclosed in braces.

Discussion

This construct causes the specified command(s) to be executed zero or more times repetitively as long as *expr* evaluates to TRUE (non-zero).

Examples

1. While the index is greater than zero, decrement the index and add 5 to the sum on every iteration of the loop.

```
-> DEFINE BYTE i = 5
-> DEFINE BYTE sum
-> WHILE (i > 0)
->> {
->> i -=1
->> sum += 5
->> }
->
```

2. While the index is greater than zero, decrement the index and subtract 10 from the variable, "a". When "a" is less than zero, exit the loop.

```
-> DEFINE BYTE i = 5
-> DEFINE INT2 a = 30
-> WHILE (i > 0)
->> {  i -= 1; a -= 10; IF (a < 0)  BREAK
->> }
->
```

Cross-References

BREAK
CONTINUE
COUNT
DO ... WHILE
<expr>

WPORT

Displays or modifies the contents
of word-wide (16-bit) I/O port access

Syntax

```
WPORT  io-address [= data]
```

Where:

WPORT	displays the contents of the I/O port at the specified *io-address*.
io-address	is a 16-bit expression or number in the current base specifying the physical I/O address. The range of available *i/o-addresses* is from 0 to 0FFFEH.
data	is any word (16 bits) of data entered in the current base. Using this parameter writes the data to the specified I/O port.

Discussion

Use the WPORT command to display a value from a word-wide port or write a value to a word-wide port. You can access up to 32K word-wide I/O ports.

The I/O port must be mapped to USER; the emulator displays an error message if the *i/o-address* is mapped to ICE.

Note that Intel Corporation reserves port locations 0F8H-0FFH for coprocessor communication.

The FASTBREAK tool variable controls port access during emulation. Setting FASTBREAK to TRUE allows you to make asynchronous port accesses while the emulator is executing your program. Setting FASTBREAK to FALSE prevents port accesses during emulation.

The WPORT command is disabled if there are any outstanding mapped-I/O port accesses.

Example

The following example displays and changes the contents of the word-wide I/O port located at address 10H.

```
hlt> WPORT 10H
0010H  ffffH
hlt> WPORT 10H = 01a0H
```

Cross-References

<expr>
FASTBREAK
IORDY
MAPIO
PORT

Appendix A
Limitations

This section lists the behavioral differences between the ICE™-186 In-Circuit Emulator and an 80C186/80C188 microprocessor. Keep these differences in mind when using the emulator to execute program code on your prototype system.

Power On	An 80C186/80C188 microprocessor powers up running, after /RES has been released. The emulator powers up halted. It does not begin the execution of program code until a GO command is entered.
NMI Latency	The NMI signal from the prototype may be delayed by up to two CLKOUT cycles before being applied to the processor.

Absolute address	An absolute address is a physical address. For load files, all address references contain physical addresses; no references to locations are determined at load time.
Activity Monitor	The Activity Monitor in the emulator consists of the emulator event recognizers, trace facilities, and associated control signals. The Activity Monitor can be started and stopped separately from emulation. (See the GO command entry in Chapter 6.)
Address space	The 80C186 or 80C188 microprocessor can address 1M byte of physical memory.
Addresses	An address is an unsigned value that corresponds to a location in memory that is addressable by the processor. Addresses can be physical, segmented, or symbolic variables that evaluate to addresses. They can also include base, index, and displacement addressing components.
Code segment	Code segments contain the instructions and immediate data for a program.
Code segment (CS) register	The CS register contains the upper 16 bits of the start address of the current code segment.
Data segment	Data segments contain the data (other than immediate data) for the program.
Data segment (DS) register	The DS register contains the upper 16 bits of the start address of the data segment.
Data type	A data type is a keyword that is used to describe the attributes of a data variable. Examples of data types include BYTE (ORD1), WORD (ORD2), INT4, and BOOL1. For a complete list of data types, refer to the <Data type> entry in Chapter 6.

Emulator controller unit (ECU)	The ECU is the emulator hardware module that is physically connected to the IBM PC AT host. It supports the following functions: Activity Monitor and word recognition, real-time trace, memory mapping, and I/O mapping.
Event	Event is an element within the GO command syntax. It specifies the condition upon which an action (e.g., trace collection or emulation break) is to occur. Typical events include instruction executions or memory accesses.
Expression	An expression is a series of operands or operators that yields a numeric (e.g., 1), Boolean (e.g., FALSE), or string (e.g., string) value. See the <Expr> entry in Chapter 6 for more information.
Extra segment	An extra segment is simply a segment that the processor can use as an instruction or an immediate data source or destination.
Extra segment (ES) register	The ES register contains the upper 16 bits of the start address of the current extra segment.
History buffer	The history buffer maintains a record of command entries. You can recall these command entries with the up-arrow key.
ICE-mapped I/O	See the Mapped I/O entry in this glossary.
ICE-mapped memory	See the Mapped memory entry in this glossary.
Keyword	A keyword is a command or command parameter that has reserved definition within the emulator command language.

Mapped I/O	Mapped I/O refers to the port addresses accessed during emulation. Port addresses can be mapped to the prototype (referred to as USER-mapped I/O ports) and/or the emulator (referred to as ICE-mapped I/O ports). See the MAPIO command entry in Chapter 6 for more information.
Mapped memory	Mapped memory refers to the memory address partitions that are accessed by the program during emulation. Memory address partitions can be mapped to the prototype (referred to as USER-mapped memory) and/or the emulator (referred to as ICE-mapped memory). See the MAP command entry in Chapter 6 for more information.
OMF	OMF is an abbreviation for Object Module Format, which is the Intel standard for the structure of object code files.
Peripheral Control Block	The 80C186 integrated peripherals are controlled by an array of 16-bit registers located in an internal 256-byte control block. Control and status registers are provided for the chip-select unit, the DMA controller, the timers, and the interrupt controller. The control block may be mapped into memory or I/O space. The control block base address is programmed by the 16-bit relocation register, which is contained in the control block itself. Each of the control and status registers are located at a fixed offset from the base address.
Physical address	Physical addresses refer to addresses seen on the processor bus. They reference a specific hardware location in memory.
Physical memory	Physical memory is composed of the random-access memory (RAM) and read-only memory (ROM) that is part of the system hardware.
Pipelining	Instruction pipelining enables the processor to fetch an instruction while still processing the previous instruction. This technique maximizes bus efficiency.

Port	The 80C186 microprocessor can address a maximum of 64K bytes of I/O ports.
Real memory	See the Physical memory entry in this glossary.
Segment	A segment is a variable-length section of contiguous physical memory, not exceeding 64K bytes.
Stack segment (SS) register	The SS register contains the upper 16 bits of the start address of the current stack.
Tool	The DEFINE command uses the TOOL parameter to determine that the emulator is the device to be activated. To the command language, "tool" is synonymous with "emulator". (See the DEFINE entry in Chapter 6.)
Tool variables	Tool variables set up and control the emulator hardware and the processor. The control variables include ALEMODE, BTHRDY, BUSACT, ENI, FASTBREAK, IORDY, IPATINT, MEMRDY, OSYNC, QSTAT, QUERY, RSTEN, and VERIFY. (Refer to the appropriate entries in Chapter 6 for more information.)
USER-mapped I/O	See the Mapped I/O entry in this glossary.
USER-mapped memory	See the Mapped memory entry in this glossary.

Wild identifier	A wild identifier is an identifier that supports the use of * and ? for wild cards. The * can be used to represent any number of characters, and the ? can be used to represent a single character. For an example of how to use a wild card, refer to Chapter 3, Section 3.5, Example 1.
Word recognizer	Word recognizers are part of the emulator circuitry that is controlled by the Activity Monitor. This circuitry detects event occurrences and signals the Activity Monitor so that appropriate action can be taken.

T

International Sales Offices

BELGIUM
Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK
Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND
Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND
Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE
Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL
Intel Semiconductors LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY
Intel Corporation S.P.A.
Milandfiori, Palazzo E/4
20090 Assago (Milano)

JAPAN
Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

NETHERLANDS
Intel Semiconductor (Nederland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN
Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvaegen 24
S-171 36 Solna

SWITZERLAND
Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY
Intel Semiconductor GmbH
Seidlestrasse 27
D-8000 Muenchen 2

SERVICE INFORMATION

For service or assistance with Intel products, call:

- **1-800-INTEL-4-U** (1-800-468-3548) — in the United States and Canada
- Your local Intel sales office — in Europe or Japan
- Your Intel distributor — in any other area

Intel stands behind its products with a world-wide service and support organization. If you have problems, need assistance, or have a question, Intel can provide:

- On-site or carry-in service for hardware products
- Phone support for all Intel products
- On-site consulting for designing with Intel products or using Intel products in your designs
- Customer training workshops
- Updates to software products

To get more information on these services or to take advantage of them, call the INTEL-4-U number above.

All Intel products have a minimum warranty of 90 days, and all warranties include one or more of the services listed above. Talk with your Intel salesperson or call the INTEL-4-U number to determine the warranty services available for this product and how to register for them.

Intel is committed to continuing service for all its products.

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95052-8130 (408) **987-8080**

Printed in U.S.A.

DEVELOPMENT TOOLS AND SOFTWARE

4810/2.3K/0788/PT/TS/DTO