intel®

THE I²ICE™ INTEGRATED INSTRUMENTATION AND IN-CIRCUIT EMULATION SYSTEM USER'S GUIDE

Intel Corp., 3065 Bowers Ave., Santa Clara, CA 95051

Order Number: 166298-001

THE I²ICE™ INTEGRATED INSTRUMENTATION AND IN-CIRCUIT EMULATION SYSTEM USER'S GUIDE

Order Number: 166298-001

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.

Additional copies of this manual or other Intel literature may be obtained from:

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

The information in this document is subject to change without notice.

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, or as defined in ASPR 7-104.9(a)(9).

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 makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel 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:

Above	iLBX	iPDS	ONCE
BITBUS	i_m	iPSC	OpenNET
COMMputer	iMDDX	iRMX	Plug-A-Bubble
CREDIT	iMMX	iSBC	PROMPT
Data Pipeline	Insite	iSBX	Promware
GENIUS	Intel	iSDM	QueX
$\overline{\text{i}}$	int$_e$l	iSXM	QUEST
i	int$_e$lBOS	Library Manager	Ripplemode
I^2ICE	Intelevision	MCS	RMX/80
ICE	int$_e$ligent Identifier	Megachassis	RUPI
iCEL	int$_e$ligent Programming	MICROMAINFRAME	Seamless
iCS	Intellec	MULTIBUS	SLD
iDBP	Intellink	MULTICHANNEL	UPI
iDIS	iOSP	MULTIMODULE	VLSiCEL

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.

* MULTIBUS is a patented Intel bus.

CONTENTS

CHAPTER 2 GUIDE TO THE I²ICE™ SYSTEM TUTORIAL

Page

CHAPTER 3 INTRODUCTION TO USING THE I²ICE™ SYSTEM

CHAPTER 4 THE I²ICE™ SYSTEM PERSONALITY MODULES (PROBES)

TABLES

FIGURES

FIGURES (continued) Page

PREFACE

This manual introduces Intel's Integrated Instrumentation and In-Circuit Emulation (I²ICE™) system. It assumes that you are familiar with the architecture of the iAPX 86, iAPX 88, iAPX 186, iAPX 188, and iAPX 286 microprocessors. It also assumes that you are familiar with the concept of in-circuit emulation.

Chapter 1	describes the hardware/software design process and explains how the I²ICE system aids this design process. Chapter 1 also provides general information about the features and components of the I²ICE system, describes the hardware components of the I²ICE system, and describes the I²ICE system software packages.
Chapter 2	provides an overview of the I²ICE system on-line tutorial and contains a listing of the program debugged during the tutorial session.
Chapter 3	introduces the user to the I²ICE system. This chapter reinforces and augments information provided in the on-line tutorial. It explains symbolic debugging, I²ICE debug procedures, file handling, the I²ICE editor, and the single-line assembler. The chapter also contains a sample program that illustrates how to get the I²ICE system up and running; it describes how to set up and work with the I²ICE memory and I/O maps; and it describes how to set breakpoints and interpret the trace buffer.
Chapter 4	is an overview of how the I²ICE system operates with the 8086/8088, 80186/80188, and 80286 personality modules (probes). This chapter also describes special considerations that pertain to each of the probes.
Chapter 5	describes the special debugging aids offered by the I²ICE system for prototypes that use coprocessors and processor extensions.
Chapter 6	describes the operation of I²ICE systems with more than one probe.
Appendix A	explains how to install I²ICE system hardware, except hardware that is installed on the host system.
Appendix B	shows how to configure the I²ICE system to run with non-standard terminals.
Host Installation Appendixes	explain how to install the I²ICE system hardware in the host system and how to install the I²ICE system software.
Glossary	defines specific I²ICE system terms used in this manual.
Index	

Related Publications

The following manuals contain additional information about the I²ICE system and its operating environment.

Copies of the publications listed are available through the Intel Literature Department.

I²ICE™ System Publications Library

The following manuals are supplied (together with this manual) with the I²ICE system; they contain additional information about the I²ICE system and its operating environment. Copies of the publications listed are also available through the Intel Literature Department, located at the following address:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051
(800) 548-4725

- *I²ICE™ Integrated Instrumentation and In-Circuit Emulation System* (data sheet), order number 210469.
  This publication provides an overview of the I²ICE system. It describes the hardware and software, provides some general application information, and lists the system specifications. The data sheet is available through Intel sales offices as well as the Intel Literature Department.
- *I²ICE™ Reference Manual*, order number 166302.
  This manual is the principle reference for the I²ICE command language. It includes descriptions of I²ICE commands and I²ICE topics that are not probe-specific.
- *PSCOPE-86 High-Level Program Debugger User's Guide*, order number 121790
  [This manual is only supplied to users with Intel hosts.] The manual describes the operation of PSCOPE-86, a high-level language symbolic debugger. It includes a tutorial and invocation sections, a command dictionary, an error message listing, and configuration information for non-Intel terminals.
- *AEDIT Text Editor User's Guide*, order number 121756
  This manual describes features and operation of the screen editor that is part of the I²ICE system software.

Reference Publications

The publications in the following sections are not supplied with the I²ICE system.

Hardware Reference Publications

- *Memory Components Handbook*, order number 210830.
  This catalog contains data sheets on the memory components manufactured by Intel Corporation.
- *Microsystems Components Handbook*, order number 230843 (two volumes).
  These handbooks contain data sheets on the microprocessor and peripheral products manufactured by Intel Corporation.
- *OEM Systems Handbook*, order number 210941.
  This catalog contains data sheets on integrated microcomputer systems, single-board computers, memory expansion boards, high-speed math boards (including the iSBC® 337 MULTIMODULE™ board), peripheral controllers, communications controllers, digital I/O expansion and signal conditioning boards, industrial control series, and analog I/O expansion. It also contains data sheets on systems software, such as the iRMX™ operating system.
- *Development Systems Handbook*, order number 210940.
  This catalog contains data sheets on microcomputer development systems (hardware and software), in-circuit emulators (including the I²ICE system), network development systems, system design kits, and third-party software.
- *iAPX 86/88, 186/188 User's Manual*, order number 210911.
  This manual contains product descriptions and operating instructions for 8-bit and 16-bit CPUs and support chips in the iAPX 86/88 and 80186/80188 families.
- *iAPX 186 High Integration 16-Bit Microprocessor* (data sheet), order number 210451.
  This publication provides an overview of the iAPX 186 microprocessor. It includes chip pinouts, a functional description, hardware, software, and timing specifications, and an instruction set summary.
- *iAPX 286/10 High Performance Microprocessor with Memory Management and Protection* (data sheet), order number 210253.
  This publication provides an overview of the iAPX 286/10 microprocessor. It includes chip pinouts, a functional description, chip architecture, hardware and timing specifications, and an instruction set summary.
- *iAPX 286 Hardware Reference Manual*, order number 210760.
  This manual is the hardware reference guide for iAPX 286 system designs.
- *iSBC® 337 MULTIMODULE™ Numeric Data Processor Hardware Reference Manual*, order number 142887.
  This manual contains design and operation information on the iSBC 337 NDP MULTIMODULE board. The manual includes installation, program interface, operation, and service information.

Software Reference Publications

- *ASM-86 Language Reference Manual*, order number 121703.
  This manual provides design and operating information about the ASM-86 assembly

language. The manual provides an overview of the language, procedures for program structuring, information about data operation, and an appendix describing the 80186 instruction set.

- *iAPX 86/88 Family Utilities User's Guide*, order number 121616.
  This guide provides a full description of the utility commands that support 86/88 program development. It is intended for use with any language translator that generates object code compatible with the utility commands.
- *8087 Support Library Reference Manual*, order number 121725.
  This manual provides design and operating information about the library of support utilities for the 8087 processor. It provides overviews of the support library, procedures for program structuring, and information on data operation.
- *iAPX 286 Programmer's Reference Manual*, order number 210498.
  This manual describes the iAPX 286 architecture and instruction set.
- *iAPX 286 Utilities User's Guide*, order number 121934.
  This guide provides a full description of the utility commands that support 286 program development. It is intended for use with any language translator that generates object code compatible with the utility commands.
- *iAPX 286 Operating Systems Writer's Guide*, order number 121960.
  This book is written for systems designers, operating system designers, and programmers using the Intel iAPX 286 microprocessor in its protected, virtual-address mode.
- *PL/M-86 Programming Manual*, order number 980466.
  This manual provides programming instructions for PL/M-86. It includes details on expressions and assignments, procedures, variables, and a sample program.
- *PL/M-86 User's Guide*, order number 121636.
- *PL/M-286 User's Guide*, order number 121945.
- *Pascal-86 User's Guide*, order number 121539.
- *FORTRAN-86 User's Guide*, order number 121570.
- *iC-86 Compiler User's Guide*, order number 122085.
  These guides provide introductory and overview information on the high-level languages supported by the I²ICE system. Each manual provides installation instructions, language information, compiler operating instructions, and information on interfacing to other software modules.

Command Syntax

The following syntax notation is used throughout the I²ICE manual set:

COMMANDS — Command keywords appear in all uppercase letters. You must enter commands exactly as they appear, except that you may enter them in either uppercase or lowercase letters.

elements	Items for which you must substitute a value, expression, file name, etc., are in lowercase letters and italicized.
{menu}	Braces indicate that you must select one and only one of the items in the enclosed menu.
{menu}*	Braces followed by an asterisk (*) indicate that you must select at least one of the items in the enclosed menu
[menu]	Brackets indicate optional items of which you can select one and only one.
[menu]*	Brackets followed by an asterisk (*) indicate optional items of which you can select more than one item.
punctuation	You must enter punctuation other than braces ({ }) and brackets ([ ]) exactly as shown. For example, you must enter all the punctuation shown in the following command: `LIST :F1:myprog.001`
apostrophe	If your terminal has two apostrophes (or single quotes), determine which one the I²ICE system accepts in command syntax.
CTRL	CTRL denotes the terminal's control key. For example, CTRL-C means enter C while pressing the control key.

REV.	REVISION HISTORY	DATE
-001	Original Issue.	9/85

SERVICE INFORMATION

The best possible service for your Intel product is provided by Intel Customer Engineers. These trained professionals provide prompt, efficient, on-site installation, preventive maintenance, and corrective maintenance services required to keep your equipment in the best possible operating condition.

Intel Customer Engineers provides the service needed through a prepaid service contract or on an hourly charge basis. For further information, contact your local Intel sales office.

In Phoenix, Arizona, there is a technical information center that will connect you with the software support group for your particular Intel product.

Telephone (602) 869-INFO (4636)

When an Intel Customer Engineer is not available, contact the Intel Product Service Center.

United States customers can obtain service and repair assistance from Intel Corporation by contacting the Intel Product Service Center in their local area. Customers outside the United States should contact their sales source (Intel Sales Office or Authorized Distributor) for service information and repair assistance.

Before calling the Product Service Center, have the following information available:

1. The date you received the product.
2. The complete part number of the product (including dash number). On boards, this number is usually silk-screened onto the board. On other MCSD products, it is usually stamped on a label.
3. The serial number of the product. On boards, this number is usually stamped on the board. On other MCSD products, the serial number is usually stamped on a label mounted on the outside of the chassis.
4. The shipping and billing address.
5. If the Intel Product warranty has expired, a purchase order number is needed for billing purposes.
6. Be sure to advise the Center personnel of any extended warranty agreements that apply.

Use the following telephone numbers for contacting the Intel Product Service Center:

Western Region:	(602) 869-4951
Midwest Region:	(602) 869-4392
Eastern Region:	(602) 869-4045
International:	(602) 869-4862

Always contact the Product Service Center before returning a product to Intel for repair. You are given a repair authorization number, shipping instructions, and other important

information which helps Intel provide you with fast, efficient service. If you are returning the product because of damage sustained during shipment, or if the product is out of warranty, a purchase order is required before Intel can initiate the repair.

If available, use the original factory packaging material when preparing a product for shipment to the Intel Product Service Center. If the original packaging material is not available, wrap the product in a cushioning material such as Air Cap SD-240, manufactured by the Sealed Air Corporation, Hawthorne, N.J. Securely enclose it in a heavy-duty corrugated shipping carton, mark it "FRAGILE" to ensure careful handling, and ship it to the address specified by the Intel Product Service Center.

1 I²ICE™ SYSTEM OVERVIEW

Intel's Integrated Instrumentation and In-Circuit Emulation (I²ICE™) system offers real-time hardware and software emulation for designs using the iAPX 86, iAPX 88, iAPX 186, iAPX 188, and iAPX 286 microprocessor systems.

This chapter is an overview of the I²ICE system and its operating environment. It contains the following sections.

- The Microcomputer Development Process—this section reviews the role of an emulator in the microcomputer development process.
- An Introduction to the I²ICE™ System—this section describes the I²ICE system configuration, options, and accessories.
- Hardware Overview
- Software Overview
- The I²ICE™ System Command Language
- I²ICE™ System Specifications

The Microcomputer Development Process

Designing a product that contains a microcomputer requires close coordination of two separate but highly dependent design efforts: hardware development and software development. These two development efforts can be accomplished independently, but it is more efficient to work on them together. Figure 1-1 illustrates a typical microcomputer development process, using a development system and an emulator.

Features of a Host Development System with an Emulator

A host development system with an emulator offers the following resources:

- A stand-alone computer
- Development software such as assemblers and compilers
- Prototype hardware interface
- Mapping capability
- Break and trace capability

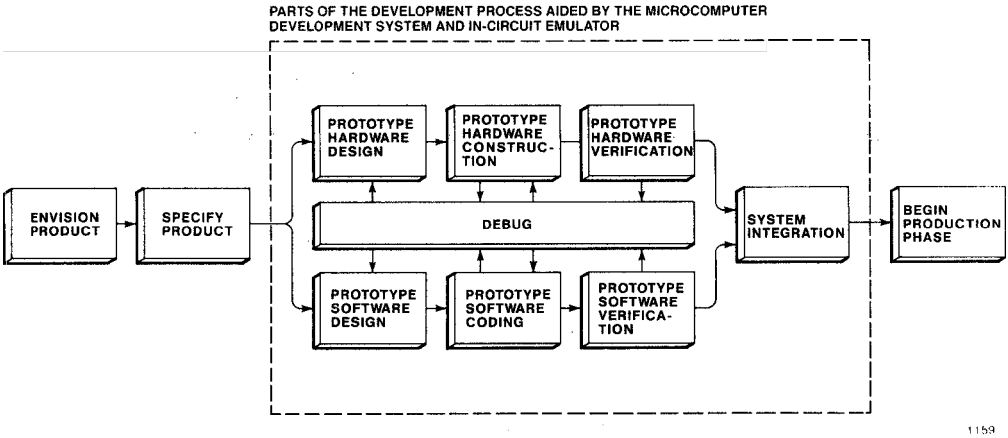


Figure 1-1 Typical Microcomputer Development Process

Generalized Hardware Design Steps

Although the complexity of hardware design varies from one design to another, the general process is the same. The following sequence illustrates the advantage of using a development system with an emulator.

- Organize the hardware into logical blocks with well-defined inputs and outputs. Determine the CPU, RAM, ROM, I/O, board layout, and bus interface requirements.
- Build a hardware prototype.
- Test the interaction of the prototype hardware with proven software. The user program resides initially in either the host development system's memory or the emulator's memory. You can reassign the user program, memory block by memory block, to prototype memory as you verify the code and as prototype memory becomes available.
- Test, debug, and verify each prototype module.

Generalized Software Design Steps

Software design follows a process similar to hardware design as illustrated in the following sequence.

- Organize the software into logical blocks with well-defined inputs and outputs. Complete the specifications for the software control logic and integrated system performance.

- Program the software modules. Desk-check each module as it is completed. Name and store the software modules as files in the development system's memory. Assemble or compile the modules. Link and load the combinations that are ready to be tested.
- Emulate the program using an emulator to debug the software.

Hardware/Software Integration

When the hardware and software designs are complete, system integration is already in progress. The usefulness of a development system with an emulator extends into the integration and test phases.

When an emulator is connected to the prototype system through the microprocessor socket and emulator hardware probes, the emulator can emulate, test, and trace prototype system operation.

After testing the prototype, the host development system with an emulator can be used to verify the product in production test. Test procedures developed for final prototype testing can serve as the basis for production test routines. The same procedures developed for hardware debugging and production test can also be used to troubleshoot and repair failing products at a repair center.

An Introduction to the I²ICE™ System

Intel developed the I²ICE system to address the requirements of designers who use Intel's iAPX microprocessors. The I²ICE system is a second-generation design tool that provides the following advantages:

- Full-speed, multiple microprocessor emulation
- Real-time emulation support for each of the iAPX microprocessors (86/88, 186/188, and 286)
- Software patching without recompiling or reassembling
- Extensive breakpoint capabilities
- Expanded logic analysis

The I²ICE system also offers the following features:

- Access to eight signals input from the prototype system and two signals output to the prototype system using the emulation clips. An additional output line is asserted when the I²ICE system breaks emulation, and another is asserted when the I²ICE system triggers a trace.
- An optional Intel Logic Timing Analyzer (iLTA) module which is a general-purpose, integrated, 16-channel, 100-MHz logic timing analyzer. With the mass storage provided by the host development system, you can store an unlimited number of scope displays for later analysis. Combined with the capabilities of the I²ICE system, the iLTA extends the range of development applications. [This option is not available for IBM PC hosts.]

- An optional high-speed (OHS) memory board. The I²ICE system provides up to 256K bytes of additional programmable wait-state RAM for each emulator. With the I²ICE system, you can specify zero wait-states for real-time emulation and simulate slow memories by inserting up to 15 wait states.
- Program memory mapping to high-speed (HS) memory (up to 32K-bytes) on the system map-I/O board, to OHS memory, or to host development system memory. [There is no mapping to IBM PC/AT or PC/XT host memory.]
- Multiprocessor debugging. The I²ICE system can control up to four emulators simultaneously.
- Coprocessor support, which provides debugging support for the 8087 numeric coprocessor and the 80287 numeric processor extension.
- Emulation flexibility. The iAPX 86/88 and 186/188 emulation personality modules (also called probes) each emulates two separate microprocessors. To change microprocessors, you need only change the personality module CPU chip and jumpers on the buffer and personality boards.

NOTE

Probe CPU chips must be provided by Intel. All probes use either bond-out chips or specially tested microprocessors.

- Symbolic debugging support for programs written in assembly language, PL/M, C, Pascal, and FORTRAN by both PSCOPE-86 and the I²ICE system command language. With the symbolic debugging capabilities you can access variables and memory locations with user-defined names.
- Two programmable event machines that allow break and trace on simple and complex event sequences.
- A real-time trace buffer that displays trace information in either a disassembled-instruction format or a microprocessor bus-cycle format.

The Base Configuration of the I²ICE™ System

Figure 1-2 shows a basic single-chassis I²ICE system. The base configuration is readily expandable to include a number of I²ICE system options.

The base configuration of the I²ICE system contains the following hardware:

- The host interface board, which resides in the host development system and handles communication between the host development system and the I²ICE instrumentation chassis.
- The I²ICE instrumentation chassis, which contains the communications board and provides slots for up to four I²ICE system boards. The communications board connects to the host interface board. The chassis slots also hold the map-I/O board and the break/trace board.
- The map-I/O board, which contains high-speed RAM and the memory map. There are 32K bytes of high-speed RAM available to user programs through the memory map. The

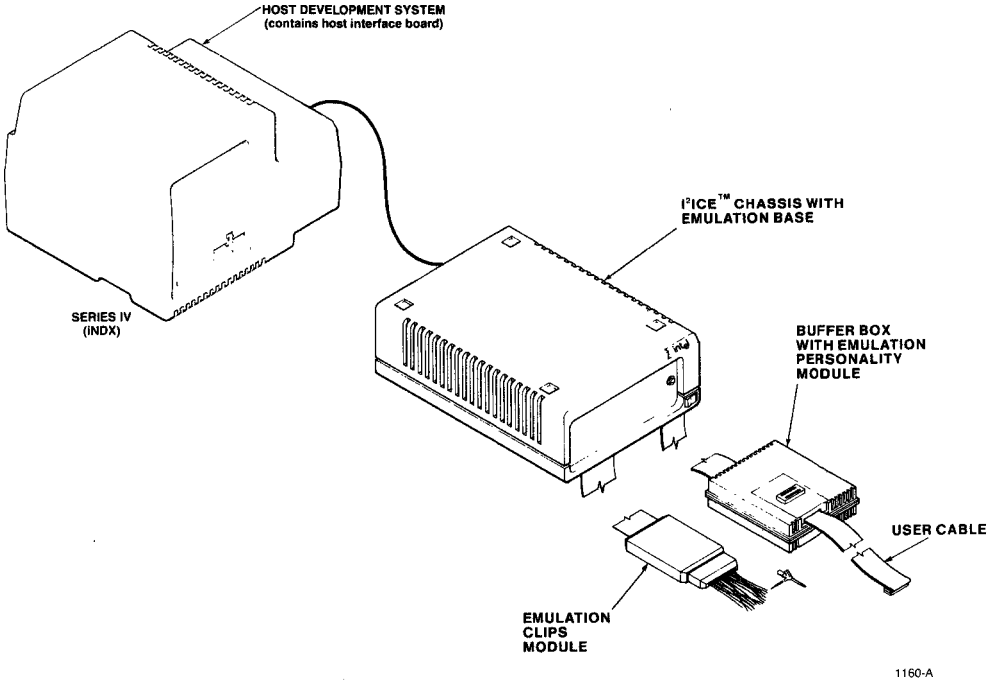


Figure 1-2 A Basic I²ICE™ System

rest of the on-board RAM contains I²ICE system software that implements probe-specific commands.

- The break/trace board, which contains two programmable event machines that implement the break and trace specifications.
- The emulation clips assembly, which enables the I²ICE system to assert signals to the prototype hardware and to read signals from the prototype hardware.
- The probe buffer box, which contains the buffer board and the personality board. The buffer board connects to the break/trace board and the map-I/O board in the instrumentation chassis. The personality board connects to the prototype hardware through the user cable.
- System cables, which connect the instrumentation chassis to the host development system and to the probe buffer box. The host/instrumentation-chassis cable is either 10 or 40 feet long.

The base configuration of the I²ICE system contains the following software.

- The I²ICE system host software, which implements the non-probe-specific I²ICE system commands. After the I²ICE system software is loaded, it resides in the host development system.
- The I²ICE system probe (personality module) software, which implements the probe-specific I²ICE system commands. After the probe software is loaded, it resides in the map-I/O board.
- The I²ICE system diagnostic software.
- The PSCOPE-86 high-level language software debugger which runs in the host development system and does not emulate in real-time. It is loaded in place of the I²ICE system host software. [PSCOPE-86 is an option for the IBM PC hosts.]
- The I²ICE system tutorial software.

I²ICE™ System Options

The I²ICE system is expandable. An I²ICE system can have up to four instrumentation chassis. Each chassis can emulate any of the five microprocessors. Each chassis provides up to 256K bytes of additional zero wait-state memory to user programs, and serves as a logic timing analyzer while retaining the basic I²ICE system functions.

Figure 1-3 shows the maximum configuration of an I²ICE system.

The I²ICE system options are as follows.

- The optional high-speed (OHS) memory board. This option increases the amount of I²ICE system memory available to user programs through the memory map. Each board adds 128K bytes of zero wait-state memory.
- The iLTA has all the features found in a stand-alone logic analyzer. In addition, the iLTA uses the mass storage and real-time breakpoint facilities of the I²ICE system. The iLTA monitors 16 channels at a maximum frequency of 100 MHz. [The iLTA system is not available for use with the IBM PC hosts.]

  The iLTA hardware consists of one board, two probe pods, and test and hook-up accessories. Each instrumentation chassis can contain only one iLTA module, and the iLTA board must reside in the top slot.

  The I²ICE system enables real-time emulation and debugging of up to four instrumentation chassis from a single host development system. A unit is composed of an instrumentation chassis, a probe buffer box, and an emulation clips pod. Each unit can emulate one of the five iAPX microprocessors. Optional high-speed memory residing in one chassis cannot be mapped into the emulation environment of another chassis.

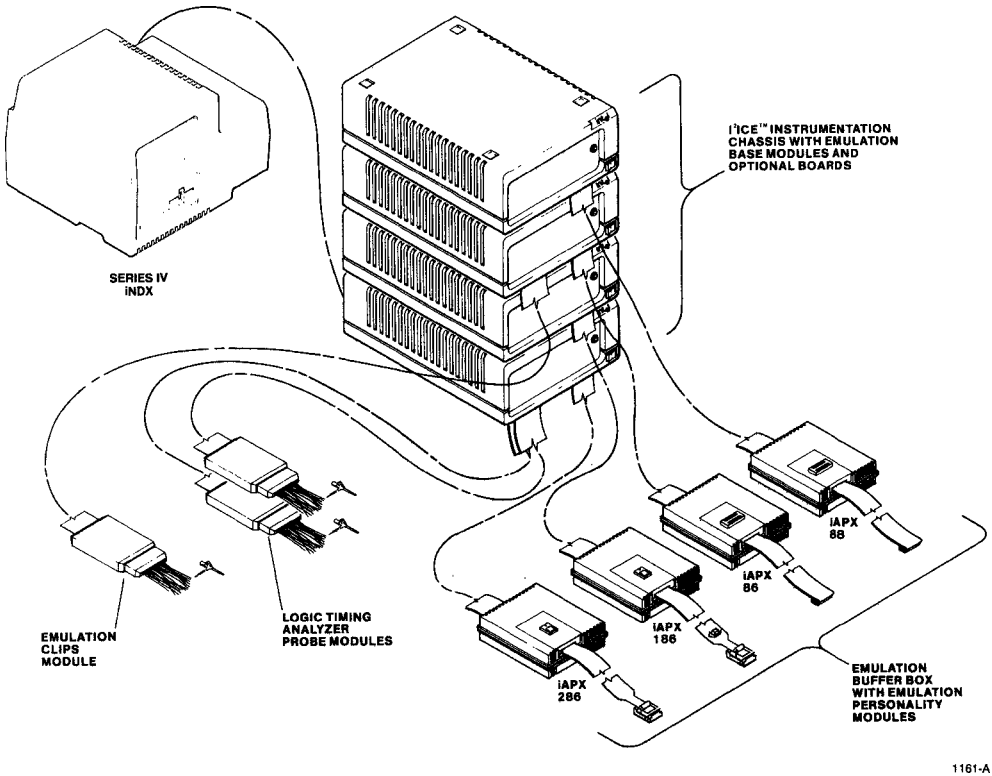


Figure 1-3 A Maximum Configuration I²ICE™ System

I²ICE™ System Accessories

The I²ICE system accessories are as follows.

- An emulation clips assembly that consists of an emulation clips pod, an emulation clips cable, and an emulation clips terminator.
- Two logic probe pods (channels 0-7 and 8-F) that supplement the iLTA pods. Each pod can be ordered separately.
- Two iLTA terminator sets that supplement the terminator sets normally supplied with the iLTA. A 16-channel terminator set is used to hook-up to 16 separate signals. An 8-channel

terminator set is used for glitch detection and multi-threshold triggering. Each terminator set can be ordered separately.

- A microhook set that consists of 40 grabber clips (microhooks) that connect to wires belonging to the emulation clips terminator or to the iLTA terminators. The microhooks connect to individual leads of a dual in-line package (DIP). This set supplements the microhooks supplied with the emulation clips pod and the iLTA.

Hardware Overview

The hardware components of the PICE system are listed in Table 1-1 and discussed in the following sections.

The Instrumentation Chassis

The instrumentation chassis encloses a card cage and a backplane with four vacant slots, a switching power supply, and a communications board. As many as four chassis may be daisy-chained together in a fully-configured PICE system to allow multiprocessor emulation under the control of a single host development system.

Both the top and front covers of the instrumentation chassis are easily removed to allow access to system components. The four-slot card cage holds 12 in. by 12 in. (.30 m by .30 m) boards. Three fans on the side of the card cage provide cooling.

The internal switching power supply uses line power (110 or 220 VAC) to develop the regulated DC voltages used by the PICE system. (The host interface board obtains power from the host development system.) The power supply has five voltages: +5VDC (two voltages), −5.2VDC, +15VDC, and −15VDC.

The communications board is part of the rear panel of the instrumentation chassis. The board contains termination switches, rear panel connectors, and interface circuitry. The communications board handles communications with the host development system (using the host interface board). The communications board also provides the link between the PICE chassis in which it resides and other PICE chassis in the system. In multiple-chassis PICE systems, the communications board assigns sequential unit numbers (0 through 3) to each chassis.

The Host Interface Board

Each PICE system requires one host interface board. It resides in the host development system and controls up to four instrumentation chassis. For Intel hosts, the host interface board is a MULTIBUS® master board that makes possible direct memory access (DMA) to MULTIBUS board memory by the PICE system; for multiple-probe systems, the probes can share common MULTIBUS memory. IBM PC/AT and PC/XT hosts are provided with a PC-specific interface board that does not support MULTIBUS mapping.

Table 1-1 I²ICE™ System Hardware Components

- Instrumentation chassis
- Host interface board
- Emulation base module
  - Break/trace board
  - Map-I/O board
  - Buffer base assembly
    - Emulation buffer board
    - Probe buffer box base
    - Probe buffer box cable
  - Emulation clips assembly
    - Emulation clips pod
    - Emulation clips cable
    - Emulation clips terminator
    - Emulation clips microhooks
- System cables
  - Host/chassis cable (10 or 40 feet)
  - Internal host cable (not needed for Model 800 and IBM PC hosts)
  - Inter-chassis cable set (2 or 10 feet)
    - Inter-chassis communications cable
    - Inter-chassis break cable
- High-speed memory module
- Optional high-speed memory board
- iLTA module (not available for IBM PC hosts)
  - iLTA board
  - External trigger wires
  - iLTA demonstration card assembly
  - Logic probe assembly
    - Logic probe pod, channels 0-7
    - Logic probe pod, channels 8-F
    - Terminator set, 16-channel
    - Terminator set, 8-channel
    - Microhook set (40 microhooks)
- 86/88 emulation personality module
  - Probe buffer cover assembly
    - Buffer box cover
    - 86/88 personality board
    - Selected 8086 CPU chip
    - Selected 8088 CPU chip
    - User system cable
- 186/188 emulation personality module
  - Probe buffer cover assembly
    - Buffer box cover
    - 186/188 personality board
    - Selected 80186 CPU chip
    - Selected 80188 CPU chip
    - User system cable
- 286 emulation personality module
  - Probe buffer cover assembly
    - Buffer box cover
    - 286 personality board and 286 self-test board
    - Selected 80286 CPU chip
    - User system cable

The Emulation Base Module

The emulation base module provides a generic environment that an emulation personality module tailors to the microprocessor being emulated. Each instrumentation chassis contains one emulation base module. A fully-expanded PICE system contains four instrumentation chassis and four emulation base modules, each with an emulation personality module. The emulation base module consists of the following hardware:

- A map-I/O map board
- A break/trace board
- A buffer base assembly
- An emulation clips assembly

The map-I/O board contains the PICE memory map and memory. The PICE memory consists of 32K bytes of user-accessible, zero wait-state RAM. The PICE system uses 128K bytes of RAM to store the code for probe-specific commands. The memory map redirects an emulated microprocessor's memory and I/O port address space to combinations of memory and I/O belonging to the host development system, the prototype, and the PICE system.

The break/trace board uses two programmable event machines (one execution event machine and one system event machine) to trigger break and trace points. The event machines recognize complex, multilevel trigger event sequences. The break/trace board also contains a 1023 by 48-bit trace data buffer.

The buffer base assembly provides buffers, latches, multiplexers, and a wait-state generator for the emulation base module. The buffer base assembly is incomplete without the appropriate emulation personality module (probe). The buffer base assembly consists of the following hardware:

- The probe buffer box base
- The probe buffer box cable
- The emulation buffer board

The emulation buffer board resides in the probe buffer box base. The emulation buffer board contains the wait-state generator and the fan control circuit for the buffer box. With the wait-state generator you can simulate slow memories by inserting up to 15 wait-states into memory accesses. The probe buffer box cable connects the emulation buffer board to the break/trace board and the map-I/O board in the instrumentation chassis.

The emulation clips assembly consists of the following hardware:

- The emulation clips cable
- The emulation clips pod
- The emulation clips terminator
- The emulation clips microhooks

The emulation clips cable assembly attaches to the break/trace board in the instrumentation chassis. This assembly enables the monitoring of eight TTL-compatible input signals and the generation of two TTL-compatible output signals during emulation. The emulation clips assembly also provides system break and system trace outputs that aid multi-microprocessor debugging by enabling the I²ICE system to communicate with other I²ICE systems.

System Interface Cables

A single host/chassis interface cable connects the I²ICE system to the host interface board. Each additional instrumentation chassis in the I²ICE system is linked to the previous instrumentation chassis with two inter-chassis interface cables (an inter-chassis communication cable and an inter-chassis break cable). All chassis-to-chassis connection, termination, and chassis addressing is performed on the communications board of each instrumentation chassis.

High-Speed Memory

The high-speed (HS) memory consists of 32K bytes of programmable wait-state memory and resides on the map-I/O board.

Optional High-Speed Memory Board

The optional high-speed (OHS) memory board supplements the 32K bytes of high-speed I²ICE memory. The OHS memory module is a 12 in. by 12 in. (.30 m by .30 m) board with static RAM memory components. Each OHS memory board contains 128K bytes of zero wait-state memory. Each I²ICE system can contain two OHS memory boards for a total of 256K bytes of OHS memory.

The Intel Logic Timing Analyzer (iLTA)

The iLTA is a test and measurement module which combines all the features of a stand-alone logic analyzer with the event machines and storage capabilities of the I²ICE system. The iLTA consists of the logic probe assembly, the external trigger wires, the iLTA demonstration card assembly, and the logic probe assembly. [The iLTA system is not available for IBM PC hosts.]

Emulation Personality Modules

An emulation personality module (also called an emulation probe) configures the I²ICE emulation base module to support a specific microprocessor. Each emulation personality module consists of the following hardware:

- The personality board
- The probe buffer box cover
- The user cable with probe
- The selected CPU chip

In addition to supporting real-time emulation and debugging for the specified microprocessor, the emulation personality modules also provide debugging support for coprocessors. For example, the I²ICE system 8086/8088 and 80186/80188 emulation personality modules provide debugging support for the 8087 coprocessor. The 80286 emulation personality module provides debugging support for the 80287 numeric processor extension.

Software Overview

The I²ICE system software is a versatile and powerful debugging language. The usefulness of an I²ICE system extends throughout the development cycle, beginning with the symbolic debugging of prototype software and ending with the final integration of debugged software and prototype hardware (see Figure 1-4). Typical I²ICE system functions include setting breakpoints, controlling trace collection, writing debug procedures, and changing program variables. The I²ICE system software consists of the following:

- The I²ICE system host software, which resides in the host development system. It implements the I²ICE commands that are not probe-specific.
- The I²ICE system probe software, which resides on the map-I/O board in the I²ICE instrumentation chassis. It implements the probe-specific commands.
- PSCOPE-86, which is a high-level language symbolic debugger, designed for use with Pascal-86, PL/M-86, and FORTRAN-86. It is a separate product included with the I²ICE system. It runs in the host development system. [PSCOPE-86 is an option for IBM PC hosts.]
- The I²ICE system diagnostic software, which checks the I²ICE system and supplies information about a failing system.
- The optional iLTA software which is integrated with the I²ICE system software [not available for IBM PC hosts]. The iLTA software controls the iLTA, interprets the data collected and displayed by the iLTA, and runs the I²ICE system software.
- The I²ICE tutorial software, which introduces the I²ICE command language and leads users through several on-line debugging sessions.

Software Environment

The I²ICE system software requires that the host operating system provide certain service routines. For example, the I²ICE system file handling commands assume a system service routine that provides access to the disk drives. The host operating system must run in the 8086 environment.

The software also requires formatted media in the work device. The software creates a workfile in the work device that contains the user program symbol table. This table is treated as a virtual symbol table; that is, the entire symbol table need not reside in memory at the same time. This has different implications depending on what host development system you are using.

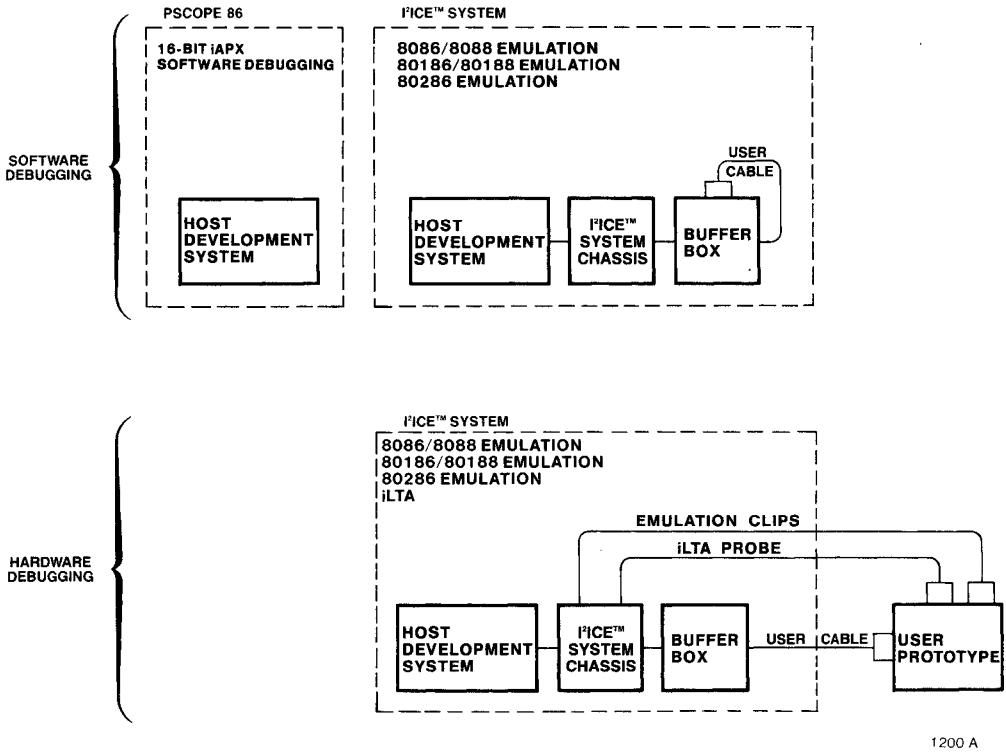


Figure 1-4 The I²ICE™ System Debugging Capabilities

- **Model 800 and Intellec® Series III development systems**
  For the ISIS operating system the default work device is :F1:. The RUN program stores the name of the current work device. You can display the name of the default work device by entering RUN WORK. You can change the default work device by entering the following command:

  RUN WORK *device-name*

  For example, RUN WORK :F2: changes the default work device to drive 2. If you reset the system, the default work device is still drive 2.

  When the Model 800 or the Series III development system is on an NDS-II, you must assign :F9: to your work device.

- **Intellec Series IV development system**
  For the iNDX operating system, the work device must be defined. For example, assume that you specify a hard disk named WD0 for the work device and the directory WORKDIR to contain the work file. The command is

  ```
  LNAME DEFINE :WORK: FOR /WD0/WORKDIR
  ```

- **IBM PC/XT and PC/AT hosts**
  For the DOS operating system, the default work device is the current default drive. To change the work device, use the following SET command:

  ```
  SET :WORK: = pathname
  ```

The I²ICE™ System Command Language

The I²ICE system command language consists of commands, pseudo-variables, and functions.

The I²ICE system commands fall into the following categories:

- Utility commands—these general-purpose commands simplify the debugging process. For example, use the EVAL command to calculate the nearest source-code line number that corresponds to the address of an assembly language instruction. Another utility command is the HELP command which provides on-line help information.
- Environment commands—these commands set up the debugging environment. For example, use the MAP command to set up a memory map which specifies that the user program memory references access memory locations in high-speed I²ICE memory.
- Debug object manipulation commands—debug objects are debug procedures, debug variables, debug registers, and LITERALLY definitions. You can create, modify, and remove debug objects. Debug objects save keystrokes during a debugging session.
- Emulation commands—you can command the I²ICE system to begin emulation and break or trace under certain specified conditions. You can specify these break and trace conditions within a debug register or within the emulation command itself.
- File handling commands—with these commands you can access disk files. For example, the LOAD command loads a user program. The LIST command sets up a list file and records what appears on the development system console. You can also save debug object definitions in a disk file and load them in later debugging sessions.
- Probe-specific commands—these commands operate on individual probes. For example, the PINS command displays the state of selected signal lines. The resulting display contains different information for different probes.

The I²ICE system functions return a requested value that depends on the current state of the environment. For example, the CI function accepts a character value from the console. It is useful when writing interactive debug procedures. Another I²ICE system function, ACTIVE, takes a symbolic user program symbol as an argument and returns a Boolean value. When TRUE, this value signifies that the user program variable exists. The variable may not exist if the I²ICE system is emulating and the current execution point has not yet reached the procedure that defines the variable.

The I²ICE system pseudo-variables are system-defined variables that can be used in expressions but cannot be removed by the user. For example, the dollar sign ($) pseudo-variable represents the current execution point. The PCHECK pseudo-variable is a probe-specific Boolean pseudo-variable. Setting PCHECK to TRUE enables protection checking for the 80286 probe.

The I²ICE™ System Software

The I²ICE system software consists of the following items:

- Host software
- Error/help software
- Probe software
- Diagnostic software
- Optional iLTA software and confidence tests
- PSCOPE-86 (optional for IBM PC hosts)
- Tutorial software

The I²ICE™ System Host Disk(s)

The host software is received on one or two disks. The disks are not system (bootable) disks. The host software is in the I2ICE.86 file for an Intellec development system (for an IBM PC host, the file is named I2ICE.EXE). The host software file is an object file.

The I²ICE™ System Error/Help Disk

The error/help disk is not a system (bootable) disk. It contains the following files:

- I2ICE.OVE—this file contains the error messages.
- I2ICE.OVH—this file contains the HELP messages.

The I²ICE™ System Probe Disks

When you invoke the I²ICE system software, the I²ICE host software is loaded into host system memory. The I²ICE host software then loads the I²ICE probe software into the I²ICE instrumentation chassis memory.

The I²ICE™ System Diagnostic Disk

The diagnostic disk is a single disk, packaged with the probe software.

The 8086/8088 diagnostic disk contains two files: ICT086.86 and ICT086.086. The ICT086.86 file loads into host memory, and the ICT086.086 file loads into I²ICE system memory.

The 80186/80188 diagnostic disk contains two files: ICT186.86 and ICT186.186. The ICT186.86 file loads into host memory, and the ICT186.186 file loads into I²ICE system memory.

The 286 diagnostic disk contains two files: ICT286.86 and ICT286.286. The ICT286.86 file loads into host memory, and the ICT286.286 file loads into I²ICE system memory.

The iLTA Disks

The iLTA software is on two double-density or three single-density disks packaged with the iLTA module. They are not system (bootable) disks.

The iLTA host software is provided on one double-density disk or two single-density disks. When contained on a double-density disk, the iLTA software has the same name as the I²ICE system host software. When contained on two single-density disks, one disk contains a file called I2ICE.1, and the other contains a file called I2ICE.2.

The remaining disk contains the iLTA confidence tests. These tests reside in an I²ICE macro file.

The PSCOPE-86 Disk

The PSCOPE-86 disk is included with every I²ICE system hosted by an Intel host. PSCOPE-86 software is an option for IBM PC hosts. The contents of the PSCOPE.86 disk for Intel hosts are as follows:

- PSCOPE.86 — the object file.
- PSCOPE.OVE — the error and HELP messages.
- PROCS.MAC — math and file handling procedures.
- *.CRT — The configuration data required for non-Intel terminals. The default configuration data are for an Intellec Series III terminal.
- DC.* SAMPLE.* — Sample programs that correspond to the examples used in the *PSCOPE-86 High-Level Program Debugger User's Guide*.
- READ.ME — A supplement to the operating information in the *PSCOPE-86 High-Level Program Debugger User's Guide*.

If you have the DOS version of PSCOPE-86, see the manual *PSCOPE-86 High Level Program Debugger User's Guide for DOS User's* for information on PSCOPE-86 files.

The I²ICE™ System Tutorial Disk(s)

The I²ICE tutorial includes more than 100 files. Depending on your host, it is received on one or two disks. The files include the following:

- SCR files. Each screen in the tutorial is contained in one file. The files are named SCR followed by the screen number. For example, the first screen is in file SCR1 and the first screen in module A is in file SCRA1.

- M files. Each module in the tutorial is activated by the commands in its associated M file. For example, to activate module B, the commands in file M.B are executed.
- I2ICE.MAC file. This I2ICE.MAC file is specially designed for use with the tutorial. When I²ICE software is activated, after the host and probe software is loaded, the I2ICE-.MAC file is executed. It causes the host system to ask users whether they want to use the tutorial.
- T.MAC file. This file is loaded when users answer "Yes" to the question displayed by the I2ICE.MAC file. The T.MAC file contains LITERALLY definitions needed to control display of the tutorial screens. It also contains definitions for several debug objects used during the tutorial.
- Source and object code files. A number of source and object code files are included. They all are versions of a change making program that is debugged during the tutorial.

I²ICE™ System Specifications

The following sections describe the I²ICE system specifications. The *I²ICE™ Data Sheet* contains additional system specifications, including the following:

- Physical characteristics
- Electrical characteristics
- Environmental requirements

Host Requirements

The host development systems that support the I²ICE system are the Model 800, the Intellec® Series III, the Intellec Series IV, and the IBM PC/XT and PC/AT. The specifications in the following sections assume a stand-alone configuration.

The Model 800 must have the following configuration:

- A system console
- Two double-density disk drives
- The iSBC® 064A memory board
- The iSBC 012B memory board
- The RPB-86 or RPC-86 board

The Model 800 does not support the iLTA.

NOTE

References to the Model 800 assume that it has been upgraded to a Series III.

The Intellec Series III must have the following configuration:

- An integral system console
- Two double-density disk drives
- The iSBC 012B memory board
- An expansion chassis (MDS-201)

The III-820 board must be installed before the iLTA can run on an Intellec Series III.

The Intellec Series IV must have the following configuration:

- An integral system console
- A hard disk—this can be the integral 5.25 in., 10M-byte Winchester disk drive or a 32M-byte peripheral Winchester disk drive
- The iSBC 012B memory board
- A 5.25 in., double-density disk drive

The IBM PC/XT and PC/AT must have the following configuration:

- 2A at 5VDC available for the I²ICE host interface board
- A hard disk
- At least 512K bytes of RAM (of which 384K bytes must be available for the I²ICE software)
- The ability to read 5 1/4 in., double-density disks (48 tpi)
- PC-DOS version 3.0 or later
- An open card slot (excluding slot J8 on the PC XT) for the I²ICE-to-PC interface board
- I/O address space 120 - 13F (hexadecimal) available for the I²ICE system

System Performance

Mappable zero wait-state memory:	Minimum 32K-bytes
Trace buffer:	1023 x 48 bits
Virtual symbol table:	The number of user program symbols is limited only by available disk space.

Emulation Clips

The emulation clipsin lines are sampled once every bus cycle when the address bits become valid on the address bus. During emulation, the I²ICE system records the value of these lines in the trace buffer, once every execution cycle. Because not all clips data is stored, a clips value can cause a break, and its data will not appear in the trace buffer. Table 1-2 lists the DC characteristics of the emulation clipsin lines.

Table 1-2 I²ICE™ System Emulation Clips—DC Characteristics

	Input Voltage		Input Current		Output Current	
	Low V_{IL} V	High V_{IH} V	Low I_{IL} uA	High I_{IH} uA	Low I_{OL} mA	High I_{OH} mA
clipsout lines					33 at 0.7 V	4.8 at 2.0 V
SYSBREAK SYSTRACE					38 at 0.7 V	1.0 at 2.0 V
clipsin lines	1.05	2.5	50	50		

The Target System User Interfaces

The three target system interfaces are the 8086/8088 probe, the 80186/80188 probe, and the 80286 probe.

Consult the I²ICE™ data sheet for the latest probe electrical characteristics and for the timing differences between the probe and the corresponding chip.

2 GUIDE TO THE I²ICE™ SYSTEM TUTORIAL

To quickly learn how to use the I²ICE™ commands and features, it is recommended that you use the I²ICE tutorial. The tutorial is the quickest way to become acquainted with the wide variety of I²ICE commands. Intel recommends that new users complete the main path of the tutorial (and many of the tutorial aid modules) before they proceed to debug their own programs.

The installation appendix for your host software explains how to install the tutorial software.

This chapter supplements the on-line I²ICE tutorial. It includes the following sections:

- Tutorial Use—This section explains how to use the tutorial, how to deactivate it, and how to reactivate it.
- Tutorial Screens and Structure—This section reproduces two screens from the tutorial. It also lists all the tutorial screens, shows a sample emulation session, provides a diagram of the tutorial flow, and provides a subject index of the tutorial.
- Tutorial Program Listings—This section provides a list file of the PL/M sample program used in the tutorial. It also includes an assembly language listing that is generated by using the CODE option while assembling the PL/M program.
- Sample Programs in C, FORTRAN, and Pascal—This section provides sample source code in C, Pascal, and FORTRAN. (The source code is also on the tutorial disk.) These programs are *not* for use with the I²ICE tutorial. They are for users who have learned I²ICE commands and wish to experiment with the commands while emulating programs in one or more of these other high-level languages.

Tutorial Use

The I²ICE tutorial is easy to access. Each time the user invokes the I²ICE software, the user's host terminal will display the following question:

```
DO YOU WANT TO USE THE I2ICE TUTORIAL? (Y OR N)
```

Invoking the Tutorial During Program Debugging

If you invoke the I²ICE software without invoking the tutorial, and later you wish to review information in one or more of the tutorial screens, you can invoke the tutorial by entering the following command.

```
INCLUDE pathname I2ICE.MAC NOLIST
```

The *pathname* provides the location of the I²ICE software. See the Pathname entry in the *I²ICE™ System Reference Manual*.

Deactivating the Tutorial

After you have become familiar with I²ICE commands, you may wish to deactivate the tutorial—that is, you may wish to eliminate the screen that asks whether you want to use the tutorial. To deactivate the tutorial, you must either delete or rename the I2ICE.MAC file.

Reactivating the Tutorial

To reactivate the tutorial, select your situation from the following two possibilities:

- **You Now Use Your Own I2ICE.MAC File.** If you created your own I2ICE.MAC file after you deactivated the tutorial I2ICE.MAC file, use operating system commands to do one of the following:
  - — Revise your I2ICE.MAC file. Locate the I2ICE.MAC file supplied with the tutorial. Append it to your I2ICE.MAC file. Examine the appended commands to ensure that they do not conflict with the commands you created.
  - — Rename your I2ICE.MAC file. Rename your I2ICE.MAC file so that you can use it later. If you previously renamed the tutorial I2ICE.MAC file, rename it again to I2ICE.MAC. If you previously deleted the tutorial I2ICE.MAC file, copy the I2ICE.MAC file from the master tutorial disk to the disk or directory that has the I²ICE software.
- **You Do Not Use Your Own I2ICE.MAC File.** If no file named I2ICE.MAC exists, use operating system commands to restore the tutorial I2ICE.MAC file. Do one of the following:
  - — If you previously renamed the tutorial I2ICE.MAC file, rename it again to I2ICE.MAC.
  - — If you deleted I2ICE.MAC from the disk or directory that has the I²ICE software, copy the I2ICE.MAC file from the master tutorial disk to the disk or directory that has the I²ICE software.

Tutorial Screens and Structure

After you have loaded the I²ICE and tutorial software (as specified in the software installation appendix for your host), screen 1 of the tutorial will be displayed (see Figure 2-1).

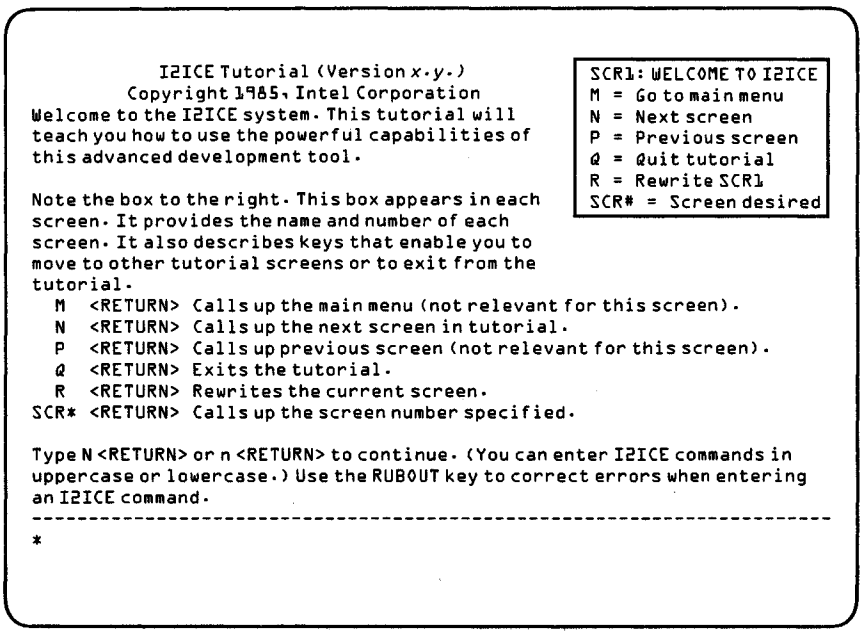


Figure 2-1 Tutorial Introductory Screen: SCR1

The tutorial is divided into a main path that has more than 45 screens, and two sets of aid screens. The main path introduces you to many of the I²ICE commands as it leads you through several emulation and debugging sessions. One set of aid screens elaborates on topics briefly mentioned in the main path. The other set of aid screens explains some of the I²ICE system features.

The I²ICE tutorial screens are created with I²ICE commands. This means that when you use the tutorial, you are also using I²ICE software. As a consequence, whenever the cursor appears next to the I²ICE prompt (*), you can enter any I²ICE commands you wish. Once you enter the I²ICE commands recommended on a particular tutorial screen, you need not immediately advance to the next screen. Instead, you can experiment with commands to ensure that you understand the concepts presented in the tutorial screen. Then, when you are ready for the next screen, you can call it up by typing N (for Next screen) followed by the <RETURN> (or <Enter>) key.

NOTE

1. For Intel hosts, commands are entered with the <RETURN> key. For IBM PC hosts, use the <Enter> key.

2. On Series III hosts, pressing <CTRL> and D at the same time produces an asterisk prompt (*), but this prompt is not for I²ICE software; it is the prompt for a development system debugger. Type G <RETURN> to return to I²ICE software.

This section provides a variety of information on the tutorial structure and contents. The main subsections and their topics are the following:

- Copies of Selected Tutorial Screens
- An Overview of the Tutorial Stucture
- List of All Tutorial Screens
- Tutorial Index

Copies of Selected Tutorial Screens

For your convenience, five tutorial screens are shown in the following subsections:

Figure 2-1 shows the first tutorial screen for an Intellec system. This screen should appear after you have loaded I²ICE tutorial software as specified in the software installation appendix for your host.

Figure 2-2 shows the main tutorial menu. (See Figures 2-3 and 2-4 for the two tutorial aid module menus.)

Some topics that are briefly introduced in the main tutorial path are explained further in emulation aid modules. Figure 2-3 shows the menu for the emulation aid modules. To display this menu on your screen, type the following (for IBM PC hosts, use <Enter> for <RETURN>):

```
AID1<RETURN>
```

Figure 2-4 shows the modules included in the I²ICE tutorial that describe I²ICE features. To display this menu on your screen, type the following (for IBM PC hosts, use <Enter> for <RETURN>):

```
AID2<RETURN>
```

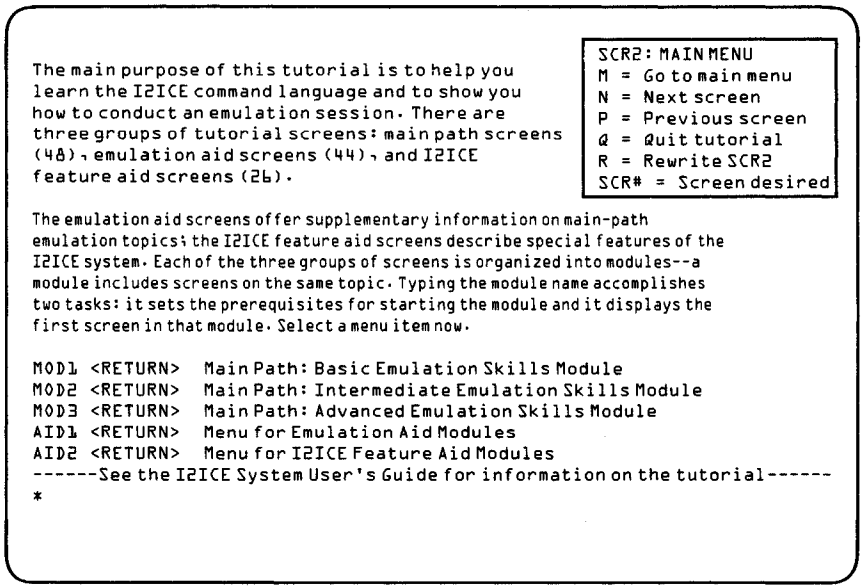


Figure 2-2 Tutorial Main Menu: SCR2

Screens SCR12 through SCR15 explain the input/output display that occurs in the first emulation session. Because the complete emulation display is long and somewhat complex, the tutorial user is asked to enter commands that interrupt the display so that individual portions of the display can be explained. Figure 2-5 shows how the display would appear if it were not interrupted. (The emulation display is longer than one screen; Figure 2-5 extends the screen size to show the entire display.)

An Overview of the Tutorial Structure

Figure 2-6 provides an overview of the tutorial structure. On the left are the three main-path modules and their associated screens. In the center are the emulation aid modules. Arrows indicate the main path screens that reference the emulation aid modules. On the right are the feature aid modules.

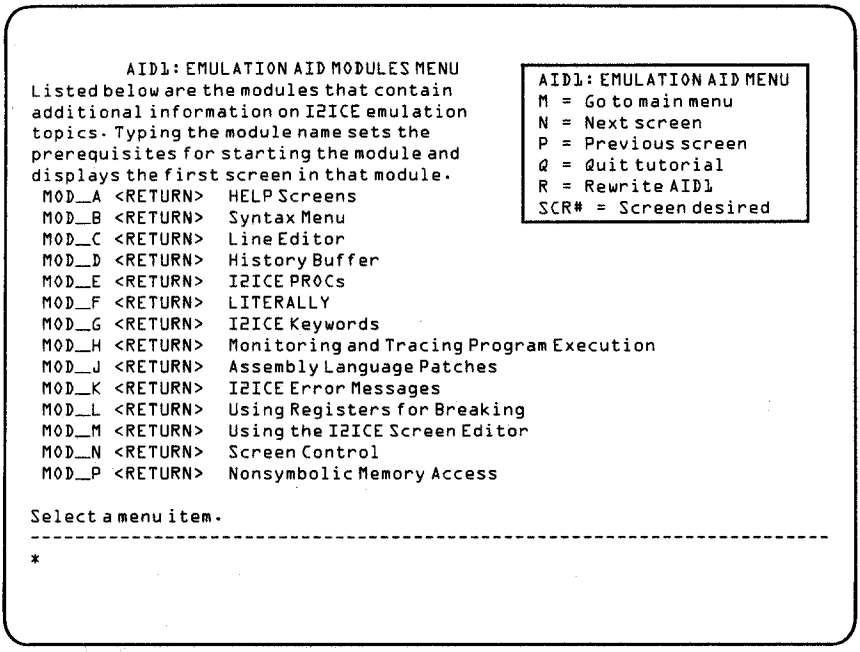


Figure 2-3 Menu for the Emulation Aid Modules: AID1

List of All Tutorial Screens

Tables 2-1, 2-2, and 2-3 list all the tutorial screens, as follows:

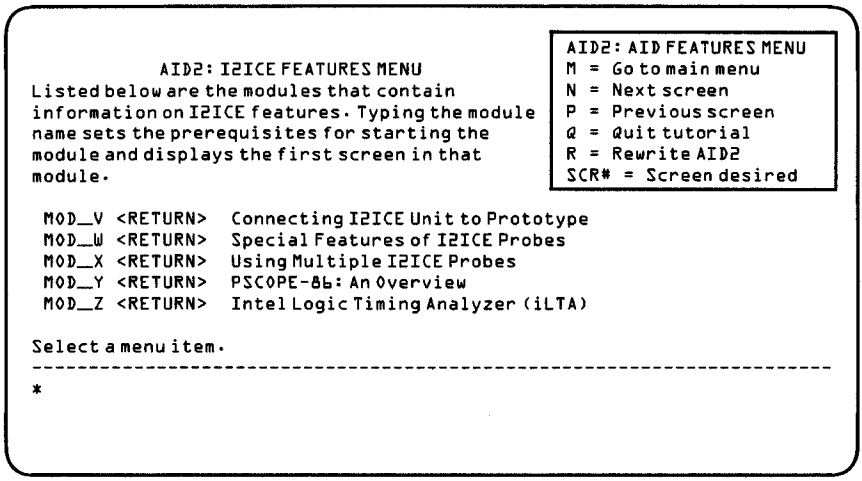
Table 2-1 Main Tutorial Path Screens

Table 2-2 Emulation Aid Module (AID1) Screens

Table 2-3 I²ICE™ System Feature Aid Module (AID2) Screens

Each module is a major division of the tutorial. The modules can be entered in two ways: typing the module name or typing the name of the first screen in the module.

- Typing the module name. Each module has a name (e.g., MOD1, MOD_A). Typing the module name loads the first screen in the module and sets up any prerequisites needed to carry out the steps in the module. (For example, if the module assumes that the first 2K bytes of memory is mapped to high-speed memory (HS), typing the module name maps 2K bytes to HS and loads the first screen in the module.)



NOTE: The tutorial for IBM hosts does not have MOD_Z.

Figure 2-4 Menu for the I²ICE™ Feature Modules: AID2

- Typing the screen name. Each screen has a name (e.g., SCR3, SCRA1). You can display any screen by simply typing its name; however, if you enter a module using the name of the first screen in the module (rather than the module name), you will not set prerequisites for that module. (Note that not all modules have prerequisites.)

SCREEN OUTPUT

```
?UNIT 0 PORT 0002H OUTPUT BYTE 000AH
?UNIT 0 PORT 0002H OUTPUT BYTE 000DH
?UNIT 0 PORT 0002H OUTPUT BYTE 0050H
?UNIT 0 PORT 0002H OUTPUT BYTE 0061H
?UNIT 0 PORT 0002H OUTPUT BYTE 0069H
?UNIT 0 PORT 0002H OUTPUT BYTE 0064H
?UNIT 0 PORT 0002H OUTPUT BYTE 0020H
?UNIT 0 PORT 0002H OUTPUT BYTE 003DH
?UNIT 0 PORT 0002H OUTPUT BYTE 0020H
?UNIT 0 PORT 0001H REQUESTS BYTE INPUT (ENTER VALUE): 31
?UNIT 0 PORT 0002H OUTPUT BYTE 0031H
?UNIT 0 PORT 0001H REQUESTS BYTE INPUT (ENTER VALUE): 35
?UNIT 0 PORT 0002H OUTPUT BYTE 0035H
?UNIT 0 PORT 0001H REQUESTS BYTE INPUT (ENTER VALUE): 30
?UNIT 0 PORT 0002H OUTPUT BYTE 0030H
?UNIT 0 PORT 0001H REQUESTS BYTE INPUT (ENTER VALUE): 0D
?UNIT 0 PORT 0002H OUTPUT BYTE 000AH
?UNIT 0 PORT 0002H OUTPUT BYTE 000DH
?UNIT 0 PORT 0002H OUTPUT BYTE 0050H
?UNIT 0 PORT 0002H OUTPUT BYTE 0072H
?UNIT 0 PORT 0002H OUTPUT BYTE 0069H
?UNIT 0 PORT 0002H OUTPUT BYTE 0063H
?UNIT 0 PORT 0002H OUTPUT BYTE 0065H
?UNIT 0 PORT 0002H OUTPUT BYTE 0020H
?UNIT 0 PORT 0002H OUTPUT BYTE 003DH
?UNIT 0 PORT 0002H OUTPUT BYTE 0020H
?UNIT 0 PORT 0001H REQUESTS BYTE INPUT (ENTER VALUE): 31
?UNIT 0 PORT 0002H OUTPUT BYTE 0031H
?UNIT 0 PORT 0001H REQUESTS BYTE INPUT (ENTER VALUE): 30
?UNIT 0 PORT 0002H OUTPUT BYTE 0030H
?UNIT 0 PORT 0001H REQUESTS BYTE INPUT (ENTER VALUE): 38
?UNIT 0 PORT 0002H OUTPUT BYTE 0038H
?UNIT 0 PORT 0001H REQUESTS BYTE INPUT (ENTER VALUE): 0D
?UNIT 0 PORT 0002H OUTPUT BYTE 000AH
?UNIT 0 PORT 0002H OUTPUT BYTE 000DH
?UNIT 0 PORT 0002H OUTPUT BYTE 000AH
?UNIT 0 PORT 0002H OUTPUT BYTE 000DH
?UNIT 0 PORT 0002H OUTPUT BYTE 004EH
?UNIT 0 PORT 0002H OUTPUT BYTE 006FH
?UNIT 0 PORT 0002H OUTPUT BYTE 0020H
?UNIT 0 PORT 0002H OUTPUT BYTE 0063H
?UNIT 0 PORT 0002H OUTPUT BYTE 0068H
?UNIT 0 PORT 0002H OUTPUT BYTE 0061H
?UNIT 0 PORT 0002H OUTPUT BYTE 006EH
?UNIT 0 PORT 0002H OUTPUT BYTE 0067H
?UNIT 0 PORT 0002H OUTPUT BYTE 0065H
?Probe 0 stopped at :CMAKER#3 because of bus not active
  Bus address = 0602F4 Trace Buffer Overflow
```

TRANSLATION

line feed
carriage return
P
a
i
d
space
=
space
1
Echoes the 1
5
Echoes the 5
0
Echoes the 0
Carriage return
Line Feed
Carriage return
P
r
i
c
e
space
=
space
1
Echoes the 1
0
Echoes the 0
8
Echoes the 8
Carriage return
Line feed
Carriage return
Line feed
Carriage return
N
o
space
c
h
a
n
g
e

Figure 2-5 Emulation Display for Screens SCR12 Through SCR15

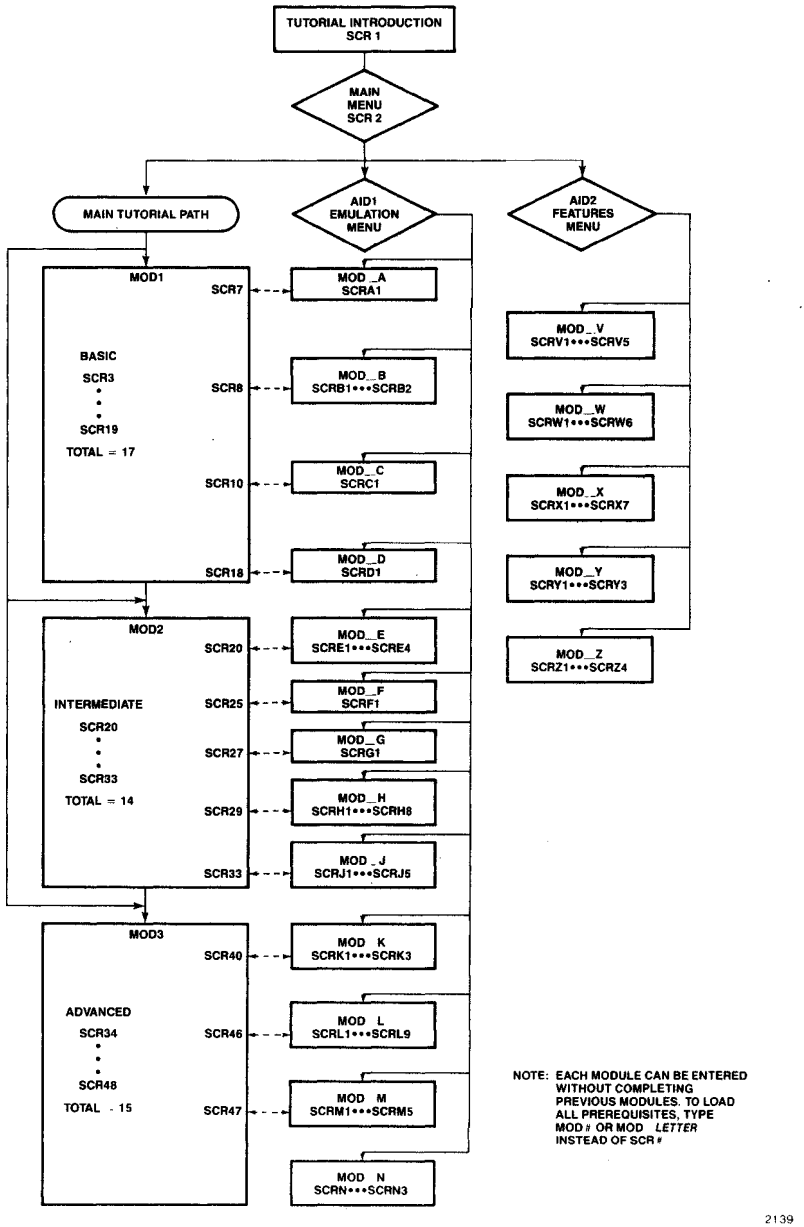


Figure 2-6 Tutorial Structure

Table 2-1 Main Tutorial Path Screens

Module Name	Screen Name	Screen Title	Topic
	SCR1	Welcome to I2ICE	(These screens are reproduced in Figures 2-1 through 2-4.)
	SCR2	Main Menu	
	AID1	Emulation Aid Menu	
	AID2	Aid Features Menu	
MOD1 *Basic Emulation Skills*	SCR3	Introduction	In this module, you work with the MAP and MAPIO commands, learn how to interpret I²ICE output, and find the first program bug.
	SCR4	Memory Mapping	
	SCR5	Memory Allocation	
	SCR6	Map Choices	
	SCR7	MAPIO	
	SCR8	I/O Allocation	
	SCR9	LOAD	
	SCR10	Save Address	
	SCR11	Output Line	
	SCR12	HOLDIO/HALT	
	SCR13	Paid Request	
	SCR14	Input Paid	
	SCR15	Price Request	
	SCR16	Bug Found	
	SCR17	Variable Values	
	SCR18	Addresses	
	SCR19	Module 1 End	
MOD2 *Inter-mediate Emulation Skills*	SCR20	Ioproc	In this module, you learn how to improve the screen display using a procedure (PROC) written with I²ICE commands. Then you use a high-level patch to fix the bug discovered in Module 1.
	SCR21	Go with Ioproc	
	SCR22	WAIT	
	SCR23	Change	
	SCR24	Value	
	SCR25	NAMESCOPE	
	SCR26	Breakpoint	
	SCR27	Bug#1	
	SCR28	Bug Explained	
	SCR29	I2ICE TRACE	
	SCR30	Fix Errors	
	SCR31	Hi-level Patch	
	SCR32	Bug1 PROC	
	SCR33	GO WITH Patch	
MOD3 *Advanced Emulation Skills*	SCR34	Module 3 Intro	In this modle, you learn some advanced debugging techniques using arm, break, and system specifications.
	SCR35	Specifying BREAK	
	SCR36	BREAK a "a"	
	SCR37	System Spec	
	SCR38	Arming at "1"	
	SCR39	Rearming	
	SCR40	Ascii_digit	
	SCR41	Decimal_digit 1	
	SCR42	Decimal_digit 2	
	SCR43	Decimal_Digit 3	
	SCR44	Array[5]	
	SCR45	BYTE	
	SCR46	Bug 2 Found	
	SCR47	Fix Source Code	
	SCR48	Main Path End	

Table 2-2 Emulation Aid Module (AID1) Screens

Module Name	Screen Name	Screen Title	Topic
MOD_A	SCRA1	HELP Screens	(Reference SCR7)
MOD_B	SCRB1 SCRB2	Syntax MENU 1 Syntax MENU 2	(Reference SCR8)
MOD_C	SCRC1	Line Editor	(Reference SCR10)
MOD_D	SCRD1	History Buffer	(Reference SCR18)
MOD_E *I²ICE PROCs*	SCRE1 SCRE2 SCRE3 SCRE4	PROCs 1 PROCs 2 PROCs 3 PROCs 4	These screens explain how to DEFINE an I²ICE procedure. (Reference SCR20)
MOD_F	SCRF1	LITERALLY, MENU	(Reference SCR25)
MOD_G	SCRG1	I²ICE Keywords	(Reference SCR27)
MOD_H *Monitoring and Tracing Program Execution*	SCRH1 SCRH2 SCRH3 SCRH4 SCRH5 SCRH6 SCRH7 SCRH8	Module H Intro LSTEP PSTEP ISTEP TRACE Data GO TRACE TO TRACE Registers CYCLES Mode	These screens explain single-stepping and the I²ICE trace capability. (Reference SCR29)
MOD_J *Assembly Language Patches*	SCRJ1 SCRJ2 SCRJ3 SCRJ4 SCRJ5	Module J Intro The Patch Patch Error Rerun Patch Patch Works	These screens explain patching with assembly language. (Reference SCR33)
MOD_K *Error Messages*	SCRK1 SCRK2 SCRK3	Error Info 1 Error Info 2 Error Info 3	These screens explain the five types of I²ICE system errors. (Reference SCR40)
MOD_L *Using Registers for Breaking*	SCRL1 SCRL2 SCRL3 SCRL4 SCRL5 SCRL6 SCRL7 SCRL8 SCRL9	BRKREG 1 BRKREG 2 BRKREG 3 BRKREG 4 BRKREG 5 BRKREG 6 BRKREG 7 BRKREG 8 BRKREG 9	These screens explain breaking with break, system, or event registers. (Reference SCR46)
MOD_M *Using the I²ICE Screen Editor*	SCRM1 SCRM2 SCRM3 SCRM4 SCRM5	Screen Editor 1 Screen Editor 2 Screen Editor 3 Screen Editor 4 Screen Editor 5	These screens explain the I²ICE editor AEDIT commands. (Reference SCR47)
MOD_N *Screen Control*	SCRN1 SCRN2 SCRN3	Module N Intro Screen Control WRITE	These screens explain how to control cursor movement. (Reference AID1)

Table 2-2 Emulation Aid Module (AID1) Screens (continued)

Module Name	Screen Name	Screen Title	Topic
MOD_P *Non-Symbolic Memory Access*	SCRP1 SCRP2 SCRP3 SCRP4 SCRP5 SCRP6 SCRP7 SCRP8	Module P Intro Number Base Memory Types 1 Memory Types 2 Memory Types 3 Setting Memory 1 Setting Memory 2 Setting Memory 3	These screens explain how to display and set memory values using the memory type of your choice. (Reference SCR45)

Table 2-3 I²ICE™ Feature Aid Module (AID2) Screens

Module Name	Screen Name	Screen Title	Topic
MOD_V *Connecting an I²ICE Unit to Prototype*	SCRV1 SCRV2 SCRV3 SCRV4 SCRV5	Module V Intro Pseudo Variables User Cable Emulation Clips 1 Emulation Clips 2	Module V tells how to connect user hardware.
MOD_W *Special Features of I²ICE Probes*	SCRW1 SCRW2 SCRW3 SCRW4 SCRW5 SCRW6	Probe REGS Alter REGS HELP REGS PINS & STATUS Probe Status End Probes	Module W explains the three I²ICE probes.
MOD_X *Using Multiple I²ICE Probes*	SCRX1 SCRX2 SCRX3 SCRX4 SCRX5 SCRX6 SCRX7	Multiple Probes UNIT Commands UNIT LOAD BREAK/TRACE Multi-probe Multi-probe TRACE Multi-probe WAIT	Module X briefly explains how to use up to four probes on one host development system.
MOD_Y *PSCOPE-86*	SCRY1 SCRY2 SCRY3	PSCOPE-86 Intro PSCOPE Loading PSCOPE Editor	Module Y introduces the PSCOPE software debugger.
MOD_Z *Intel Logic Timing Analyzer— iLTA*	SCRZ1 SCRZ2 SCRZ3 SCRZ4 SCRZ5	iLTA Intro Install iLTA Trigger Setup Timing Display State Display	Module Z introduces the Intel Logic Timing Analyzer.

Tutorial Index

The following index correlates I²ICE tutorial screen suffixes with tutorial topics and I²ICE commands. To display any screen cited in the index, add the tutorial screen prefix SCR to the suffix and type <RETURN> (or <Enter>). For example, if the index entry of interest is K1, type SCRK1 <RETURN> (or <Enter>) to display the screen.

Subject	Screen Suffix
Error info	K1, K2, K3
ESC key	Y3
Event specifications	37
FLAGS	W2
Fully qualified reference	17, 26, 40, 43
GO	12
GO FROM	14, 15, 16, 21, 25, 33, 42, H6
GO TIL	35, 37, 42
GO/TRACE	H6
GO USING	33
HELP	7, A1, V2, Y2
HELP REGS	W3
History buffer	18, 22, D1
I²ICE keywords	17, 29, G1
iLTA	Z1 through Z5
INCLUDE	20, E3
INSTRUCTIONS	H4, H5, H6
ISTEP	H3, H4
LENGTH	5, 8, 21, 45
Line editor	10, C1
Line numbers	26
LITERALLY	25, F1
LOAD	9, 34, 47, X3,
LSTEP	H1, H2
MAP	4, 5, 6
MAPIO	7, 8, 20, 21
Memory access	P1 through P8
Memory types	P3 through P5
MENU	8, B1, B2
Menu screens	2, AID1, AID2*
Monitoring and Tracing	H1
Multiple probes	X1 through X7
NAMESCOPE	25
NEWEST	see OLDEST
NOLIST	E3
OLDEST	H4, H5
Output	11, 37, 38, 39

*Do not type the SCR prefix for AID1 and AID2

Subject	Screen Suffix
Patch, assembly-language	33, J1 through J5
Patch, high-level	30, 31
PINS	W4
POINTER	10
PORTDATA	20
PRINT	H4 through H8
Probes	W1 through W6
Probe status	W5
PROCs	20, 30, 31, 32, E1 through E4
Prototype hardware	V1 through V5
PSCOPE-86	Y1, Y2, Y3
Pseudo-variables	V2
PSTEP	H3
PUT	E3
Rearming	39
REGS	W1, W2
REMOVE	E3
RESET BREAK	32, 34, H6
RESET MAPIO	8
SASM	J1, J2
SAVE	P6
Screen control	N1, N2, N3
Screen editor	47, M1 through M5
STATUS	V2, W4
Symbols	18
Syntax menu	B1
SYSBREAKIN	X5, X6
SYSTEM ARM	X5
System specifications	35, 36, 37
SYSTRACE	X1, X4, X6
SYSTRIG	X1, X4, X5
Tab key	B1
Trace	29, H5, H6, X4
Trace buffer	16, 29, H4, H6
TRCBUS	H7, H8
TRCREG	H7
TRIG	38, 39
Tutorial menu	2
UNIT	11, X2, X3, W1
UNITHOLD	V3
Up arrow	18, 22, D1

Subject	Screen Suffix
Variable address	18
Variable values	17, 23, 24
WAIT	22, 23, 33, X7
WRITE	E2, M4, N3

Tutorial Program Listings

The I²ICE tutorial disk has source programs in PL/M, C, FORTRAN, and Pascal. However, the tutorial is designed to only be used with the PL/M program. The C, FORTRAN, and Pascal programs are included on the tutorial disk for convenience. After you learn I²ICE commands, you may wish to experiment with the commands while emulating the sample C, FORTRAN, or Pascal programs. See the sample programs in C, FORTRAN, and Pascal in this chapter to learn more about them. This section focuses only on the PL/M program (and its equivalent ASM program).

This part has three main sections:

- Overview of the PL/M Tutorial Program
- PL/M Program Listing for the Two-bug Version of the Change Maker Program
- The ASM-86 Listing for the No-bug Version of the Change Maker Program

Overview of the PL/M Tutorial Program

The program used for the tutorial debug session is written in PL/M. It is a change making program. The user is first asked to enter an integer amount (cents) for the money paid; then, the user is asked to enter an integer amount (cents) for the purchase price. The program calculates the change in dollars and coins, and prints out a listing of the change.

The tutorial disk contains six files associated with the PL/M tutorial program. All of these files begin with the prefix CMKER.

There are three versions of the PL/M program in absolute code. To make the debugging session realistic, the initial version of the PL/M program contains two errors that do not affect compilation, but show up at run time. A second version has one of the errors corrected, and the third version is error-free. The names of the three versions are as follows:

CMKER2.ABS	absolute code with two bugs
CMKER1.ABS	absolute code with one bug
CMKER0.ABS	absolute code with no bugs

There are two versions of the PL/M program in source code—the code in each version is the same but the comments are different. These two versions are used in the tutorial for two tutorial sessions that introduce the I²ICE screen editor. In the sessions, users are asked to edit the PL/M source code to correct the two bugs detected during emulation. The names of the two source code files are as follows:

CMKER2.S1 Source code with two bugs. This code is used in editing session 1.

CMKER2.S2 Also source code with two bugs. This version is used in editing session 2.

In addition to the three absolute code CMKER files and the two source code CMKER files, the tutorial disk also contains the following CMKER list file:

CMKER0.LST A list file that includes the assembly code.

This file is supplied for your convenience. It is not used during the tutorial.

PL/M Program Listing for the Two-Bug Version of the Change Maker Program

The tutorial refers to the list file in this section; the listing is for the CMKER program version with two bugs. (The two bugs are marked in the listing.)

The program was compiled using OPTIMIZE(0). Compile options are explained in the *PL/M-86 Programming Manual* (order number 121636).

```
1       Cmaker:DO;
2  1    DECLARE
          TRUE literally '0FFH',
          FALSE literally '0',
          IN_PORT literally '1',
          OUT_PORT literally '2',
          lf_cr (2) BYTE DATA (0AH, 0DH),                    /* Line feed, return     */
          paid text (*) BYTE DATA (7, 'Paid = '),            /* Facilitates user      */
          purchased_text (*) BYTE DATA (8, 'Price = '),      /* interaction           */
          coins                WORD,
          change               WORD,
          dollars              WORD,
          quarters             WORD,
          nickels              WORD,
          dimes                WORD,
          pennies              WORD,
          paid                 WORD,
          purchased            WORD;
3  1    read: PROCEDURE BYTE;                  /* Reads port 1 input from an ASCII  */
4  2    DECLARE                                /* keyboard and stores 1 byte at a   */
```

```
                char BYTE;                          /* time; also writes an input byte to      */
                                                    /* an output port                          */
 5  2             char = INPUT(in_port);
 6  2             IF char = ODH THEN
 7  2                 CALL write (@lf_cr, 1);
 8  2             CALL write (@char, 1);
 9  2             RETURN char;
10  2         END read;
11  2         write: PROCEDURE (text_ptr, text_cnt);      /* Write text_cnt characters  */
12  2             DECLARE                                 /* at text_ptr to output port */
                    text_ptr POINTER,
                    text_cnt WORD,
                    text_BASED text_ptr (1) BYTE,
                    i WORD;
13  2             DO i = 0 TO text_cnt - 1;
14  3                 OUTPUT(out_port) = text(i);
15  3                 END;
16  2         END write;

17  1         write_decimal: PROCEDURE (value);           /* Store digits in            */
18  2             DECLARE                                 /* stack-like array           */
                    value WORD,
                    i WORD,
                    digit_stack(5) BYTE,                /* 0..4                       */
                    stk_top WORD;

19  2             stk_top = 5;
20  2             DO WHILE value > 0;
21  3                 digit_stack(stk_top := stk_top - 1) = value MOD 10;
22  3                 value = value/10;
23  3             END;

24  2             DO i = stk_top TO 5;                    /* BUG  5 should be 4         */
25  3                 CALL convert_decimal_digit(digit_stack(i));
26  3             END;
27  2         END write_decimal;

28  1         convert_decimal_digit: PROCEDURE (decimal_digit);  /* To ASCII           */
29  2             DECLARE
                    decimal_digit BYTE,
                    ascii_digit BYTE;

30  2                 ascii_digit = decimal_digit + '0';      /* Character '0' is 30H   */
31  2                 CALL write(@ascii_digit, 1);
32  2         END convert_decimal_digit;
33  1         print_coin: PROCEDURE (text_ptr, value);    /* Formats writing of change  */
34  2             DECLARE
                    text_ptr POINTER,                   /* Tells where                */
                    value WORD,                         /* Tells what                 */
```

```
        text BASED text_ptr STRUCTURE (cnt BYTE, string(1) BYTE),
                                    /*  Pointer location can change with the program   */
        text_cnt BYTE;

35  2   IF value = 0 THEN
36  2    RETURN;
37  2   CALL write (@('   '), 2);                 /*  Insert 2 blanks                 */
38  2   CALL write_decimal(value);                /*  Value of change                 */
39  2   text_cnt = text.cnt;
40  2    IF value = 1 THEN
41  2      text_cnt = text_cnt - 1;               /*  Dollars = dollar, etc.          */
42  2    CALL write(@text,string, text_cnt);
43  2    CALL write (@lf_cr, 2);                  /*  Next line                       */
44  2  END print_coin;

45  1  payment PROCEDURE;                         /*  Formats writing change          */
46  2     DECLARE
          s_text (*) BYTE DATA (8, ' dollars'),   /*  (*) is an implicit              */
          q_text (*) BYTE DATA (9, ' quarters'),  /*  dimension specifier             */
          n_text (*) BYTE DATA (8, ' nickels'),
          d_text (*) BYTE DATA (6, ' dimes'),
          p_text (*) BYTE DATA (8, ' pennies'),
          p1_text(*) BYTE DATA (7, ' penny');

47  2     IF (dollars OR quarters OR dimes OR nickels OR pennies) = 0 THEN
48  2      CALL write(@('No change'), 9);
49  2     ELSE
             DO;
50  3           CALL write (@('Change = '), 15);
51  3           CALL write (@lf_cr, 2);
52  3           CALL print_coin(@s_text, dollars);
53  3           CALL print_coin(@q_text, quarters);
54  3           CALL print_coin(@d_text, dimes);
55  3           CALL print_coin(@n_text, nickels);
56  3            IF pennies = 1 THEN
57  3              CALL print_coin(@p1_text, pennies);
58  3           ELSE
                    CALL print_coin(@p_text, pennies);
59  3         END;
60  2  END payment;

61  1  get_input: PROCEDURE (text_ptr) WORD;      /*  Converts ASCII code             */
62  2     DECLARE                                /*  to decimal value                */
          text_ptr POINTER,
          value_ptr POINTER,
          value WORD,
          char BYTE,
          not_done BYTE,
          text BASED text_ptr STRUCTURE (cnt BYTE, string(1) BYTE);
```

```
63  2       CALL write(@text.string, text.cnt):
64  2       value = 0;
65  2       not_done = TRUE;
66  2       DO WHILE not_done;                          /* Flag                          */
67  3        char = read;
68  3       IF (char >= '0') AND (char <= '9') THEN     /* Keyboard entry can            */
69  3          DO;                                      /* be 0 through 9                */
                          /* Translate the ASCII hex character to the decimal value    */
70  4             value = value + char - '0';           /* BUG  value=value*10           */
71  4             value = value / 10;                   /* BUG  =value+char-'0'          */
72  4          END:
73  3       ELSE
                  not_done = false;
74  2        END;
75  2        RETURN value;
76  2  END get_input;

77  1  begin;:                                          /* Mainline                      */
78  1     DO;
79  2        CALL write (@lf_cr, 2);                   /* Get amount paid               */
80  2        paid = get_input(@paid_text);
81  2        purchased = get_input(@purchased_text);   /* and purchase price            */
82  2        CALL write (@lf_cr, 2);
83  2        change = paid - purchased;                /* Figure change                 */
84  2        dollars = change /100;                    /* How many dollars?             */
85  2        coins = change MOD 100;                   /* Are there coins?              */
86  2        quarters = coins/25;                      /* How many quarters             */
87  2        coins = coins MOD 25;                     /* etc.                          */
88  2        dimes = coins/10;
89  2        coins = coins MOD 10;
90  2        nickels = coins/5;
91  2        pennies = coins MOD 5;
92  2        IF paid < purchased THEN
93  2           CALL write(@('NO CHEATING!'), 12);
94  2        ELSE
              CALL payment;
95  2     END;
96  1     HALT;
97  1  END;
```

The ASM-86 Listing for the No-Bug Version of the Change Maker Program

The ASM-86 language listing for the change maker program was obtained by compiling the corrected version of the PL/M program using the CODE option. It is recommended that the compilation be done using OPTIMIZE(0) when debugging. In the assembly listing, notice the statement numbers at the right. The numbers reference the lines in the PL/M program list file. On the tutorial disk, this list file of corrected code is called CMKER0.LST.

Note that the compiler was invoked by the following command:

```
PLM86.86 CMKERN.SRC COMPACT OPTIMIZE(0) DEBUG CODE
```

ASSEMBLY LISTING OF OBJECT CODE

```
                                                        ; STATEMENT # 3
                READ            PROC NEAR
00D8  55            PUSH    BP
00D9  8BEC          MOV     BP,SP
                                                        ; STATEMENT # 5
00DB  E401          IN      1H
00DD  88061E00      MOV     CHAR,AL
                                                        ; STATEMENT # 6
00E1  803E1E000D    CMP     CHAR,0DH
00E6  7403          JZ      $+5H
00E8  E90C00        JMP     @1
                                                        ; STATEMENT # 7
00EB  B80000        MOV     AX,OFFSET(LF_CR)
00EE  1E            PUSH    DS                          ; 1
00EF  50            PUSH    AX                          ; 2
00F0  B80100        MOV     AX,1H
00F3  50            PUSH    AX                          ; 3
00F4  E81200        CALL    WRITE
                                                        ; STATEMENT # 8
                @1:
00F7  B81E00        MOV     AX,OFFSET(CHAR)
00FA  1E            PUSH    DS                          ; 1
00FB  50            PUSH    AX                          ; 2
00FC  B80100        MOV     AX,1H
00FF  50            PUSH    AX                          ; 3
0100  E80600        CALL    WRITE
                                                        ; STATEMENT # 9
0103  8A061E00      MOV     AL,CHAR
0107  5D            POP     BP
0108  C3            RET
                                                        ; STATEMENT # 10
                READ            ENDP
                                                        ; STATEMENT # 11
                WRITE           PROC NEAR
```

```
0109  55              PUSH   BP
010A  8BEC            MOV    BP,SP
                                                    ; STATEMENT # 13
010C  C70612000000    MOV    I,0H
                  @2:
0112  8B4604          MOV    AX,[BP].TEXT_CNT
0115  48              DEC    AX
0116  3B061200        CMP    AX,I
011A  7303            JAE    $+5H
011C  E91A00          JMP    @3
                                                    ; STATEMENT # 14
011F  C45E06          LES    BX,[BP].TEXT_PTR
0122  8B361200        MOV    SI,I
0126  268A00          MOV    AL,ES:[BX].TEXT[SI]
0129  E602            OUT    2H
                                                    ; STATEMENT # 15
012B  8B061200        MOV    AX,I
012F  40              INC    AX
0130  89061200        MOV    I,AX
0134  7403            JZ     $+5H
0136  E9D9FF          JMP    @2
                  @3:
                                                    ; STATEMENT # 16
0139  5D              POP    BP
013A  C20600          RET    6H
                  WRITE                 ENDP
                                                    ; STATEMENT # 17
                  WRITE_DECIMAL         PROC NEAR
013D  55              PUSH   BP
013E  8BEC            MOV    BP,SP
                                                    ; STATEMENT # 19
0140  C70616000500    MOV    STK_TOP,5H
                                                    ; STATEMENT # 20
                  @4:
0146  837E0400        CMP    [BP].VALUE,0H
014A  7503            JNZ    $+5H
014C  E92900          JMP    @5
                                                    ; STATEMENT # 21
014F  8B061600        MOV    AX,STK_TOP
0153  48              DEC    AX
0154  89061600        MOV    STK_TOP,AX
0158  50              PUSH   AX          ; 1
0159  8B4604          MOV    AX,[BP].VALUE
015C  B90A00          MOV    CX,0AH
015F  31D2            XOR    DX,DX
0161  F7F1            DIV    CX
```

```
0163  5B              POP     BX          ; 1
0164  88971F00        MOV     DIGIT_STACK[BX],DL
                                                        ; STATEMENT # 22
0168  8B4604          MOV     AX,[BP].VALUE
016B  B90A00          MOV     CX,0AH
016E  31D2            XOR     DX,DX
0170  F7F1            DIV     CX
0172  894604          MOV     [BP].VALUE,AX
                                                        ; STATEMENT # 23
0175  E9CEFF          JMP     @4
                  @5:
                                                        ; STATEMENT # 24
0178  8B061600        MOV     AX,STK_TOP
017C  89061400        MOV     I,AX
                  @6:
0180  813E14000400    CMP     I,4H
0186  7603            JBE     $+5H
0188  E91900          JMP     @7
                                                        ; STATEMENT # 25
018B  8B1E1400        MOV     BX,I
018F  FFB71F00        PUSH    DIGIT_STACK[BX]; 1
0193  E81200          CALL    CONVERT_DECIMAL_DIGIT
                                                        ; STATEMENT # 26
0196  8B061400        MOV     AX,I
019A  40              INC     AX
019B  89061400        MOV     I,AX
019F  7403            JZ      $+5H
01A1  E9DCFF          JMP     @6
                  @7:
                                                        ; STATEMENT # 27
01A4  5D              POP     BP
01A5  C20200          RET     2H
                  WRITE_DECIMAL     ENDP
                                                        ; STATEMENT # 28
                  CONVERT_DECIMAL_DIGIT     PROC NEAR
01A8  55              PUSH    BP
01A9  8BEC            MOV     BP,SP
                                                        ; STATEMENT # 30
01AB  8A4604          MOV     AL,[BP].DECIMAL_DIGIT
01AE  80C030          ADD     AL,30H
01B1  88062400        MOV     ASCII_DIGIT,AL
                                                        ; STATEMENT # 31
01B5  B82400          MOV     AX,OFFSET(ASCII_DIGIT)
01B8  1E              PUSH    DS                        ; 1
01B9  50              PUSH    AX                        ; 2
01BA  B80100          MOV     AX,1H
01BD  50              PUSH    AX                        ; 3
```

```
01BE  E848FF          CALL    WRITE
                                                        ; STATEMENT # 32
01C1  5D              POP     BP
01C2  C20200          RET     2H
                    CONVERT_DECIMAL_DIGIT       ENDP
                                                        ; STATEMENT # 33
                    PRINT_COIN              PROC NEAR
01C5  55              PUSH    BP
01C6  8BEC            MOV     BP,SP
                                                        ; STATEMENT # 35
01C8  837E0400        CMP     [BP].VALUE,0H
01CC  7403            JZ      $+5H
01CE  E90400          JMP     @8
                                                        ; STATEMENT # 36
01D1  5D              POP     BP
01D2  C20600          RET     6H
                                                        ; STATEMENT # 37
                    @8:
01D5  B84600          MOV     AX,OFFSET(@@LONG$CONSTANT$0046H)
01D8  1E              PUSH    DS                        ; 1
01D9  50              PUSH    AX                        ; 2
01DA  B80200          MOV     AX,2H
01DD  50              PUSH    AX                        ; 3
01DE  E828FF          CALL    WRITE
                                                        ; STATEMENT # 38
01E1  FF7604          PUSH    [BP].VALUE; 1
01E4  E856FF          CALL    WRITE_DECIMAL
                                                        ; STATEMENT # 39
01E7  C45E06          LES     BX,[BP].TEXT_PTR
01EA  268A07          MOV     AL,ES:TEXT[BX]
01ED  88062500        MOV     TEXT_CNT,AL
                                                        ; STATEMENT # 40
01F1  817E040100      CMP     [BP].VALUE,1H
01F6  7403            JZ      $+5H
01F8  E90A00          JMP     @9
                                                        ; STATEMENT # 41
01FB  8A062500        MOV     AL,TEXT_CNT
01FF  FEC8            DEC     AL
0201  88062500        MOV     TEXT_CNT,AL
                                                        ; STATEMENT # 42
                    @9:
0205  C45E06          LES     BX,[BP].TEXT_PTR
0208  8D4701          LEA     AX,TEXT[BX+1H]
020B  06              PUSH    ES                        ; 1
020C  50              PUSH    AX                        ; 2
020D  8A062500        MOV     AL,TEXT_CNT
0211  B400            MOV     AH,0H
```

```
0213  50            PUSH   AX                                  ; 3
0214  E8F2FE        CALL   WRITE
                                                               ; STATEMENT # 43
0217  B80000        MOV    AX,OFFSET(LF_CR)
021A  1E            PUSH   DS                                  ; 1
021B  50            PUSH   AX                                  ; 2
021C  B80200        MOV    AX,2H
021F  50            PUSH   AX                                  ; 3
0220  E8E6FE        CALL   WRITE
                                                               ; STATEMENT # 44
0223  5D            POP    BP
0224  C20600        RET    6H
                PRINT_COIN      ENDP
                                                               ; STATEMENT # 45
                PAYMENT         PROC NEAR
0227  55            PUSH   BP
0228  8BEC          MOV    BP,SP
                                                               ; STATEMENT # 47
022A  8B060400      MOV    AX,DOLLARS
022E  0B060600      OR     AX,QUARTERS
0232  0B060A00      OR     AX,DIMES
0236  0B060800      OR     AX,NICKELS
023A  0B060C00      OR     AX,PENNIES
023E  7403          JZ     $+5H
0240  E90F00        JMP    @10
                                                               ; STATEMENT # 48
0243  B84800        MOV    AX,OFFSET(@@LONG$CONSTANT$0048H)
0246  1E            PUSH   DS                                  ; 1
0247  50            PUSH   AX                                  ; 2
0248  B80900        MOV    AX,9H
024B  50            PUSH   AX                                  ; 3
024C  E8BAFE        CALL   WRITE
024F  E96E00        JMP    @11
                                                               ; STATEMENT # 49
                @10:
                                                               ; STATEMENT # 50
0252  B85100        MOV    AX,OFFSET(@@LONG$CONSTANT$0051H)
0255  1E            PUSH   DS                                  ; 1
0256  50            PUSH   AX                                  ; 2
0257  B80900        MOV    AX,9H
025A  50            PUSH   AX                                  ; 3
025B  E8ABFE        CALL   WRITE
                                                               ; STATEMENT # 51
025E  B80000        MOV    AX,OFFSET(LF_CR)
0261  1E            PUSH   DS                                  ; 1
0262  50            PUSH   AX                                  ; 2
0263  B80200        MOV    AX,2H
```

```
0266  50              PUSH  AX                  ; 3
0267  E89FFE          CALL  WRITE
                                                ; STATEMENT # 52
026A  B81300          MOV   AX,OFFSET(S_TEXT)
026D  1E              PUSH  DS                  ; 1
026E  50              PUSH  AX                  ; 2
026F  FF360400        PUSH  DOLLARS             ; 3
0273  E84FFF          CALL  PRINT_COIN
                                                ; STATEMENT # 53
0276  B81C00          MOV   AX,OFFSET(Q_TEXT)
0279  1E              PUSH  DS                  ; 1
027A  50              PUSH  AX                  ; 2
027B  FF360600        PUSH  QUARTERS            ; 3
027F  E843FF          CALL  PRINT_COIN
                                                ; STATEMENT # 54
0282  B82F00          MOV   AX,OFFSET(D_TEXT)
0285  1E              PUSH  DS                  ; 1
0286  50              PUSH  AX                  ; 2
0287  FF360A00        PUSH  DIMES               ; 3
028B  E837FF          CALL  PRINT_COIN
                                                ; STATEMENT # 55
028E  B82600          MOV   AX,OFFSET(N_TEXT)
0291  1E              PUSH  DS                  ; 1
0292  50              PUSH  AX                  ; 2
0293  FF360800        PUSH  NICKELS             ; 3
0297  E82BFF          CALL  PRINT_COIN
                                                ; STATEMENT # 56
029A  813E0C000100    CMP   PENNIES,1H
02A0  7403            JZ    $+5H
02A2  E90F00          JMP   @12
                                                ; STATEMENT # 57
02A5  B83F00          MOV   AX,OFFSET(P1_TEXT)
02A8  1E              PUSH  DS                  ; 1
02A9  50              PUSH  AX                  ; 2
02AA  FF360C00        PUSH  PENNIES             ; 3
02AE  E814FF          CALL  PRINT_COIN
02B1  E90C00          JMP   @13
                                                ; STATEMENT # 58
                @12:
02B4  B83600          MOV   AX,OFFSET(P_TEXT)
02B7  1E              PUSH  DS                  ; 1
02B8  50              PUSH  AX                  ; 2
02B9  FF360C00        PUSH  PENNIES             ; 3
02BD  E805FF          CALL  PRINT_COIN
```

```
                                                                    ; STATEMENT # 59
                    @13:
                                                                    ; STATEMENT # 60
                    @11:
02C0  5D                POP     BP
02C1  C3                RET
                    PAYMENT         ENDP
                                                                    ; STATEMENT # 61
                    GET_INPUT       PROC NEAR
02C2  55                PUSH    BP
02C3  8BEC              MOV     BP,SP
                                                                    ; STATEMENT # 63
02C5  C45E04            LES     BX,[BP].TEXT_PTR
02C8  8D4701            LEA     AX,TEXT[BX+1H]
02CB  06                PUSH    ES                                 ; 1
02CC  50                PUSH    AX                                 ; 2
02CD  C45E04            LES     BX,[BP].TEXT_PTR
02D0  268A07            MOV     AL,ES:TEXT[BX]
02D3  B400              MOV     AH,0H
02D5  50                PUSH    AX                                 ; 3
02D6  E830FE            CALL    WRITE
                                                                    ; STATEMENT # 64
02D9  C7061C000000      MOV     VALUE,0H
                                                                    ; STATEMENT # 65
02DF  C6062700FF        MOV     NOT_DONE,0FFH
                                                                    ; STATEMENT # 66
                    @14:
02E4  8A062700          MOV     AL,NOT_DONE
02E8  D0D8              RCR     AL,1
02EA  7203              JB      $+5H
02EC  E94E00            JMP     @15
                                                                    ; STATEMENT # 67
02EF  E8E6FD            CALL    READ
02F2  88062600          MOV     CHAR,AL
                                                                    ; STATEMENT # 68
02F6  803E260030        CMP     CHAR,30H
02FB  B0FF              MOV     AL,0FFH
02FD  7301              JAE     $+3H
02FF  40                INC     AX
0300  803E260039        CMP     CHAR,39H
0305  B1FF              MOV     CL,0FFH
0307  7601              JBE     $+3H
0309  41                INC     CX
030A  22C1              AND     AL,CL
030C  D0D8              RCR     AL,1
030E  7203              JB      $+5H
0310  E92200            JMP     @16
```

```
                                                                    ; STATEMENT # 70
0313  8B061C00        MOV     AX,VALUE
0317  B90A00          MOV     CX,0AH
031A  F7E1            MUL     CX
031C  89061C00        MOV     VALUE,AX
                                                                    ; STATEMENT # 71
0320  8A062600        MOV     AL,CHAR
0324  B400            MOV     AH,0H
0326  03061C00        ADD     AX,VALUE
032A  81E83000        SUB     AX,30H
032E  89061C00        MOV     VALUE,AX
0332  E90500          JMP     @17
                                                                    ; STATEMENT # 73
                    @16:
0335  C606270000      MOV     NOT_DONE,0H
                                                                    ; STATEMENT # 74
                    @17:
033A  E9A7FF          JMP     @14
                    @15:
                                                                    ; STATEMENT # 75
033D  8B061C00        MOV     AX,VALUE
0341  5D              POP     BP
0342  C20400          RET     4H
                                                                    ; STATEMENT # 76
                    GET_INPUT       ENDP
                                                                    ; STATEMENT # 77
0000  8BEC            MOV     BP,SP
0002  FB              STI
                    BEGIN:
                                                                    ; STATEMENT # 79
0003  B80000          MOV     AX,OFFSET(LF_CR)
0006  1E              PUSH    DS                                    ; 1
0007  50              PUSH    AX                                    ; 2
0008  B80200          MOV     AX,2H
000B  50              PUSH    AX                                    ; 3
000C  E8FA00          CALL    WRITE
                                                                    ; STATEMENT # 80
000F  B80200          MOV     AX,OFFSET(PAID_TEXT)
0012  1E              PUSH    DS                                    ; 1
0013  50              PUSH    AX                                    ; 2
0014  E8AB02          CALL    GET_INPUT
0017  89060E00        MOV     PAID,AX
                                                                    ; STATEMENT # 81
001B  B80A00          MOV     AX,OFFSET(PURCHASED_TEXT)
001E  1E              PUSH    DS                                    ; 1
001F  50              PUSH    AX                                    ; 2
```

ASSEMBLY LISTING OF OBJECT CODE (continued)

```
0020  E89F02      CALL   GET_INPUT
0023  89061000    MOV    PURCHASED,AX
                                                    ; STATEMENT # 82
0027  B80000      MOV    AX,OFFSET(LF_CR)
002A  1E          PUSH   DS                          ; 1
002B  50          PUSH   AX                          ; 2
002C  B80200      MOV    AX,2H
002F  50          PUSH   AX                          ; 3
0030  E8D600      CALL   WRITE
                                                    ; STATEMENT # 83
0033  8B060E00    MOV    AX,PAID
0037  2B061000    SUB    AX,PURCHASED
003B  89060200    MOV    CHANGE,AX
                                                    ; STATEMENT # 84
003F  8B060200    MOV    AX,CHANGE
0043  B96400      MOV    CX,64H
0046  31D2        XOR    DX,DX
0048  F7F1        DIV    CX
004A  89060400    MOV    DOLLARS,AX
                                                    ; STATEMENT # 85
004E  8B060200    MOV    AX,CHANGE
0052  B96400      MOV    CX,64H
0055  31D2        XOR    DX,DX
0057  F7F1        DIV    CX
0059  89160000    MOV    COINS,DX
                                                    ; STATEMENT # 86
005D  8B060000    MOV    AX,COINS
0061  B91900      MOV    CX,19H
0064  31D2        XOR    DX,DX
0066  F7F1        DIV    CX
0068  89060600    MOV    QUARTERS,AX
                                                    ; STATEMENT # 87
006C  8B060000    MOV    AX,COINS
0070  B91900      MOV    CX,19H

0073  31D2        XOR    DX,DX
0075  F7F1        DIV    CX
0077  89160000    MOV    COINS,DX
                                                    ; STATEMENT # 88
007B  8B060000    MOV    AX,COINS
007F  B90A00      MOV    CX,0AH
0082  31D2        XOR    DX,DX
0084  F7F1        DIV    CX
0086  89060A00    MOV    DIMES,AX
                                                    ; STATEMENT # 89
008A  8B060000    MOV    AX,COINS
008E  B90A00      MOV    CX,0AH
```

ASSEMBLY LISTING OF OBJECT CODE (continued)

```
0091  31D2          XOR     DX,DX
0093  F7F1          DIV     CX
0095  89160000      MOV     COINS,DX
                                                    ; STATEMENT # 90
0099  8B060000      MOV     AX,COINS
009D  B90500        MOV     CX,5H
00A0  31D2          XOR     DX,DX
00A2  F7F1          DIV     CX
00A4  89060800      MOV     NICKELS,AX
                                                    ; STATEMENT # 91
00A8  8B060000      MOV     AX,COINS
00AC  B90500        MOV     CX,5H
00AF  31D2          XOR     DX,DX
00B1  F7F1          DIV     CX
00B3  89160C00      MOV     PENNIES,DX
                                                    ; STATEMENT # 92
00B7  8B060E00      MOV     AX,PAID
00BB  3B061000      CMP     AX,PURCHASED
00BF  7203          JB      $+5H
00C1  E90F00        JMP     @18
                                                    ; STATEMENT # 93
00C4  B85A00        MOV     AX,OFFSET(@@LONG$CONSTANT$005AH)
00C7  1E            PUSH    DS                      ; 1
00C8  50            PUSH    AX                      ; 2
00C9  B80C00        MOV     AX,0CH
00CC  50            PUSH    AX                      ; 3
00CD  E83900        CALL    WRITE
00D0  E90300        JMP     @19
                                                    ; STATEMENT # 94
                    @18:
00D3  E85101        CALL    PAYMENT
                                                    ; STATEMENT # 95
                    @19:
                                                    ; STATEMENT # 96
00D6  FB            STI
00D7  F4            HLT
                                                    ; STATEMENT # 97
```

Sample Programs in C, FORTRAN, and Pascal

On the tutorial disk are source code files for change maker programs written in C (CMAKR.C), FORTRAN (CMAKR.FOR), and Pascal (CMAKR.PAS). Though similar to the PL/M change maker program used in the I²ICE tutorial, the C, FORTRAN, and Pascal programs are provided only as examples and cannot be used with this tutorial.

Before emulating these programs with the system, they must be compiled, linked, located, and memory mapped in the I²ICE system. You will also have to add I/O routines if you want to simulate user interaction.

A Change Maker Program in C

To debug a C program using the I²ICE system, use C86 V2.0 or higher. Load the absolute code and then type, "go til :*main module name*" so that all symbolic information becomes available. Then, because the C compiler adds an underscore to the tail of every symbolic name, when entering a symbol name, use the underscore (i.e., *symbolname_*).

```
/*      The Changemaker Program
 *
 *      This program is written in C-86.
 *
 *    The program is designed to function as a simple change maker.
 *      Amount paid and price are part of the program. The program
 *      calculates change distribution.
 */

main( )

{
 int dollars, quarters, dimes, nickels, pennies, remainder;
 float amt_paid, price, total_change;
 int response = 'y';

{
            /* INITIALIZE PRICE AND AMT TENDERED */

        price = 9.49;
        amt_paid = 10.00;
        total_change = amt_paid - price;

            /* FIGURE OUT COIN DISTRIBUTION */

            remainder = total_change * 100;

            dollars = remainder/100;
            remainder = remainder % 100;

            quarters = remainder / 25;
            remainder = remainder % 25;
```

```
                dimes = remainder / 10;
                remainder = remainder % 10;

                nickels = remainder / 5;
                remainder = remainder % 5;

                pennies = remainder;
    }
} /* END */
```

A Change Maker Program in FORTRAN

```
*       The Changemaker Program
*
*   This program assumes an amount paid for an item
*   of an assumed value and computes the change due and
*   how to make that change in U.S. currency.
*
      program CMAKER

      real*4 price, amt_paid, change, coins
      integer*4 dollars, quarters, dimes, nickels, pennies

*******       CALCULATE CHANGE

    price = 46.33
    amount_paid = 50.00
    change = amt_paid - price

*******       FIGURE BILL AND COIN DISTRIBUTION

      change = change * 100.0
      dollars = change / 100
      coins = MOD (change, 100.0)
      quarters = coins / 25
      coins = MOD (coins / 25.0)
      dimes = coins / 10
      coins = MOD (coins, 10.0)
      nickels = coins / 5
      coins = MOD (coins, 5.0)

*******       CORRECTION FACTOR FOR REAL NUMBER ANOMALIES

      coins = coins + 0.1
      pennies = coins

      stop
      end
```

A Change Maker Program in Pascal

```
PROGRAM cmaker (INPUT, OUTPUT);

(*   This Pascal program is non-interactive. It contains a purchase price   *)
(*   and an amount paid, and puts the change in the memory location of the   *)
(*   variables.                                                              *)

VAR
   purchase  :  INTEGER;
   paid      :  INTEGER;
   change    :  INTEGER;
   coins     :  INTEGER;
   dollars   :  INTEGER;
   quarters  :  INTEGER;
   dimes     :  INTEGER;
   nickels   :  INTEGER;
   pennies   :  INTEGER;

PROCEDURE init (* variables will be global *);

   BEGIN
      purchase  :=  0;
      paid      :=  0;
      change    :=  0;
      coins     :=  0;
      dollars   :=  0;
      quarters  :=  0;
      dimes     :=  0;
      nickels   :=  0;
      pennies   :=  0
   END;

PROCEDURE getinput (* purchase, paid *);

BEGIN
   paid := 150;
   purchase := 108
END;
```

```
PROCEDURE payout; (* how many dollars, quarters, etc. *)

   BEGIN
      dollars   :=  change  DIV   100;
      coins     :=  change  MOD   100;
      quarters  :=  coins   DIV    25;
      coins     :=  coins   MOD    25;
      dimes     :=  coins   DIV    10;
      coins     :=  coins   MOD    10;
      nickels   :=  coins   DIV     5;
      pennies   :=  coins   MOD     5
   END;

BEGIN (* mainline *)
   init (* clear memory *);
   getinput (* user interaction *);
   change := paid - purchase;
   payout (* put amount in memory *)
END.
```

3 INTRODUCTION TO USING THE I²ICE™ SYSTEM

In the installation appendix for your host software, you are encouraged to install the I²ICE tutorial software so that you can quickly learn I²ICE commands and features. The information in this chapter provides more detail on many of the topics covered in the tutorial.

The main sections of the chapter are the following:

- Invoking the I²ICE Software
- Entering I²ICE System Commands. This section describes command entry, the command line editor, the syntax menu, the command history buffer, string handling, and block commands.
- Creating Debug Objects. This section describes how to create PROCs, LITERALLY definitions, and debug registers.
- The I²ICE Screen Editor
- File Handling. This section explains the file handling commands LIST, INCLUDE, PUT, APPEND, and SAVE.
- Memory Types
- Managing the Memory and I/O Spaces
- The Emulation Clips
- Emulating a Program
- Breaking, Tracing, and Arming
- Hardware Slipping on a Breakpoint
- Even Addresses, Odd Addresses, and Breaking
- Moving the User Cable

Invoking the I²ICE™ Software

The host development system can be an Intellec Series III, a Model 800 upgraded to a Series III, an Intellec Series IV, or an IBM PC/XT or PC/AT. The Intellec Series III and the Model 800 use the ISIS operating system, the Series IV uses the iNDX operating system, and the IBM PC hosts use the DOS operating system PC DOS (version 3.0 or greater).

NOTE

Version 1.8 of the I²ICE software requires that your host development system be configured with at least 512K bytes of mass memory.

- Series III

  Invoke the I²ICE software by entering the following command (which assumes that the I²ICE software is on drive 0):

  ```
  -RUN I2ICE
  ```

- Series IV

  Invoke the I²ICE software by entering its filename, as follows.

  ```
  >I2ICE.86
  ```

- IBM PC/XT or IBM PC/AT

  Invoke the I²ICE software by entering the following command. (The prompt shown in the example assumes that you have set your PC prompt using the command PROMPT = PG. It is further assumed that the I²ICE software is installed on the hard disk in the directory ICEDIR, as recommended in the installation appendix for the IBM PC hosts.)

  ```
  C:\ICEDIR>I2ICE
  ```

If your system disk contains a file called I2ICE.CRT, the I²ICE system obtains the character definitions for the system console from this file. If your system disk does not contain a file called I2ICE.CRT, the I²ICE system assumes an Intel or IBM PC terminal.

If your system disk contains a macro file called I2ICE.MAC, the I²ICE system executes the I²ICE commands in that file upon invocation. (The I²ICE tutorial software includes an I2ICE.MAC file that controls tutorial file loading.) The following example is a typical I2ICE.MAC file:

```
BASE = 16T                                  /* set default number base */
DEFINE LITERALLY l = 'literally'            /* define LITERALLYs */
DEFINE l def = 'define'
def l g = 'go'
def l len = 'length'
```

If this macro file is present when you invoke the I²ICE software, the default radix is set to hexadecimal, and some LITERALLYs are predefined.

If your system disk contains CRT and MAC files, but you do not want to use them, you must specifically exclude them in the invocation command, as follows (for a Series III standalone host):

```
-RUN I2ICE NOCRT NOMACRO
```

If you rename any of the I²ICE files and you want them to be used as the default files during invocation, you must rename them all. For example, if you want MYFILE for a name, you must change I2ICE.CRT to MYFILE.CRT, I2ICE.MAC to MYFILE.MAC, I2ICE.OVE to MYFILE.OVE, I2ICE.OVH to MYFILE.OVH, and the probe file I2ICE.086 (for the 86/88 probe) to MYFILE.086. Then you would invoke I²ICE software with the following command:

```
-RUN MYFILE
```

For more information on the I2ICE command, see the I2ICE entry in the *I²ICE™ System Reference Manual*.

Entering I²ICE™ System Commands

The following subsections describe command entry.

Extending a Command to Another Line

Commands that exceed 80 characters continue invisibly. The 80th character displays as an exclamation point (!). To extend a command to another line, enter an ampersand (&) followed by a carriage return and complete the command on the next line. Text that appears between the ampersand and the carriage return is interpreted as a comment.

The following example shows a command that extends over more than one line. Note that the ampersand causes the next line to have a double prompt (**).

```
*DEFINE SYSREG atquarters = WRITE AT .:cmaker.quarters &
**CALL OUT1
```

The ampersand is necessary in the previous example because CALL OUT1 is an optional clause.

You may omit the ampersand if the command needs more syntactic information to be complete. If you enter a carriage return and the command is incomplete, the I²ICE system prompts for more information. The following example shows an incomplete command and the I²ICE prompt for more information.

```
*GO FROM begin TIL BOTH (:cmaker.payment) AND (WRITE AT
**.:cmaker.quarters)
```

Aborting Commands

Abort a command by entering CTRL-C (for IBM PC hosts, use CTRL-Break). Note that the CTRL-C has no effect on emulation. Emulation continues until a breakpoint is reached or until you enter a HALT command.

Multiple Commands On a Line

To enter more than one command on a line, separate each command with a semicolon (;). The following example shows two commands on the same line.

```
*MAP 0 LENGTH 32K HS;MAPIO 0 LENGTH 64T ICE
```

Comments

Enclose a comment within a slash-asterisk combination. The symbol /* begins the comment, and the symbol */ ends the comment. The following example illustrates a comment.

```
*GO USING qwrite                    /* Emulate until quarters is modified */
```

The Command Line Editor

The command line editor controls various keys that enable you to edit command lines.

Use the right and left arrow keys to move the cursor to the desired location within the command line before entering the correction. Pressing the HOME key after pressing the left arrow key moves the cursor to the beginning of the line. Pressing the HOME key after pressing the right arrow key moves the cursor to the end of the line. The up arrow key restores the previous line from the history buffer for editing. The down arrow key moves to the next line in the history buffer.

Use the RUBOUT key to delete the character to the left of the cursor. (For the IBM PC hosts, use the left arrow key at the top of the keyboard—above the <Enter> key—to perform the rubout function.) To delete other characters on the command line, use the following control characters:

CTRL-A Deletes the line to the right of the cursor, including the character at the cursor position.

CTRL-F Deletes the character at the cursor position; adjusts line spacing.

CTRL-X Deletes the line to the left of the cursor, including the character at the cursor position, and adjusts line spacing.

CTRL-Z Deletes the entire line.

For more information on the command line editor, see the Editors entry in the *I²ICE™ System Reference Manual*. Note that you can also edit commands using the I²ICE screen editor.

The I²ICE™ System Syntax Menu

The I²ICE syntax menu lists your options when you enter I²ICE commands. If you follow its choices, you cannot construct a syntactically incorrect I²ICE command, although it may be semantically incorrect.

When you invoke the I²ICE software, the first line of the menu appears on the bottom of the screen. Call up subsequent lines by pressing the TAB key. The menu is circular in one direction. Press TAB enough times, and you come back to where you started. You cannot reverse the menu.

Each line contains a list of choices. The keywords are shown in uppercase letters. The menu also contains lowercase entries enclosed in angle brackets. These represent user-defined names or a set of I²ICE keywords. For example, <variable> represents the name of a debug variable or a program variable; <mtype> represents one of the keywords identifying an I²ICE memory type, such as INTEGER or REAL.

Your choice need not appear on the screen when you enter it. If you enter a space after you enter your choice, the menu displays the next level. As shown in the following example, when you enter the keyword DEFINE and follow it with a space, the new menu displayed is

```
---- more ----
GLOBAL BRKREG TRCREG EVTREG ARMREG SYSREG PROC LITERALLY
```

Press the TAB key to see the rest of the menu.

```
---- more ----
<mtype>
```

Pressing the TAB key again returns you to the first line.

You can return to higher menu levels. If, after you entered the space, you press the RUBOUT key or the left-arrow key, the menu returns to the top level, the one from which you chose DEFINE.

The I²ICE menu recognizes LITERALLY definitions. If you have a LITERALLY name called def representing DEFINE, entering def followed by a space displays the DEFINE menu and also automatically expands the LITERALLY name.

You can disable the menu display with the MENU command. (Note that disabling the menu display also disables automatic LITERALLY expansion.) The following command turns off the menu display:

```
*MENU = 0
```

You can also switch the menu display (and the automatic LITERALLY expansion feature) on and off by entering CTRL-V.

The following command turns on the menu display:

```
*MENU = 1
```

The I²ICE™ Command History Buffer

The I²ICE system stores commands in a 400-character, last-in first-out buffer. Using the history buffer, you do not need to enter a previous command to re-execute it or change it. Scroll through the buffer by pressing the up arrow key until you reach the command you want, and

use the line-editing functions to modify the command. Entering a carriage return (or, for IBM PC hosts, using the <Enter> key) executes that command. The new version becomes the latest entry in the command buffer. The old version is still in its original place in the buffer.

String Handling

A string has memory type CHAR. Use the DEFINE command to define a string. For example, define the string called A as the character 5.

```
*DEFINE CHAR A = '5'
```

To add 1 to the string A, use the string name in an expression. The following example assumes a hexadecimal number base.

```
*A+1
36
```

The answer is 36 because the ASCII value of 5 is 35 (hexadecimal). This example illustrates memory type conversion. The string A is memory type CHAR, and the constant 1 is memory type DWORD. The answer (the constant 36) has memory type DWORD.

The CONCAT function concatenates strings. The following example defines a string called B and concatenates it with string A.

```
*DEFINE CHAR B = '12'
*CONCAT(A,B)
512
```

The NUMTOSTR function temporarily treats a number as a string. The following example concatenates the string B with the string derived from the debug variable four.

```
*DEFINE BYTE four = 4
*CONCAT(NUMTOSTR(four),B)
412
```

The SUBSTR function picks out a portion of a string. The following debug procedure steps through a user program and displays all the MOV instructions. When a memory location is disassembled, the opcode field is a four-character field starting at position 20H.

```
*REPEAT
. *IF SUBSTR(ASM $,20,4) = = 'MOV ' THEN
. . *ASM $
. . *END
. *ISTEP
. *UNTIL $ = = :cmaker#14
. *END
0021:0045H      B90E0E00        MOV WORD PTR 000EH,CX
:CMAKER#13
0021:0049H      8BC1            MOV AX,CX
0021:004CH      BE6400          MOV SI,0064H ; +100T
0021:0057H      89161000        MOV WORD PTR 0010H,DX
```

The INSTR function searches a string for a substring and returns the index on which the substring begins. As shown in the following example, the index is always in decimal.

```
*INSTR('abcdefghijklmn','klm')
11
```

For more information on string commands, see the string command entries in the *I²ICE™ System Reference Manual*.

Block Commands

A block command begins with one of the following keywords:

COUNT
DO
IF
REPEAT

A block command contains one or more I²ICE commands and terminates with END. All I²ICE commands except EDIT, INCLUDE, HELP, and LOAD can be included within a block command.

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

A period (.) before the asterisk prompt indicates that the I²ICE system recognizes the beginning of a block command and has not yet detected the end. 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.

```
*COUNT 5
. *ISTEP
. *EVAL $ LINE
. *END
```

The I²ICE system executes a block command when you press the carriage return after the END of the outermost block.

Creating Debug Objects

Debug objects are uniquely-named, user-created software constructs that the I²ICE system uses to manage the debugging environment. The four types of debug objects are debug procedures, LITERALLY definitions, debug registers, and debug variables.

Debug procedures	are user-named groups of I²ICE commands. Execute them just as you would an I²ICE command.
LITERALLY definitions	enable you to substitute shorthand names for previously-defined character strings. LITERALLYs save keystrokes.
Debug registers	are user-named software registers that hold arm, break, system, and trace specifications.
Debug variables	are user-defined variables used with I²ICE commands. While program variables are stored in program memory, debug variables are stored in I²ICE system memory.

Creating a Debug Procedure

The following example uses the DEFINE command to define a debug procedure. The I²ICE commands are enclosed within a DO-END block.

```
*DEFINE PROC in-paid = DO
. *PORTDATA = 100T
. *END
```

The keyword PROC identifies the definition as a debug procedure. The I²ICE commands that make up the debug procedure must be enclosed within a DO-END block. When you follow the DO with a carriage return, the I²ICE system returns a prompt that represents the command nesting level. The single period signifies that the I²ICE system is waiting for only one END.

The following example shows more than one nesting level.

```
*DEFINE PROC money = DO
. *IF %0 = =2 THEN
. . *PORTDATA = 100T
. . *ELSE IF %0 = =4 THEN
. . . *PORTDATA = 75T
. . . *END
. . *END
. *END
*
```

Creating a LITERALLY Definition

The following example uses the DEFINE command to create a LITERALLY definition.

```
*DEFINE LITERALLY def = 'DEFINE'
```

Now def can be used in place of DEFINE. The character string to the right of the equal sign can be up to 254 characters long.

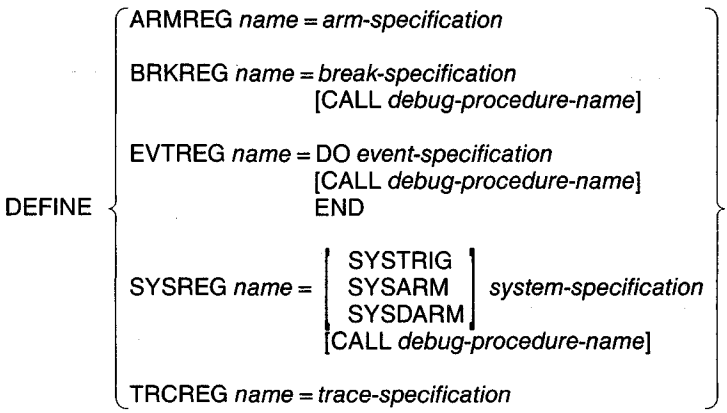
You can also use a LITERALLY definition to replace a command line as shown in the following example.

```
*def LITERALLY mp = 'MAP 0 LENGTH 32K HS;MAPIO 0 LENGTH 64T ICE;MAP;MAPIO'
```

When you enter mp followed by a carriage return, the I²ICE system executes all the commands in the character string. If you store this LITERALLY definition in the I2ICE.MAC file, it is executed immediately after invocation. Note that this LITERALLY definition takes advantage of the previous example by using def instead of DEFINE. In your I2ICE.MAC file, the LITERALLY definition of def must precede that of mp. This example also shows that multiple commands on the same line must be separated by semicolons (;).

Creating a Debug Register

Use the following syntax to define arm, break, event, system, and trace registers. The ARMREG, BRKREG, EVTREG, SYSREG, and TRCREG entries in the *I²ICE™ System Reference Manual* describe these registers in detail.



Creating a Debug Variable

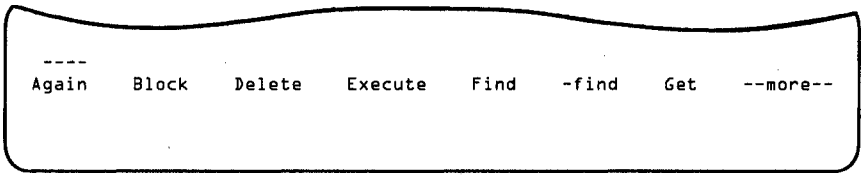
The syntax for defining a debug variable is as follows:

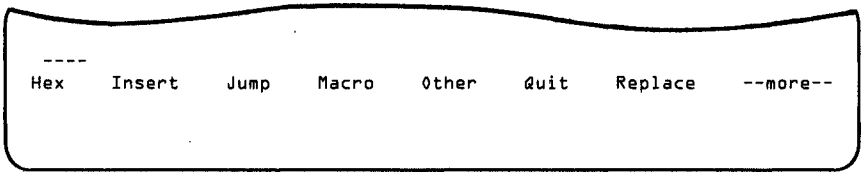
DEFINE *mtype name* [= *value*]

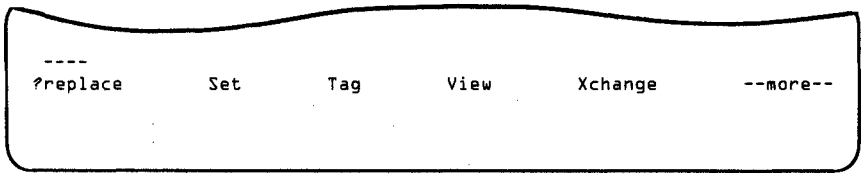
An mtype is one of the I²ICE memory types. (See the memory types section in this chapter for more information on memory types and on creating debug variables.) If you do not set the debug variable equal to a value, the I²ICE system assumes zero.

The I²ICE™ Screen Editor

The screen editor has all the features of the AEDIT V1.0 editor. It is menu-driven and, when invoked (with the EDIT command or the ESC key), displays the edit menu at the bottom of the screen. The main menu is displayed on three screens; press the TAB key to advance to the next screen. The following screens show the main menu prompt lines.







Choose a menu item by pressing the key representing its first letter. Several of the screen-editor commands prompt for additional information or display sub-menus. The following sections describe some of the I²ICE system screen editor commands. The AEDIT manual (order number 121756) describes all the screen editor commands in detail and gives examples.

Inserting Text

To begin inserting text when you are in the screen editor, position the cursor, then press the I key. The menu prompts

```
[insert]
```

What you enter is inserted into the buffer at the cursor position. Return to the main menu by pressing the ESC key or by entering CTRL-C. (Note that CTRL-C deletes all the text you inserted.)

Deleting and Moving Text

The I²ICE screen editor uses the same control characters as the line editor. To delete a character or line, use the CTRL-A, CTRL-F, CTRL-X, or CTRL-Z key.

To delete or move a block of text, use the screen editor temporary buffer. First delimit the text that you want loaded into the temporary buffer by moving the cursor to the start of the block and pressing the B key. Pressing the B key sets the first delimiter for the temporary buffer and displays the buffer menu as shown in the following example:

```
Buffer Delete Find -find Jump Put
```

The first delimited character appears as an at sign (@).

Pressing the D key deletes the delimited text from the screen and copies it into the temporary buffer. Pressing the B key copies the delimited text into the buffer without deleting it from the screen. The system editor then returns you to the main menu.

To move the deleted or copied text elsewhere in the edited text, first move the cursor to the desired position. Then press G (for Get) and either the ESC key or the RETURN (or, for IBM PC hosts, Enter) key.

Viewing Text

The View command is useful when the file you are editing is longer than one screen. Pressing the V key rewrites the screen display with the line containing the cursor in the middle of the screen (unless the cursor is so near the beginning or end of the text that the line cannot be centered).

Overwriting Text

To begin overwriting text, position the cursor and press the X key. The bottom line now reads as follows:

```
[exchange]
```

When you enter a character, it replaces the character at the cursor position, and the cursor moves to the next position. Return to the main menu by pressing the ESC key or by entering CTRL-C. (Note that CTRL-C deletes all the changes you made.)

Editing External Files

With the screen editor you can edit development system files without returning to the I²ICE command line. The Editors entry in the *I²ICE™ System Reference Manual* describes external file editing in detail.

Exiting the Screen Editor

To exit the screen editor, return to the main menu and press the Q key. If you were editing a debug procedure, the bottom line displays the following quit menu:

```
Abort      Execute      Init      Write
```

If you were editing an external file, the bottom line displays the following quit menu:

```
Abort    Execute    eXit    Init    Update    Write
```

— The Abort sub-command returns to the I²ICE command line, and all changes (if any) are lost.

— The Execute sub-command returns to the I²ICE command line and executes the edited command or debug procedure.

— The eXit sub-command incorporates all changes and returns to the I²ICE command line.

— The Init sub-command enables you to start another editing session without returning to the I²ICE command line.

— The Update sub-command incorporates all changes without returning to the I²ICE command line.

— The Write sub-command prompts for an output file without returning to the I²ICE command line.

For an on-line demonstration of the command and screen editors, see the corresponding modules in the I²ICE tutorial.

File Handling

The next three subsections describe list files, include files, and the LOAD and SAVE commands.

List Files

List files record the console interactions of a debug session. The LIST command syntax is as follows:

LIST *pathname*

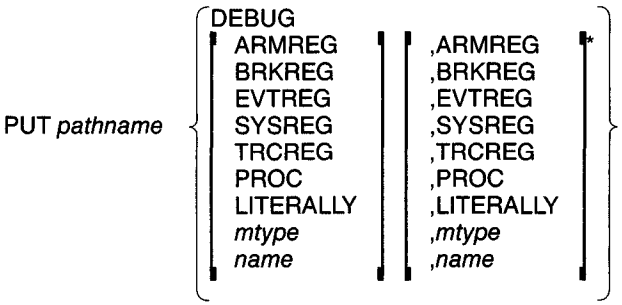
A typical *pathname* for a standalone Series III host is :F1:lst.001. This represents a file on disk drive 1 whose file name is lst.001. (See the Pathname entry in the *I²ICE™ System Reference Manual* for more information on *pathname*.) After you enter the LIST command, all console

activity is written to that file. You can stop recording in the list file by entering the NOLIST command. You can restart listing, but if you use the same pathname, you are prompted as follows: Overwrite existing file? (y or [n]). If you do not want to overwrite the existing list file, answer "n" and re-enter the LIST command using another file name for the new list file.

Include Files: The INCLUDE, PUT, and APPEND Commands

Include files are text files that contain I²ICE commands. You can construct them with a text editor, or you can use the PUT and APPEND commands.

The PUT command creates an include file. If the file already exists, you are prompted with the following message: Overwrite existing file? (y or [n]). The syntax for the PUT command is as follows:



The keyword DEBUG writes the definitions of all currently defined debug objects to the specified include file. You can also write all debug registers of the specified type, all debug procedures, all LITERALLY definitions, or all debug variables of the specified memory type. In addition, you can specify only the debug objects you want saved.

The following command saves the definitions of all debug variables of type BYTE, the debug objects named even and odd, and all debug procedures to the current directory on an IBM PC host.

```
*PUT deb.001 BYTE, even, odd, PROC
```

When you enter the PUT command and if the file name already exists on the disk, you are asked by the I²ICE system whether you want to overwrite an existing file. If you do not want to overwrite the existing file, change the file name. You can add to the include file rather than writing over it by using the APPEND command. The syntax is the same as that for PUT.

To retrieve the debug object definitions stored in the include file, use the INCLUDE command. For example, to include the standalone Series III file :F1:deb.001:, enter the following:

```
*INCLUDE :F1:deb.001
```

After you enter the INCLUDE command, the I²ICE system displays the contents of the specified include file on the console screen. You can suppress that display with the NOLIST option, as in the following standalone Series III example:

```
*INCLUDE :F1:deb.001 NOLIST
```

The LOAD and SAVE Commands

Program files must contain absolute code. All memory references must be resolved. The I²ICE system will not accept load-time locatable files. Construct program files by compiling (or assembling) your source code (use the DEBUG option), linking the compiled (or assembled) file, and locating the linked file.

The following syntax is for the LOAD command:

LOAD *pathname* [NOCODE] [NOSYMBOLS] [NOLINES] [APPEND]

For example, if you are using a Series III host system to load the program file called cmaker.86 from disk drive 1 into program memory, enter the following:

```
*LOAD :F1:cmaker.86
```

There must be enough unguarded program memory to contain the program file. These memory locations must be mapped to the physical locations expected by the program file. (See the memory mapping section of this chapter for information on mapping.)

Because the I²ICE system does not clear memory before each load, you can load multiple files by issuing successive LOAD commands. The user program symbol table resides in host memory, and it is cleared by the LOAD command. After successive loads, the I²ICE system only retains the symbol table from the last load. If you use the APPEND option, however, the symbol table retains program symbols from any previous loads.

The LOAD command can also be used to load files into memory that were created with the SAVE command. The SAVE command saves the contents of a specified memory partition to the file specified by a pathname. The memory image is saved in 8086 OMF format.

Use SAVE to save assembly-level patches for future debugging sessions or to save modified data table values that improve performance of the software being debugged.

Memory Types

A debug variable always has one of the following I²ICE memory types associated with it.

ADDRESS	16-bit unsigned number
ASM	Assembly language mnemonic, read-only
BCD	80-bit packed binary coded decimal number
BOOLEAN	8 bits, but only the least significant bit (LSB) has meaning (TRUE has LSB = 1; FALSE has LSB = 0)
BYTE	8-bit unsigned number
CHAR	8-bit ASCII value
DWORD	32-bit unsigned number
EXTINT	64-bit signed decimal number
INTEGER	16-bit signed decimal number
LONGINT	32-bit signed decimal number
LONGREAL	64-bit floating point decimal number
POINTER	For the 8086/8088 and 80186/80188 probes, 32-bit selector:offset pair For the 80286 probe in real address mode, 32-bit selector:offset pair For the 80286 probe in protected mode, 48-bit LDT-selector: segment-selector:offset triplet
REAL	32-bit floating point number
SELECTOR	16-bit unsigned number
SHORTINT	8-bit signed decimal number
TEMPREAL	80-bit floating point decimal number
WORD	16-bit unsigned number

The I²ICE system does not distinguish between ADDRESS, SELECTOR, and WORD. For more information on I²ICE memory types, see the Mtype entry in the *I²ICE™ System Reference Manual*.

A program variable always has the type defined by the user program. Use I²ICE memory types to determine how you read the program variable from memory.

Debug Variables

A debug variable is defined with an I²ICE command during a debugging session. With certain restrictions, you can assign debug variables of one type equal to debug variables of another type as shown in the following example:

```
*DEFINE WORD answer
*DEFINE BYTE ans = 4FH
*answer = ans
```

In this example, the I²ICE system pads the value in ans with leading zeros to complete the word occupied by answer.

The following example uses signed integers. The debug variable small is an 8-bit signed integer; its type is SHORTINT. The debug variable large is a 16-bit signed integer; its type is INTEGER.

```
*DEFINE INTEGER large
*DEFINE SHORTINT small = -4
*large = small
```

The I²ICE system sign-extends the value of small to complete the word occupied by large.

Some conversions are illegal. For example, you cannot set a debug variable of type CHAR (an 8-bit ASCII value) equal to any of the signed types.

```
*DEFINE CHAR yes
*DEFINE SHORTINT small = -4
*yes = small
ERROR #69
Invalid type conversion
```

Program Variables and Symbolic Debugging

When you first load a program, the execution point is at the beginning of the main module's prologue. The prologue clears interrupts, loads the stack segment register, the stack pointer, and the data segment register, then jumps to the beginning of your program. The first instruction of your program begins statement #1. The locater puts this prologue at the beginning of your code. Before you can access a program variable symbolically, you must execute the prologue and enter your program.

You can take a look at the prologue by applying the ASM memory template, as shown in the following example:

```
*LOAD :F1:cmaker.86
*ASM $ LENGTH 7T
0020:0006H   FA              CLI                          /*ISTEP*/
0020:0007H   2E8E160000      MOV SS,CS:WORD PTR 0000H     /*ISTEP*/
```

```
0020:000CH  BCA001          MOV SP,01A0H ; +416T
0020:000FH  2E8E1E0200      MOV DS,CS:WORD PTR 0002H    /*ISTEP*/
0020:0014H  EA0A002100      JMP (#1)0021H:000AH
0020:0019H  90          NOP
:CMAKER#1
0021:000AH  8BEC            MOV BP,SP
```

You can execute the next instruction (using the ISTEP command) as well as display it. Apply the ASM memory template to the execution point and enter the command ISTEP.

```
*ASM $;ISTEP
0020:0006H FA       CLI
```

You can then re-execute this command by entering a CTRL-E (hold down the CTRL key and press the E key). The screen displays the command again.

```
*ASM $; ISTEP
0020:0007H  2E8E160000      MOV SS,CS:WORD PTR 0000H
```

If you enter CTRL-E again, the screen displays the next instruction.

```
*ASM $; ISTEP
0020:000FH  2E8E1E0200      MOV DS,CS:WORD PTR 0002H
```

The second and third ISTEP commands each executed two instructions. The next instruction to be executed is MOV BP,SP because the 8086 architecture excutes two instructions if you single-step through an instruction that is a move into a segment register. No interrupt, not even a non-maskable interrupt, can occur between these two instructions. After you entered CTRL-E, you would see the next instruction.

```
*ASM $;ISTEP
:CMAKER#1
0021:000AH  8BEC        MOV BP,SP
```

Now your program has begun execution. To display a program variable named coinrelease, enter its name:

```
*:cmaker.coinrelease
+1
```

To modify *coinrelease*, set it to another value as in the following example:

```
*:cmaker.coinrelease = 0
*:cmaker.coinrelease
0
```

Note that cmaker.coinrelease is the fully-qualified reference to coinrelease. A fully-qualified reference to a symbol includes the module name and the names of all procedures that enclose the symbol from outer-most to inner-most. Because a fully-qualified reference completely

identifies the symbol, such a reference is always valid. A partially-qualified reference omits the module name and one or more of the outer procedure names. A partially-qualified reference is valid only if the current execution point is inside the outer-most procedure referenced.

Program Variables and the I²ICE™ Memory Types

The I²ICE memory types are templates for reading and writing program memory. You can read and write program memory by specifying just the program variable's symbolic name, which reads the program variable as its program-defined type.

If coinrelease is defined by the user program as a WORD, you can nevertheless read coinrelease as an integer. Apply the I²ICE memory template over program memory, as shown in the following example:

```
*INTEGER .coinrelease
003E:0004H          +1
```

The user program type for coinrelease would remain WORD.

You can also assign values with a memory template. The following example sets coinrelease to 0:

```
*INTEGER .coinrelease = 0
```

You can use a debug variable without the period. The debug variable is a value, and the I²ICE memory template command uses that value as the address. The debug variable's value rather than its address becomes the operand. The value of a POINTER variable named begin (which, it is assumed, has been set to the beginning program address) is 0020:0006H. Applying the WORD template to begin tells you that memory location 0020:0006H contains FA, and 0020:0007H contains 2E.

```
*WORD begin
0020:0006H    2EFA
```

Managing the Memory and I/O Spaces

This section explains how to set up the memory map and the I/O map and how to read and write memory locations.

The I²ICE™ Memory Map

The user program must be absolute code, and every memory reference must have a unique physical address. The I²ICE memory map determines where this memory space physically resides. You can split up the memory space among the following:

USER	The prototype hardware contains the memory.
HS	The I²ICE system contains the memory. It resides on the map-I/O board in the I²ICE instrumentation chassis. There are 32K bytes of high-speed (HS) memory available on the map-I/O board.
OHS	The I²ICE system contains the memory. It resides in optional high-speed (OHS) memory. Each OHS board provides 128K bytes of high-speed memory. You can install one or two OHS boards.
MB	The host development system contains the memory. It resides on memory boards within the host chassis. The MB stands for the MULTIBUS bus. [IBM PC hosts cannot use the MB option.]
GUARDED	The memory does not reside anywhere. An attempt to access guarded memory results in an error.

In addition to providing access to as much as 32K bytes of HS memory, 256K bytes of OHS memory, and all MB memory not used by the I²ICE system in the host development system, the I²ICE system also allows mapping I/O to the development system console or to an I²ICE debug procedure. The MB memory can be mapped in named or unnamed blocks. When named blocks are used, the same physical MB memory can be accessed as dual port memory by multiple probes in a multiprobe enivronment. (LOCK is supported.) In a single probe environment, the same memory can be mapped to multiple addresses in the same probe.

Both HS and OHS memory are zero-wait-state memory. This means that a user program does not take longer to access either HS or OHS memory than it does to access user memory. The I²ICE emulation is real-time. However, with the WAITSTATE command you can insert up to 15 wait-states into memory accesses. In this way, you can emulate slow memories.

In most designs today, hardware and software development take place in parallel. For example, if your software is ready for prototype memory, but the hardware is not yet available, use HS or OHS memory.

Running your prototype software completely within I²ICE memory may also help you distinguish between hardware and software problems. Later in the design, after you have verified your code and prototype memory becomes available, you can map your software to prototype memory, memory block by memory block.

You can view or set the memory map using the MAP command. All program memory is initially guarded, as shown in the following example.

```
*MAP
MAP OK LENGTH 1024K GUARDED
```

To change the memory map, specify a memory partition and a physical location for program memory. For example, to map the first 32K bytes of program memory to HS memory, enter the following command:

```
*MAP 0K LENGTH 32K HS
```

A partition is a range of addresses. The partition 0K LENGTH 32K represents a starting address of 0 and a range of blocks 32K bytes long. You can also represent a partition with starting and ending addresses, using the keyword TO instead of LENGTH. For example, you could map the first 32K bytes of program memory to HS memory by entering the following command:

```
*MAP 0K TO 32K-1 HS
```

The MAP command shows the result.

```
*MAP
MAP 0K LENGTH 32K HS
MAP 32K LENGTH 992K GUARDED
```

The MAP command also enables you to assign the two attributes READ and WRITE to memory partitions.

READ designates the partition as read-only. If you are simulating prototype ROM in HS RAM, you may want to designate that partition as read-only.

WRITE suppresses read-after-write verification. Read-after-write verification means that the I²ICE system follows a write to memory with a read from the same location and verifies the result. Normally, the I²ICE system performs a read-after-write verification when you load a user program and when you write memory from the console. You would not want a read-after-write verification if it would change the state of the I/O device.

NOTE

In one instance the I²ICE system reads after a write even when the partition is designated as WRITE. With the memory template commands (described later in this chapter) you can read and write program memory and interpret the data as one of the I²ICE memory types. When writing iterative data, the command repeats a data pattern over a memory partition. For example, when you enter the following command:

```
*BYTE .start LENGTH 7 = 1,2
```

After the command is executed, program memory values are the following:

Address	Value at that address
.start	1
.start + 1	2
.start + 2	1
.start + 3	2
.start + 4	1
.start + 5	2
.start + 6	1

The algorithmn used by the I²ICE system reads from program memory, even though you designated the partition as WRITE with the MAP command.

Regardless of the memory or I/O maps, if the BTHRDY pseudo-variable equals TRUE the target system must return a valid READY signal for each bus cycle (except HALT and SHUTDOWN) that the processor initiates. Otherwise, a time-out will occur if one or more of the time-out pseudo-variables are enabled (time-outs are enabled by default). The time-out pseudo-variables are the following:

- BUSACT allows a system time-out when the processor bus is inactive for more than one second.
- IORDY allows a system time-out when an I/O access takes more than one second.
- MEMRDY allows a system time-out when memory access time is longer than one second.
- PHANG allows a system time-out when coprocessor memory accesses exceed one second (8086/8088 and 80186/80188 probes only).

Erratic operation can occur if READY violates the setup or hold requirements or is asserted at the wrong time during a bus cycle. If the time-out pseudo-variables are disabled and the target system does not provide READY, the I²ICE HALT command will not return the probe to interrogation mode; return to interrogation mode by entering the RESET UNIT command.

Mapping Input/Output

The MAPIO command displays or changes the I²ICE I/O map. If you display the I/O map right after invoking I²ICE software, you will see all I/O ports mapped to USER, as shown in the following example:

```
*MAPIO
MAPIO 0 LENGTH 10000H USER
```

The I/O map determines the source of input data and the destination of output data. You can map I/O ports to USER or ICE. The prototype hardware reads or writes I/O ports mapped to USER.

All bus cycles (memory reads and writes, interrupt acknowledges, I/O reads and writes, and halt cycles) initiated by the processor in the I²ICE system are active in the target system

regardless of where memory or I/O is mapped. When memory or I/O is mapped to MB, OHS, or HS memory or ICE I/O, the data is written to both MB, OHS, or HS memory or ICE I/O and the target system. The data is read from both the target system and MB, OHS, or HS memory or ICE I/O. However, the data from MB, OHS, or HS memory or ICE I/O is used, and the data from the target system is ignored. This feature enables you to use the I²ICE system as a signal generator to debug hardware problems in target systems without relying on the data integrity of the hardware system.

The I²ICE system has 64K byte-wide ports or 32K word-wide ports available. You can only map I/O ports in blocks of 64 bytes. If you specify a partition that does not fall on a 64-byte boundary, the I²ICE system expands the partition to the next higher boundary.

For example, to map the first 64 ports to ICE, enter the following:

```
*MAPIO 0 LENGTH 64T ICE
```

The T specifies 64 as a decimal number (64T has the same value as 40H). Now look at the I/O map.

```
*MAPIO
MAPIO 00000H LENGTH 00040H ICE
MAPIO 00040H LENGTH 0FFC0H
```

You could also have specified the I/O partition with a starting and an ending address. The following example produces the same result:

```
*MAPIO 0 TO 64T-1 ICE
```

Simulating I/O from the Console

To simulate I/O from the console, map one or more I/O partitions to ICE. When the user program requests input, the console displays a message requesting an input. Respond by entering the input value at the console keyboard. For example, if the first I/O partition is mapped to ICE and the user program requests a word from port 2, then the console displays the following message:

```
?UNIT 0 PORT 2H REQUESTS WORD INPUT (ENTER VALUE) :
```

Enter the desired value in the space provided and follow it with a carriage return (or, for IBM PC hosts, <Enter>). The I²ICE system reads the value in the current number base.

When the user program writes an I/O port, the console displays the result. For example, if the first I/O partition is mapped to ICE and the user program writes port 2 with the value 1 expressed as a word, the console displays the following:

```
?UNIT 0 PORT 0002H OUTPUT WORD 0001H
```

The I²ICE system displays output in hexadecimal regardless of the current number base.

Simulating I/O with a Debug Procedure

A debug procedure is a named group of I²ICE commands. To simulate I/O using a debug procedure, map one or more I/O partitions to ICE and follow the keyword ICE with the name of a previously-defined debug procedure, as follows:

```
*MAPIO 0 LENGTH 64T ICE(money)
```

Input

When the user program requests input, the I²ICE system calls the specified debug procedure, and that procedure supplies the value. Use the following commands to define a debug procedure called money that supplies 100 when the program reads port 2 and 65 when the program reads port 4:

```
*DEFINE PROC money = DO
. *IF %0 = = 2 THEN
. . *PORTDATA = 100T
. . *ELSE IF %0 = = 4 THEN
. . . *PORTDATA = 65T
. . . *END
. . *END
. *END
```

The percent sign (%) identifies a system parameter passed to the debug procedure. Table 3-1 lists the system parameters used with I/O debug procedures.

Table 3-1 System Parameters Used with I/O Debug Procedures

System Parameter	Description
%0	The port number.
%1	TRUE for a read and FALSE for a write.
%2	TRUE for a byte-port access; FALSE for a word-port access.

To map the first I/O partition to ICE(money), enter the following:

```
*MAPIO 0 LENGTH 64T ICE(money)
*MAPIO 64T LENGTH 64T USER
*MAPIO
MAPIO 00000H LENGTH 00040H ICE MONEY
MAPIO 00040H LENGTH 00080H USER
```

If you run the example program cmaker.86 in Chapter 2, the console does not prompt for input. Instead, the input is supplied by the debug procedure money.

Output

When the user program writes an I/O port which belongs to an I/O partition mapped to ICE-(money), the debug procedure money handles the output. The program cmaker.86 writes to port 64. To change the I/O map so that port 64 is mapped to money, enter the following command:

```
*MAPIO 0 LENGTH 128T ICE(money)
```

The debug procedure must write the output value to the console or store the output value in a debug variable.

The following example is a modification of money that reads 100 from port 2, reads 65 from port 4, and writes to the console.

```
*DEFINE PROC money = DO
. *IF %1 THEN
. . *IF %0 = = 2 THEN
. . . *PORTDATA = 100T
. . . *ELSE IF %0 = = 4 THEN
. . . . *PORTDATA = 65T
. . . . *END
. . . *END
. . *ELSE WRITE PORTDATA
. . *END
. *END
```

To store the output in the previously-defined debug variable answer instead of writing it to the console, replace WRITE PORTDATA with answer = PORTDATA.

If the user program is writing a word-wide port, then answer should be of type WORD. A debug variable of type WORD is treated as a 16-bit unsigned integer. Define the type WORD debug variable answer as follows:

```
*DEFINE WORD answer
```

If the user program is writing a byte-wide port, then the debug variable should be of type BYTE. A debug variable of type BYTE is treated as an 8-bit unsigned integer. Define the type BYTE debug variable ans as follows:

```
*DEFINE BYTE ans
```

The next example is a modification of the debug procedure money that stores the output value in answer if the write is to a word-wide port and in ans if the write is to a byte-wide port.

```
*DEFINE PROC money = DO
. *IF %1 THEN
. . *IF %0 = = 2 THEN
. . . *PORTDATA = 100T
```

```
. . . *ELSE IF %0 = = 4 THEN
. . . . *PORTDATA = 65T
. . . . *END
. . . *END
. . *ELSE IF %2 THEN
. . . *ans = PORTDATA
. . . *ELSE answer = PORTDATA
. . . *END
. . *END
. *END
```

The Emulation Clips

The emulation clips pod has eight input signals and four output signals. Note that these lines are TTL inputs and outputs.

The Clipsin Lines

Applications for the clipsin lines are setting up the I²ICE system to trigger on a hardware event (the assertion of a signal or signals by the prototype) and using the I²ICE system to inspect the state of user signals. For example, to break emulation if a prototype signal goes high (becomes 1) when an over temperature condition exists, connect that signal to clipsin 0. Then, enter the following I²ICE command:

```
*GO TIL CLIPS 0XXXXXX1Y
```

The following example displays the current setting of the clipsin lines:

```
*CLIPSIN
1EH
```

Interpret the result in the following way:

line number	7	6	5	4	3	2	1	0
value	0	0	0	1	1	1	1	0

When a clipsin line is disconnected, it returns zero.

The Clipsout Lines

You can set and read the last value written to the two clipsout lines with the CLIPSOUT command. The following example sets clipsout1 to 0 and clipsout0 to 1:

```
*BASE = BINARY
*BASE
BINARY
```

```
*CLIPSOUT=01Y
*CLIPSOUT
01Y
```

The two clipsout lines that can be set by users with the CLIPSOUT command are zero by default. The other two clipsout lines are the system break and trace lines. An armed I²ICE system asserts the system break line when it encounters a breakpoint. An I²ICE system, armed or not, always asserts the system trace line when it collects trace data.

The SYS BREAK/ and SYS TRACE/ clips output lines are provided to enable users to connect to devices that are not part of your I²ICE system.

Emulating a Program

This section contains a sample Pascal program that illustrates how to get the I²ICE system up and running. The sample program is a procedure in a larger program that controls an automatic change making and coin release mechanism.

The steps shown are compiling, linking, and locating the program, invoking the I²ICE system, loading the located program for emulation, setting a breakpoint, and resuming emulation.

Preparing a Pascal Program

The program reads in the amount tendered (the variable paid) and the amount of the purchase (the variable purchase). It determines whether the resulting change contains any coins. If so, the program sets the variable coinrelease to 1. Then, the program writes coinrelease to I/O port 64T (40H).

Use a text editor to construct a file containing the following Pascal source code.

```
PROGRAM cmaker;

VAR    coins,change                 :INTEGER;
       coinrelease, paid, purchase  :WORD;

BEGIN
INWRD(2,paid);
INWRD(2,purchase);

change := paid-purchase;
coins := change MOD 100;

IF coins > 0 THEN
      coinrelease := 1
ELSE coinrelease := 0;

OUTWRD(64,coinrelease);
END.
```

Call the file cmaker.SRC and write it on the disk mounted on drive 0. The file's pathname is :F0:cmaker.SRC.

Compiling the Source File

This example assumes that the Pascal compiler is on disk drive 1 of a Series III development system and that the source file is on disk drive 0. The resulting object file is on disk drive 0.

```
-RUN :F1:PASC86 cmaker.SRC DEBUG XREF

SERIES-III Pascal-86, Vx.y
PARSE(0), ANALYZE(0), XREF, OBJECT

    COMPILATION OF cmaker COMPLETED, 0 ERRORS DETECTED
    END OF PASCAL-86 COMPILATION
```

Linking the Object File

This example assumes that the linker, library files, and resulting link file are on disk drive 1.

```
-RUN :F1:LINK86 cmaker.OBJ, :F1:P86RN0.LIB, :F1:P86RN1.LIB, &
>>:F1:EH87.LIB, :F1:E8087.LIB, :F1:E8087, :F1:RTNULL.LIB &
>>to cmaker.LNK
SERIES III 8086 LINKER, Vx.y
```

Locating the Link File

This example assumes that the locater, linker, run-time library, and source file are on disk drive 1. Note that the I²ICE system does not accept load-time-locatable code.

```
-RUN :F1:LOC86 cmaker.LNK TO cmaker.86
SERIES III LOCATER, Vx.y
```

Creating a SUBMIT File

The following submit file named lnkloc links and locates an object file and assumes that the linker, the run-time library files, and the locater are on disk drive 1 and that the object file is on disk drive 0. Execute the submit file by entering the SUBMIT command. Note that the %0 parameter passes the root name of the object file named cmaker.

```
RUN
:F1:LINK86 %0.OBJ, &
            :F1:P86RN0.LIB, :F1:P86RN1.LIB, &
            :F1:EH87.LIB, :F1:E8087.LIB, :F1:E8087, &
            :F1:RTNULL.LIB TO %0.LNK
:F1:LOC86 %0.LNK TO %0.T01
EXIT
```

```
-SUBMIT :F1:lnkloc(cmaker)
-RUN
ISIS-II RUN 8086, Vx.y
>:F1:LINK86 cmaker.OBJ, &
>>              :F1:P86RN0.LIB, :F1:P86RN1.LIB, &
>>              :F1:EH87.LIB, :F1:E8087.LIB, :F1:E8087, &
>>              :F1:RTNULL.LIB TO cmaker.LNK
SERIES-III 8086 LINKER, Vx.y
COPYRIGHT 19xx, INTEL CORPORATION
>:F1:LOC86 cmaker.LNK TO cmaker.T01
SERIES-III 8086 LINKER, Vx.y
COPYRIGHT 19xx, INTEL CORPORATION
>EXIT
-:F0:SUBMIT RESTORE :F1:lnkloc.CS(:VI:)
```

Getting Ready to Emulate

Before starting emulation, you must invoke the I²ICE software, set up the memory and I/O maps, and load the program file.

Invoke the I²ICE software on the Series III by entering the following command:

```
-RUN I2ICE
```

This example assumes that you have a Series III with a system disk in drive 0 that contains host code, probe code, the error file, the help file, and the RUN program. For information on how to install and invoke software on your host system, see the appendix in this manual that explains software installation for your host. For more information on the use of the I2ICE command, see the I2ICE entry in the *I²ICE™ System Reference Manual.*

The host development system's console responds with the following sign-on message:

```
SERIES III I2ICE Vx.y
Copyright 1984, 1985 INTEL CORPORATION
86 Probe Version Vx.y
*
```

The asterisk (*) is the I²ICE prompt.

The I²ICE software provides a menu at the bottom of the screen that shows all the commands and parameters you can enter at a particular command level. The following menu is the first menu you see after invoking the I²ICE software:

```
---- more ---- Use [TAB] to cycle through prompts when "more" appears.
APPEND  ARMREG  BASE  BRKREG  CALLSTACK  CAUSE  CLEAREOL  CLEAREOS
```

In most cases, the complete menu will not fit on one line. To see more of the menu, press the TAB key.

```
---- more ----
CLIPSIN  CLIPSOUT  COUNT  CURHOME  DEFINE  DIR  DISABLE  EDIT  ENABLE
```

The I²ICE menu is circular. Press TAB enough times, and you will return to the first line of the menu.

Use the MENU command to suppress the menu display. Enter MENU = 0 to suppress menu display and MENU = 1 to display the menu. CTRL-V also changes the menu display mode. (Note that the MENU and CTRL-V commands also control automatic expansion of LITERALLY definitions; thus, the menu display and the automatic expansion are both on together or off together.)

Now look at the I²ICE memory map.

```
*MAP
MAP 0K LENGTH 1024K GUARDED
```

Use the MAP command to direct memory references to high-speed memory (HS). Use the MAPIO command to transfer I/O data values between the first 128 I/O ports to the host development system's console and the I²ICE probe.

```
*MAP 0 LENGTH 32K HS
*MAPIO 0 LENGTH 128T ICE
```

Confirm this by looking at the map again (note that 80 hexadecimal is 128 decimal).

```
*MAP
MAP 0K LENGTH 32K HS
MAP 32K LENGTH 992K GUARDED
*MAPIO
MAPIO 00000H LENGTH 00080H ICE
MAPIO 00080H LENGTH 0FF80H USER
```

Load the program file by entering the following command (the example assumes that the program file is located in drive 1):

```
*LOAD :F1:cmaker.86
```

Emulating Your Program

First display the I²ICE pseudo-variable $, which represents the current execution point.

```
*$
0020:0006H
```

The address 20:6 is where LOC86 placed the beginning of your program. This address is easier to remember if you give it a name (such as begin) by defining a debug variable, as follows.

```
*define pointer begin = $
```

To begin emulation, enter the GO command.

```
*GO
```

The console requests a value for paid, then a value for purchase. Enter 75 for each. Note that the I²ICE system default radix is decimal.

```
?UNIT 0 PORT 0002H REQUESTS WORD INPUT (ENTER VALUE) :75
?UNIT 0 PORT 0002H REQUESTS WORD INPUT (ENTER VALUE) :75
```

The console displays the output written by the program to I/O port 40H (64T), in this case a 0 because you paid the exact amount of your purchase.

```
?UNIT 0 PORT 0040H OUTPUT WORD 0
```

Finally, emulation halts because of bus inactivity.

```
?Probe 0 stopped at location 0027:016EH because of bus not active
  Bus address = 0203DE   Trace Buffer Overflow
```

To resume emulation, you must return the execution point to the beginning of the program.

```
*$ = begin
```

For the next emulation, create a debug register that enables you to break when the program writes coinrelease. To instruct the I²ICE system to break at coinrelease, you must give the fully qualified reference to the address of coinrelease (.:cmaker.coinrelease). Define the debug register (called cwrite) and begin emulation as follows:

```
*DEFINE SYSREG cwrite = WRITE AT .:cmaker.coinrelease
*GO USING cwrite
```

The console displays a request for the value of paid and purchase. Enter 80 for paid and 75 for purchase.

```
?UNIT 0 PORT 0002H REQUESTS WORD INPUT (ENTER VALUE) :80
?UNIT 0 PORT 0002H REQUESTS WORD INPUT (ENTER VALUE) :75
*Probe 0 stopped at :CMAKER#11 because of bus break
  Break register is CWRITE   Trace Buffer Overflow
```

The break occurs. The program has not yet completed execution. It has already set coinrelease (presumably to 1 because this time you have a nickel coming), but it has not yet written coinrelease to the output port. You can display the value stored in coinrelease symbolically.

```
*:cmaker.coinrelease
+1
```

You can also display both the value and address of coinrelease by applying the WORD memory template.

```
*WORD .:cmaker.coinrelease
003E:0004H          1
```

Now use the disassembly command ASM to find the current execution point and the instruction stored there.

```
*ASM $
:cmaker#11
0021:0054H    A10400              MOV AX,WORD PTR 0004H
```

The closest source code line to the current execution point (the address of the next instruction to be executed) is statement #11 in the module called cmaker. The current execution point is 0021:0054H. The hexadecimal content of this memory location is A10400, which represents the following instruction:

```
MOV AX,WORD PTR 0004H
```

This instruction loads the 16-bit accumulator AX with the contents of memory location 0004H.

Now resume emulation with the GO command.

```
*GO
?UNIT 0 PORT 0040H OUTPUT WORD 0001H
?Probe 0 stopped at location 0027:016EH because of bus not active
  Bus address = 0203DE
```

During this session you defined two debug objects: a debug variable named begin and a debug register named cwrite. Look at the debug objects with the DIR command.

```
*DIR DEBUG
BEGIN     .     .     pointer 0020:0006H
CWRITE    .     .     sysreg
```

To save the definitions of the debug objects for future use, write them to a file named deb.001 with the PUT command. (This example assumes that you will be writing to drive :F1:. For information on how to specify directories and drives on the Series IV or on an IBM PC host, see the Pathname entry in the *I²ICE™ System Reference Manual.*)

```
*PUT :F1:deb.001 begin,cwrite
```

Enter the EXIT command to return to the host development operating system.

```
*EXIT
```

You cannot use the EXIT command in two cases:

- If any probe has any memory mapped to MULTIBUS (MB) memory (reset MAP by entering RESET MAP before exiting).
- If any of I/O memory is mapped to ICE while any probe is emulating (reset MAPIO by entering RESET MAPIO before exiting).

The following example shows how to put your prototype into emulation, exit I²ICE software, and then return without losing the program.

```
*GO
?EXIT
I2ICE terminated
-RUN I2ICE RESTART
```

Although you exited and re-invoked the I²ICE software without interrupting the emulation of the user program, you did lose your program's symbol table. The following example shows reloading just the symbol table and line numbers by issuing the LOAD command with the NOCODE option.

```
*LOAD :F1:cmaker.86 NOCODE
```

Breaking, Tracing, and Arming

This section explains how to set breakpoints and how to control and interpret the trace buffer.

The Example

The sample program used in this section is an expansion of the program used in previous section. This version first reads the amount tendered (paid) and the amount of the purchase (purchase). The program then calculates the number of quarters, dimes, nickels, and pennies needed and sends each result to a different I/O port. The list file produced by the PLM86 compiler is as follows:

```
Source File: CMAKER.SRC
Object File: CMAKER.OBJ
Controls Specified: XREF, DEBUG.

STMT   LINE   NESTING    SOURCE TEXT: CMAKER.SRC
  1      1     0  0      PROGRAM CMAKER;

  2      3     0  0      VAR    coins, change                  :INTEGER;
  3      4     0  0             quarters,nickels,dimes,pennies :INTEGER;
  4      5     0  0             paid, purchase                 :WORD;

  5      7     0  0      PROCEDURE payment;
  6      8     1  0        BEGIN
  6      9     1  1        OUTWRD(70,quarters);
  7     10     1  1        OUTWRD(72,dimes);
  8     11     1  1        OUTWRD(74,nickels);
  9     12     1  1        OUTWRD(76,pennies)
                            END;

 10     15     0  0      BEGIN
 10     16     0  1      INWRD(2,paid);
 11     17     0  1      INWRD(4,purchase);

 12     19     0  1        change  :=  paid-purchase;
 13     20     0  1        coins   :=  change MOD 100;
 14     21     0  1        quarters :=  coins DIV 25;
 15     22     0  1        coins   :=  coins MOD 25;
 16     23     0  1        dimes   :=  coins DIV 10;
 17     24     0  1        coins   :=  coins MOD 10;
 18     25     0  1        nickels :=  coins DIV 5;
 19     26     0  1        pennies :=  coins MOD 5;

 20     28     0  1        payment;
 21     29     0  1        END.
```

Emulating a User Program

The previous section explained that you begin emulation by using the command GO. By default, GO starts emulation from the current execution point and with the last specifications that were given with the GO command.

GO FOREVER is the default condition. The GO FOREVER command causes emulation to proceed without any breakpoint, trace, or arm specifications. Unless the following time-out pseudo-variables are enabled (which is their default condition), GO FOREVER causes emulation to continue until you enter the HALT command.

BUSACT	allows a system time-out when the processor bus is inactive for more than one second.
IORDY	allows a system time-out when an I/O access takes more than one second.
MEMRDY	allows a system time-out when memory access time is longer than one second.
PHANG	allows a system time-out when coprocessor memory accesses exceed one second (8086/8088 and 80186/80188 probes only).

These and the other pseudo-variables displayed by the STATUS command control the emulation environment. Refer to the STATUS entry in the *I²ICE™ System Reference Manual* for information on the STATUS command and the pseudo-variables.

The GO specifications include breakpoint, trace, and arm specifications. You can include these specifications within the GO command itself, or you can store them in debug registers for use in later emulations.

A breakpoint specification specifies a condition that causes a break to occur. (For more information on what kinds of break specification conditions can be selected, see the GO entry in the *I²ICE™ System Reference Manual.*) For example, the following command specifies a breakpoint when statement #11 of the user program cmaker.86 is executed.

```
*GO TIL :cmaker#11
```

A trace specification specifies a combination of one or more events that activate trace collection. (For more information on what kinds of trace specification conditions can be selected, see the GO entry in the *I²ICE™ System Reference Manual.*) Normally the I²ICE system collects trace data during emulation. However, when you include a trace specification, tracing only occurs when the trace specification is satisfied. For example, the following command activates tracing whenever the procedure payment is executed.

```
*GO TRACE :cmaker.payment
```

An arm specification specifies both a breakpoint and an arm window. The arm window determines when the I²ICE system can recognize the breakpoint. (For more information on what kinds of arm specification conditions can be selected, see the GO entry in the *I²ICE™ System Reference Manual.*) For example, the following command opens the arm window only when the procedure payment is executing. If the program variable quarters is read and the arm window is open, a break occurs.

```
*GO TIL ARM :cmaker.payment
**DISARM OUTSIDE :cmaker.payment
**TRIG READ AT .:cmaker.quarters
```

The Event Machines

The I²ICE system contains two event machines: the execution event machine (XEM) and the system event machine (SEM). When you set breakpoint, trace, and arm specifications, you are programming one or both of these event machines.

An execution event is the execution of an instruction. You specify an execution event as an address or range of addresses. The execution event occurs when the microprocessor is ready to execute the instruction that came from the address or address range. Execution begins when the first byte of the instruction is taken from the instruction queue.

A system event describes activity on the microprocessor address lines, the microprocessor data lines, the microprocessor status lines, or the clips input lines. The I²ICE system can read the state of the clips input lines and break, trace, or arm on their values.

You can program the event machines directly if you need to specify a complex event. Each event machine has four states (S0 through S3). Each state represents a control branch that can detect match conditions (e.g., break or trace), initiate actions, or branch to a new state. State S3 sets up a communication link between the two event machines so that decisions can be made in one machine based on the condition of the other.

In addition, each event machine has a counter that you can set and conditionally increment. Then, you can change the event machine's state if the counter is equal to the set value. (For more information on the event machines, see the Event machines entry in the *I²ICE™ System Reference Manual.*)

The Debug Registers

A debug register is a software register that you create and name with an I²ICE command. The five types of debug registers are arm (ARMREG), break (BRKREG), system (SYSREG), trace (TRCREG), and event (EVTREG). You can define any number of each type, edit them, emulate with them, and write them to a disk file for later debug sessions. (Detailed information on each register type is given in the corresponding entry in the *I²ICE™ System Reference Manual.*)

Arm Registers

Arm registers set conditional breakpoints that enable breaking within windows. A break window is opened when an arm condition is encountered and closed when a disarm condition is encountered. For example, you can define an arm register to open the arm window when your program is executing the procedure payment. If your program reads the variable quarters while the arm window is open, a break occurs. The following example defines an arm register called arm1.

```
*DEFINE ARMREG arm1 =
**ARM :cmaker.payment
**DISARM OUTSIDE :cmaker.payment
**TRIG READ AT .:cmaker.quarters
**END
```

To emulate with this arm register, enter the following GO command:

```
*GO USING arm1
```

Break Registers

Break registers stop emulation when the target line of code is executed. For example, define a break register called break1 that breaks when your program begins to execute either statement #15 or statement #19.

```
*DEFINE BRKREG break1 = :cmaker#15,:cmaker#19
```

To emulate with this break register, enter the following GO command:

```
*GO USING break1
```

System Registers

System registers define breaks on operand access, operand data, logic clips, system breaks, and coprocessor cycles. For example, define a system register called sys1 that breaks when your program reads the variable quarters.

```
*DEFINE SYSREG sys1 = READ AT .:cmaker.quarters
```

To emulate with this system register, enter the following GO command:

```
*GO USING sys1
```

Trace Registers

Trace registers specify under what conditions trace information is collected. For example, define a trace register called trace1 that causes the I²ICE system to begin collecting trace information when your program begins executing statement #12 and to stop collecting trace information at statement #13.

```
*DEFINE TRCREG trace1 = :cmaker#12 TO :cmaker#13
```

To emulate with this trace register, enter the GO command:

```
*GO TRACE trace1
```

Note that with a trace register, you must use the TRACE option of the GO command and not the USING option of the GO command.

Event Registers

Event registers control the event machines directly. For example, define an event register called qwriteinpay.

```
*DEFINE EVTREG qwriteinpay = DO
**SEM S0 IF READ AT .:cmaker.quarters IS 2 THEN GOTO S1
**       S1 IF XLINK THEN BREAK ELSE GOTO S0
**XEM S0 IF :cmaker.payment THEN GOTO S3
**       S3 ALWAYS GOTO S3
**END
```

To emulate with this event register, enter the following GO command:

```
*GO USING qwriteinpay
```

Debug Registers Calling Debug Procedures

One advantage to using debug registers is that you can store them in a file for use in later debug sessions. Another advantage is the ability to automatically execute a debug procedure when the specification in the debug register is satisfied. For example, the debug register qwriteinpay executes the debug procedure query when the event specification is satisfied. This debug procedure must return a TRUE or FALSE value. If the value is TRUE, a break occurs, and the I²ICE system enters interrogation mode. If the value is FALSE, a break occurs, but the I²ICE system restarts emulation and does not enter interrogation mode.

```
*DEFINE EVTREG qwriteinpay = DO
**SEM S0 IF READ AT .:cmaker.quarters IS 2 THEN GOTO S1
**       S1 IF XLINK THEN BREAK ELSE GOTO S0
**XEM S0 IF :cmaker.payment THEN GOTO S3
**       S3 ALWAYS GOTO S3
**CALL query
**END
```

Interpreting the Trace Buffer

The trace buffer contains trace information and consists of 1023 48-bit frames. Using the PRINT command, you can display the trace buffer as either disassembled instructions (INSTRUCTIONS mode) or as execution and bus cycles (CYCLES mode).

This section contains an example of a trace display. To obtain the trace display, compile, link, and locate the Pascal program listed in the Example section in this chapter. Call the resulting program file cmaker.86.

The following commands load the example program, set the debug variable begin to $, and start emulation. A break occurs when statement #6 begins executing. Tracing starts when statement #12 begins executing and ends when statement #15 begins executing. The example shows that the user entered 80 and 75 when prompted for input.

```
*MAP 0 LENGTH 32K HS
*MAPIO 0 LENGTH 128T ICE
*LOAD cmaker.86
*DEFINE POINTER begin = $
*GO FROM begin TIL :cmaker#6 TRACE :cmaker#12 TO :cmaker#15
?UNIT 0 PORT 2H REQUESTS WORD INPUT (ENTER VALUE) :80
?UNIT 0 PORT 4H REQUESTS WORD INPUT (ENTER VALUE) :75
*Probe 0 stopped at :CMAKER#6 + 3 because of execute break
```

The INSTRUCTIONS trace display shown in Figure 3-1 assumes that the I²ICE system is running the 8086/8088 probe.

The Pascal statement #12 is the following:

```
change := paid-purchase;
```

This high-level-language statement consists of three assembly language instructions. The first instruction reads a word from memory and places that word in the CX register. This is the access of the program variable paid. The second instruction subtracts the contents of the register AX from the contents of CX and places the answer in CX. CX now contains the program variable change. The third instruction writes change to memory.

Note that the data write belonging to statement #12 occurs during the execution of statement # 13. The 8086 architecture provides for an execution unit and a bus interface unit. These two units operate independently of each other. The 8086 microprocessor realizes that the MOV instruction is finished except for a memory access. The execution unit takes the next instruction off the queue while the bus interface unit writes to memory.

Figure 3-2 displays the trace buffer in CYCLES mode. The trace buffer contains the same trace data as the previous example. The only difference is that the data is now displayed in CYCLES mode rather than INSTRUCTIONS mode.

The first frame (f 000) is an execution cycle. The execution address is 00024F. This frame shows the execution of the following instruction:

```
MOV CX,WORD PTR 0012H
```

```
*PRINT INSTRUCTIONS ALL
FRAME    ADR          BYTE          MNEMONICS  OPERANDS                UNIT 0
:CMAKER#12
000 0021:003FH  M 8B0E1200       MOV  CX,WORD PTR 0012H
004 0021:0043H  M 2BC8           SUB  CX,AX
006 0021:0045H    890E0400       MOV  WORD PTR 0004H,CX
:CMAKER#13
00A 0021:0049H    8BC1           MOV AX,CX
       000444H-DW-0005H
00C 0021:004BH    99             CWD
00E 0021:004CH    BE6400         MOV  SI,0064H ; +100T
010 0021:004FH    F7FE           IDIV  SI
013 0021:0051H    0BD2           OR   DX,DX
014 0021:0053H    7D02           JGE    $+0004H ; A=0057H
018 0021:0057H    8916060D6      MOV  WORD PTR 0D606H,DX
:CMAKER#14
01D 0021:005BH    8BC2           MOV  AX,DX
       000446H-DW-0005H
01F 0021:005DH    99             CWD
021 0021:005EH    B91900         MOV  CX,0019H ; +25T
023 0021:0061H    F7F9           IDIV  CX
026 0021:0063H    A30E00         MOV  WORD PTR 000EH,AX
:CMAKER#15
028 0021:0066H    A10600         MOV  AX,WORD PTR 0006H
       00044EH-DW-0000H  000446H-DR-0005H
```

Figure 3-1 Sample Trace Buffer INSTRUCTIONS Display Showing the Data Write for Instruction #12

When displayed in INSTRUCTIONS mode, the trace buffer shows this instruction's address as a selector:offset pair, 0021:003F. The 8086 microprocessor calculates the 20-bit address by shifting the selector left four bits and adding the offset as shown in the following example:

```
 00210
+ 003F
------
 0024F
```

Figure 3-2 shows this calculation in the margin.

The next frame (f 001) is a bus cycle. The symbol DW identifies it as a write to the data segment. (DW is an access code. The access codes are probe-specific and are defined in the entry entitled Trace buffer display in the *PICE™ System Reference Manual.*) The data is 004B (75 in decimal). This is the write of the program variable purchase. It is the write associated with the previous Pascal statement: INWRD(4,purchase).


```
*PRINT CYCLES ALL
EXEC ADR  BUS ADR    DATA    STATUS   CLIPS FRAME  TIME          LEVEL  UNIT 0
x 00024F  b          d       s        c     f 000  0.0 nanosecs  0
x         b 000450   d 0048  s 00DE DW c     f 001
x         b 000254   d 89C8  s 00D4 CF c     f 002
x         b 000452   d 0050  s 00DD DR c     f 003
x 000253  b          d       s        c     f 004  3.0 microsecs  0
x         b 000256   d 040E  s 00D4 CF c     f 005
x 000255  b          d       s        c     f 006  3.6 microsecs  0
x         b 000258   d 8800  s 00D4 CF c     f 007
x         b 00025A   d 99C1  s 00D4 CF c     f 008
x         b 00025C   d 648E  s 00D4 CF c     f 009
x 000259  b          d       s        c     f 00A  6.8 microsecs  0
x         b 000444   d 0005  s 00DE DW c     f 00B
x 00025B  b          d       s        c     f 00C  7.2 microsecs  0
x         b 00025E   d F700  s 00D4 CF c     f 00D
x 00025C  b          d       s        c     f 00E  8.2 microsecs  0
x         b 000260   d 0BFE  s 00D4 CF c     f 00F
x 00025F  b          d       s        c     f 010  9.0 microsecs  0
x         b 000262   d 7DD2  s 00D4 CF c     f 011
x         b 000264   d 0302  s 00D4 CF c     f 012
x 000261  b          d       s        c     f 013  42.0 microsecs 0
x 000263  b          d       s        c     f 014  42.6 microsecs 0
x         b 000266   d 89D6  s 00D4 CF c     f 015
x         b 000268   d 0616  s 00D4 CF c     f 016
EXEC ADR  BUS ADR    DATA    STATUS   CLIPS FRAME  TIME          LEVEL  UNIT 0
x         b 000267   d 89D6  s 00D4 CF c     f 017
x 000267  b          d       s        c     f 018  46.2 microsecs  0
x         b 000268   d 0616  s 00D4 CF c     f 019
x         b 00026A   d 8800  s 00D4 CF c     f 01A
x         b 00026C   d 99C2  s 00D4 CF c     f 01B
x         b 00026E   d 19B9  s 00D4 CF c     f 01C
x 00026B  b          d       s        c     f 01D  50.2 microsecs  0
x         b 000446   d 0005  s 00DE DW c     f 01E
x 00026D  b          d       s        c     f 01F  50.6 microsecs  0
x         b 000270   d F700  s 00D4 CF c     f 020
x 00026E  b          d       s        c     f 021  51.6 microsecs  0
x         b 000272   d A3F9  s 00D4 CF c     f 022
x 000271  b          d       s        c     f 023  52.4 microsecs  0
x         b 000274   d 000E  s 00D4 CF c     f 024
x         b 000276   d 06A1  s 00D4 CF c     f 025
x 000273  b          d       s        c     f 026  85.4 microsecs  0
x         b 000278   d 9900  s 00D4 CF c     f 027
x 000276  b          d       s        c     f 028  87.6 microsecs  0
x         b 00044E   d 0000  s 00DE DW c     f 029
x         b 00027A   d F9F7  s 00D4 CF c     f 02A
x         b 000446   d 0005  s 00DD DR c     f 02B
```

Figure 3-2 Sample Trace Buffer Display in CYCLES Mode Showing Frames f 006-008 and f 00B

The next frame (f 002) is a fetch from the code segment. The data is 89C8. The C8 belongs to statement #12's second instruction, SUB CX,AX. The 89 begins statement #12's third instruction, MOV WORD PTR 0004H,CX.

The next frame (f 003) is a data read. The data is 0050 (80 in decimal). This is the read of the program variable *paid*. Note that this read does not appear in the trace buffer when the buffer is displayed in INSTRUCTIONS mode because the first entry in the trace buffer is an execution cycle. The trace buffer was not active when the probe fetched that instruction. Hence, the I²ICE system must go to memory to disassemble the instruction (which is what the M signifies in INSTRUCTIONS mode). When the I²ICE system goes to memory, it does not use any of the trace buffer for disassembly until the next execution cycle. The read would be visible if the trace were started early enough to allow the trace buffer to record the fetch of the instruction that does the read.

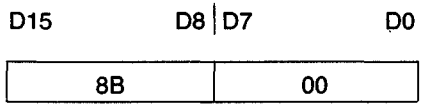
The next frame (f 004) is an execution cycle. This is the execution of the instruction SUB CX,AX

Frame 00B (underlined in Figure 3-2) is the data write focused on in Figure 3-1. This frame is the data write performed by the instruction MOV WORD PTR 000EH, CX. Its execution frame is f 006 (also underlined in Figure 3-2). The previous trace display listed the instruction's logical execution address as 0021:0049. This trace display shows 259, the corresponding physical address.

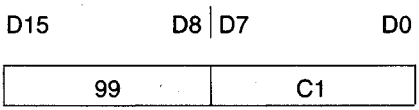
Notice the two code fetches occurring in frames 007 and 008. (In Figure 3-2, the bus address and data for these two frames are high-lighted.) In f 007 the microprocessor reads 8B00 from memory. In f 008 the microprocessor reads 99C1. iAPX microprocessors store the least significant byte of a word in the lower address. To see how memory looks, use the following BYTE command.

```
*BASE = 10H
*BYTE 258 TO 25B
000258H          00   8B   C1   99          '....'
                  .    .    .    .
                  .    .    .    .
                258  259  25A  25B
```

When the iAPX microprocessor reads a word from memory, the byte at the even address travels on the lower half of the data bus, and the byte at the odd address travels on the upper half. When the iAPX microprocessor reads the word at 258, the data bus looks like the following:



When the iAPX microprocessor reads the word at 25A, the data bus looks like the following:



The lower byte on the data bus <D7-D0> enters the instruction queue before the upper byte <D15-D8>. The following example illustrates the queue:

From address	Data bus	Queue		
258	<D7-D0>	(top)	00	
259	<D15-D8>		8B	MOV AX,CX
25A	<D7-D0>		C1	
25B	<D15-D8>	(bottom)	99	CWD

The word 8BC1 represents the instruction MOV AX,CX, the first instruction of statement :cmaker#13. The byte 99 represents the instruction CWD, the second instruction of statement :cmaker#13.

The trace buffer normally collects both execution cycles and bus cycles. It overflows after 1023 frames have been collected. When the trace buffer overflows, it shifts all the frames toward frame 0. The old frame 0 is lost. The 1023rd frame is always the latest frame.

The Timetag

In the CYLES mode display, the column labeled TIME shows the execution time. Frame 0 always begins at time 0.

The I²ICE system contains a free-running counter that returns to zero after 2K (2048) counts. The time increment for each count is set by the TIMEBASE pseudo-variable. Its default is 200 nanoseconds. The largest value you can set TIMEBASE to is 6 milliseconds. TIMEBASE must be a multiple of 100 nanoseconds.

When the I²ICE system begins tracing, the value of this free-running counter goes directly into the trace buffer. The I²ICE system, however, offsets the value so that frame 0 always begins at time 0.

If you interrupt the trace, the I²ICE system starts another clock that runs until tracing resumes. If a wrap-around occurs (i.e., the counter reaches 2048), the I²ICE system sets the level flag. When you resume tracing, the LEVEL column is incremented by one (regardless of how many wrap-arounds occurred), and the TIME column is reset to 0.0. You will have lost time calibration because you do not know how many wrap-arounds occurred while the trace was interrupted. Note that these difficulties caused by wrap-around can be avoided by setting TIMEBASE to a higher value.

The Pseudo-Variable TRCBUS

If you are not interested in bus activity and you want to collect more execution cycles before overflowing, set the I²ICE pseudo-variable TRCBUS to FALSE. Its default is TRUE.

Trace Buffer Information

For more information on the trace buffer, see the entries PRINT and Trace buffer display in the *I²ICE™ System Reference Manual.*

Hardware Slipping On a Breakpoint

Because emulation is in real time, for the 8086/8088 and 80186/80188 probes you cannot break exactly where you specified. To break on the execution of an instruction, the I²ICE system must first recognize that the instruction is executing. (The 80286 probe has a special feature that prevents hardware slipping.)

The following example illustrates hardware slipping.

```
*GO FROM begin TIL :cmaker#13 TRACE :cmaker#12 to :cmaker#13
?UNIT 0 PORT 2H REQUESTS WORD INPUT (ENTER VALUE) :80
?UNIT 0 PORT 4H REQUESTS WORD INPUT (ENTER VALUE) :75
Probe 0 stopped at :CMAKER#13 + 3 because of execute break
```

The I²ICE system is designed to break right after the execution of the specified instruction, and specifying statement #13 specifies the first assembly language instruction making up statement #13. The I²ICE system informs you that the probe stopped three bytes past the beginning of statement #13. The trace buffer gives you more information.

```
*PRINT INSTRUCTIONS ALL
FRAME    ADR          BYTE          MNEMONICS  OPERANDS      UNIT 0
:CMAKER#12
000 0021:003FH  M 8B0E1200       MOV  CX,WORD PTR 0012H
004 0021:0043H  M 2BC8           SUB  CX,AX
006 0021:0045H    890E0400       MOV  WORD PTR 0004H,CX
:CMAKER#13
00A 0021:0049H    8BC1           MOV AX,CX
        000444H-DW-0005H
00C 0021:004BH    99             CWD
```

The I²ICE system slipped three bytes. The first two bytes make up the instruction MOV AX,CX. These are bytes :CMAKER#13 + 0 and :CMAKER#13 + 1. The next byte (:CMAKER#13 + 2) is the instruction CWD. Execution will resume at :CMAKER#13 + 3.

Even Addresses, Odd Addresses, and Breaking

The iAPX architecture causes a word written to an even address to appear on the data bus once in the normal order (high byte, low byte). A word written to an odd address appears on the data bus twice in reversed order (low byte, high byte) because of the standard 86 architecture. Note this when you specify a breakpoint to occur when the data bus contains a certain value.

When you write a byte to an even address, the byte appears on the lower eight bits of the data bus. When you write a byte to an odd address, the byte appears on the upper eight bits of the data bus.

The examples in this section assume the 8086/8088 probe.

Word Writes to Even and Odd Addresses

First verify that the base is hexadecimal and load the register AX with AB12H, as follows.

```
*BASE=DECIMAL;BASE
HEX
*AX=0AB12H
*AX
AB12
```

The next example fills a section of memory with a number of NOPs and a MOV instruction. The first BYTE command loads the NOPs. The second BYTE command loads the MOV instruction. This MOV instruction moves the value from AX to the even address, 0FC00.

```
*BASE=HEX
*BASE
HEX
*MAP 32K LENGTH 32K HS
*BYTE 32K LENGTH 32K = 90H                    /*These are the NOPs*/
*BYTE 33K LENGTH 3 = 0A3,0,0FC
*ASM 33K
008400H     A300FC          MOV   WORD PTR 0FC00H,AX
```

The next example starts emulation and breaks when a word is written on the data bus whose upper byte is AB.

```
*DEFINE SYSREG even = WRITE IS 0ABXX
*GO FROM 32K USING even
Probe 0 stopped at location 0800:0405H because of bus break
 Break register is EVEN   Trace Buffer Overflow
```

The following example looks at the last five instructions in the trace buffer (see Figure 3-3). Notice the MOV instruction and the data write of AB12 (the data write is underlined in Figure 3-3).

```
*PRINT INSTRUCTIONS NEWEST 5
FRAME    ADR         BYTE          MNEMONICS  OPERANDS                UNIT 0
3F5 0083FEH          90              NOP
3F6 0083FFH          90              NOP
3F8 008403H          A300FC          MOV  WORD PTR 0FC00H,AX
3FA 008403H          90              NOP
        00FC00H-DW-AB12H
3FC 008404H          90              NOP
```

Figure 3-3 Sample Trace Buffer Display in INSTRUCTIONS Mode for Emulation with the System Register EVEN

The next example looks at the last 16 cycles in the trace buffer (see Figure 3-4).

The following example modifies the MOV instruction so that the write is to the odd address 0FC01H.

```
*BYTE 33K LENGTH 3 = 0A3,1,0FC
*ASM 33K
008400H     A301FC          MOV  WORD PTR 0FC01H,AX
```

The following example defines a system register called *odd*. This register specifies a break when the data bus contains the value 12ABH.

```
*DEFINE SYSREG odd = WRITE IS 12AB
*GO FROM 32K USING odd
Probe 0 stopped at location 0800:0404H because of bus break
 Break register is 0DD  Trace Buffer Overflow
```

The following example looks at the trace buffer in INSTRUCTIONS mode and then in CYCLES mode (see Figure 3-5). Note that because you have written to an odd address, two write cycles occurred and the bytes on the bus are reversed.

```
*PRINT CYCLES NEWEST 10H
EXEC ADR  BUS ADR    DATA    STATUS  CLIPS FRAME    TIME          LEVEL UNIT 0
x 0083FA b         d       s         c     f 3EF   402.6 microsecs  0
x        b 0083FE  d 9090  s 0054 CF c     f 3F0
x 0083FB b         d       s         c     f 3F1   403.2 microsecs  0
x 0083FC b         d       s         c     f 3F2   403.8 microsecs  0
x 0083FD b         d       s         c     f 3F3   404.2 microsecs  0
x        b 008400  d 00A3  s 0054 CF c     f 3F4
x 0083FE b         d       s         c     f 3F5   405.0 microsecs  0
x 0083FF b         d       s         c     f 3F6   405.6 microsecs  0
x        b 008402  d 90FC  s 0054 CF c     f 3F7
x 008400 b         d       s         c     f 3F8   406.2 microsecs  0
x        b 008404  d 9090  s 0054 CF c     f 3F9
x 008403 b         d       s         c     f 3FA   408.4 microsecs  0
x        b 00FC00  d AB12  s 005E DW c     f 3FB
x 008404 b         d       s         c     f 3FC   409.0 microsecs  0
x        b 008406  d 9090  s 0054 CF c     f 3FD
x        b 008408  d 9090  s 0054 CF c     f 3FE
```

Figure 3-4 Sample Trace Buffer Display in CYCLES Mode for Emulation with the System Register EVEN

The following list summarizes how to handle breaks on word writes to even and odd addresses.

Assume that word AB12 will be written to the even address FC00. FC00 12 FC01 AB FC02 90 FC03 90 One memory write occurs. The data bus contains AB12.	Assume that word AB12 will be written to the odd address FC01. FC00 90 FC01 12 FC02 AB FC03 Two memory writes occur. The data bus contains 12AB each time.
To break when the word AB12 is written to an even address: ***DEFINE SYSREG evenword =** ****WRITE AT 0FC00 IS 0AB12**	To break when the word AB12 is written to an odd address: ***DEFINE SYSREG oddword =** ****WRITE AT 0FC01 IS 12ABH**

```
*PRINT INSTRUCTIONS NEWEST 5
FRAME   ADR        BYTE              MNEMONICS  OPERANDS                  UNIT 0
3F3 0083FDH        90                  NOP
3F5 0083FEH        90                  NOP
3F6 0083FFH        90                  NOP
3F8 008400H        A301FC              MOV   WORD PTR 0FC01H,AX
        00FC01H-DW-12ABH  ------------------------------> 8086 writes the 12.
3FB 008403H        90                  NOP
        00FC02H-DW-12ABH  ------------------------------> 8086 writes the AB.
*
*PRINT CYCLES NEWEST 10
EXEC ADR  BUS ADR   DATA   STATUS  CLIPS FRAME   TIME          LEVEL  UNIT 0
x 0083FA b        d      s         c     f 3EF  402.4 microsecs   0
x        b 0083FE d 9090 s 0054 CF c     f 3F0
x 0083FB b        d      s         c     f 3F1  403.0 microsecs   0
x 0083FC b        d      s         c     f 3F2  403.6 microsecs   0
x 0083FD b        d      s         c     f 3F3  404.2 microsecs   0
x        b 008400 d 01A3 s 0054 CF c     f 3F4
x 0083FE b        d      s         c     f 3F5  404.8 microsecs   0
x 0083FF b        d      s         c     f 3F6  405.4 microsecs   0
x        b 008402 d 90FC s 0054 CF c     f 3F7
x 008400 b        d      s         c     f 3F8  406.0 microsecs   0
x        b 008404 d 9090 s 0054 CF c     f 3F9
x        b 00FC01 d 12AB s 005E DW c     f 3FA
x 008403 b        d      s         c     f 3FB  409.0 microsecs   0
x        b 00FC02 d 12AB s 005E DW c     f 3FC
x        b 008406 d 9090 s 0054 CF c     f 3FD
x        b 008408 d 9090 s 0054 CF c     f 3FE
```

Figure 3-5 Sample Trace Buffer Displays in Both Modes for Emulation with the System Register ODD

Byte Writes to Even and Odd Addresses

The following example uses a BYTE PTR instead of a WORD PTR in the MOV instruction, puts AB12 into AX, and writes AL (which contains 12) to memory instead of AX.

```
*BYTE 33K LENGTH 3 = 0A2,0,0FC;ASM 33K
008400H     A201FC          MOV   BYTE PTR 0FC00H,AL
*BYTE 33K LENGTH 3 = 0A2,1,0FC;ASM 33K
008400H     A201FC          MOV   BYTE PTR 0FC01H,AL
```

When you write to an even address, the data bus contains AB12. When you write to an odd address, the data bus contains 12AB.

The following list summarizes how to handle breaks on byte writes to even and odd addresses.

Assume that byte 12 is written to the even address FC00.	Assume that byte 12 is written to the odd address FC01.
FC00 12 FC01 90 FC02 90 FC03 90	FC00 90 FC01 12 FC02 90 FC03 90
One memory write occurs. The data bus contains AB12.	One memory access occurs. The data bus contains 12AB.
To break when the byte 12 is read: ***DEFINE SYSREG evenbyte =** ****WRITE AT 0FC00 IS 0XX12**	To break when the byte AB is read: ***DEFINE SYSREG oddbyte =** ****WRITE AT 0FC01 IS 12XX**

Word Reads From Even and Odd Addresses

Assume that you want to break when the user program reads a particular word from memory and that the word AB12 is stored at an even address. The word AB12 appears once on the data lines. Assume that the even address is FC00 and that memory looks as follows:

```
FC00  12
FC01  AB
FC02  34
FC03  56
```

The following example fills HS memory with NOPs.

```
*BASE
HEX
*MAP 32K LENGTH 32K HS
*BYTE 32K LENGTH 32K = 90H
*BYTE 0FC00 = 12,0AB,34,56
*BYTE 0FC00 LENGTH 4
00FC00H        12 AB 34 56                    '..4V'
```

The following example uses the single-line assembler to put the MOV instruction in memory offset 33K. The MOV instruction reads the word at FC00 into AX.

```
*SASM 33K = 'MOV AX,WORD 0FC00'
008400H        8B06 00FC
```

Note that the 8086 assembler (ASM-86) reads the instruction as MOV AX, WORD PTR 0FC00. The I²ICE single-line assembler does not recognize the assembler operator PTR. In this case, what is a correct form for the I²ICE single-line assembler is an incorrect form for ASM-86. The following example checks the entry using the ASM memory template.

```
*ASM 33K
0084000H     8B06000FC          MOV   AX,WORD PTR 0FC00H
```

Notice that the PTR operator appears when the instruction is disassembled.

The following example sets the accumulator to 0 and displays its value.

```
*AX=0;AX
0000
```

The following example defines a system register that causes a break when the program reads the word AB12 from FC00.

```
*DEFINE SYSREG evenword =
**READ AT 0FC00 IS 0AB12
```

The following example shows emulation starting from 32K using the system register *evenword*.

```
*GO FROM 32K USING evenword
*Probe 0 stopped at location 0800:0405H because of bus break
  Break register is EVENWORD   Trace Buffer Overflow
```

The following example checks AX to see whether it received the word.

```
*AX
AB12
```

The following example looks at the trace buffer in INSTRUCTIONS mode and then in CYLES mode (see Figure 3-6). Note that the word AB12 appears on the data bus once.

Memory remains the same after the last example. The following example uses the single-line assembler to modify the MOV instruction to read a word from the address FC01. This word is 34AB. The most significant byte, 34, is at FC02, and the least significant byte, AB, is at FC01.

```
*SASM 33K='MOV AX,WORD 0FC01'
0084000H     8B0601FC
*ASM 33K
0084000H     8B0601FC           MOV   AX,WORD PTR 0FC01H
```

```
*PRINT INSTRUCTIONS NEWEST 5
FRAME    ADR        BYTE              MNEMONICS  OPERANDS                UNIT 0
3F4 0083FDH         90                  NOP
3F6 0083FEH         90                  NOP
3F7 0083FFH         90                  NOP
3F9 008400H         8B0600FC            MOV  AX,WORD PTR 0FC00H
        00FC00H-DR-AB12H
3FD 008404H         90                  NOP
*
*PRINT CYCLES NEWEST 10
EXEC ADR  BUS ADR   DATA   STATUS  CLIPS FRAME   TIME          LEVEL  UNIT 0
x        b 0083FC d 9090 s 0054 CF c    f 3EF
x 0083FA b        d      s          c    f 3F0  403.2 microsecs   0
x        b 0083FE d 9090 s 0054 CF c    f 3F1
x 0083FB b        d      s          c    f 3F2  403.8 microsecs   0
x 0083FC b        d      s          c    f 3F3  404.4 microsecs   0
x 0083FD b        d      s          c    f 3F4  405.0 microsecs   0
x        b 008400 d 06BB s 0054 CF c    f 3F5
x 0083FE b        d      s          c    f 3F6  405.6 microsecs   0
x 0083FF b        d      s          c    f 3F7  406.2 microsecs   0
x        b 008402 d FC00 s 0054 CF c    f 3F8
x 008400 b        d      s          c    f 3F9  406.8 microsecs   0
x        b 008404 d 9090 s 0054 CF c    f 3FA
x        b 008406 d 9090 s 0054 CF c    f 3FB
x        b 00FC00 d AB12 s 005D DR c    f 3FC
x 008404 b        d      s          c    f 3FD  409.8 microsecs   0
x        b 008408 d 9090 s 0054 CF c    f 3FE
```

Figure 3-6 Sample Trace Buffer Displays in Both Modes for Emulation with the System Register EVENWORD

Because the word is at an odd address, there are two memory accesses. In the first access, the least significant byte, AB, appears on the upper data lines. In the second access, the most significant byte, 34, appears on the lower data lines. The following example defines an event register that causes a break when those two conditions occur.

```
*DEFINE EVTREG ODDWORD = DO
**SEM S0 IF READ AT 0FC01 IS 0ABxx
**         THEN GOTO S1
**      S1 IF READ AT 0FC02 IS 0xx34
**         THEN BREAK
**         ELSE GOTO S0
**END
```

The following example sets AX to 0.

```
*AX=0;AX
0000
```

The following example starts emulation from 32K using the event register.

```
*GO FROM 32K USING oddword
Probe 0 stopped at location 0800:0405H because of bus break
 Break register is ODDWORD  Trace Buffer Overflow
```

The following example checks AX to see whether the register received the word and then prints the trace buffer in INSTRUCTIONS and CYCLES modes (see Figure 3-7).

The following list summarizes how to handle breaks on word reads from even and odd addresses.

Assume that word AB12 is stored at the even address FC00. FC00 12 FC01 AB FC02 34 FC03 56 One memory access occurs. The data bus contains AB12.	Assume that word 34AB is stored at the odd address FC01. FC00 12 FC01 AB FC02 34 FC03 56 Two memory accesses occur. The data bus contains 12AB, then 5634

To break when the word AB12 is read:
 	
`*DEFINE SYSREG evenword =`
`**READ AT 0FC00 IS 0AB12`
`**END`
	To break when the word 34AB is read:
	
	`*DEFINE EVTREG oddword = D0`
	`**SEM S0 IF READ AT 0FC01 is 0ABXX`
	`**  THEN GOTO S1`
	`**  S1 IF READ AT 0FC02 IS 0XX34`
	`**  THEN BREAK`
	`**  ELSE GOTO S0`
	`**END`

Byte Reads From Even and Odd Addresses

Memory remained unchanged after the last example. The following example modifies the MOV instruction so that it reads a byte from FC00 into AL. The memory offset FC00 contains 12. It appears once on the lower data lines.

```
*SASM 33K = 'MOV AL,BYTE 0FC00'
0084000H     8A06000FC
*ASM 33K
0084000H     8A06000FC          MOV   AL,BYTE PTR 0FC00H
```

```
*AX
34AB
*PRINT INSTRUCTIONS NEWEST 5
FRAME    ADR        BYTE              MNEMONICS  OPERANDS                UNIT 0
3F3 0083FDH         90                  NOP
3F5 0083FEH         90                  NOP
3F6 0083FFH         90                  NOP
3F8 008400H         8B0601FC            MOV  AX,WORD PTR 0FC01H
        00FC01H-DR-AB12H   00FC02H-DR-5634H
3FD 008404H         90                  NOP
*
*PRINT CYCLES NEWEST 10
EXEC ADR  BUS ADR    DATA    STATUS   CLIPS FRAME   TIME          LEVEL  UNIT 0
x 0083FA b         d       s          c     f 3EF  402.6 microsecs   0
x        b 0083FE  d 9090  s 0054 CF  c     f 3F0
x 0083FB b         d       s          c     f 3F1  403.2 microsecs   0
x 0083FC b         d       s          c     f 3F2  403.8 microsecs   0
x 0083FD b         d       s          c     f 3F3  404.4 microsecs   0
x        b 008400  d 068B  s 0054 CF  c     f 3F4
x 0083FE b         d       s          c     f 3F5  405.0 microsecs   0
x 0083FF b         d       s          c     f 3F6  405.6 microsecs   0
x        b 008402  d FC01  s 0054 CF  c     f 3F7
x 008400 b         d       s          c     f 3F8  406.2 microsecs   0
x        b 008404  d 9090  s 0054 CF  c     f 3F9
x        b 008406  d 9090  s 0054 CF  c     f 3FA
x        b 00FC01  d AB12  s 005D DR  c     f 3FB
x        b 00FC02  d 5634  s 005D DR  c     f 3FC
x 008404 b         d       s          c     f 3FD  410.0 microsecs   0
x        b 008408  d 9090  s 0054 CF  c     f 3FE
```

Figure 3-7 Sample Trace Buffer Displays in Both Modes for Emulation Using the Even Register ODDWORD

The following example sets AX to 0 and then defines a system register that causes a break when the data bus contains 12 on the lower lines. The X's represent don't-care bits.

```
*AX = 0
*DEFINE SYSREG evenbyte =
**READ AT 0FC00 IS 0xx12
```

The following example starts emulation with this system register, displays AX, and displays the trace buffer in both INSTRUCTIONS and CYCLES mode (see Figure 3-8). Note that AX's least significant byte has the correct value.

The following example modifies the MOV instruction to read the byte from the odd address 0FC01. That byte is AB.

```
*SASM 33K = 'MOV AL,BYTE 0FC01'
008400H     8A0601FC
*ASM 33K
008400H     8A0601FC          MOV   AL,BYTE PTR 0FC01H
```

```
*GO FROM 32K USING evenbyte
*Probe 0 stopped at location 0800:0405H because of bus break
  Break register is EVENBYTE   Trace Buffer Overflow
*AX
0012
*
*PRINT INSTRUCTIONS NEWEST 5
FRAME   ADR        BYTE              MNEMONICS  OPERANDS                UNIT 0
3F4 0083FDH        90                  NOP
3F6 0083FEH        90                  NOP
3F7 0083FFH        90                  NOP
3F9 008400H        8A0600FC            MOV  AL,BYTE PTR 0FC00H
        00FC00H-DR-AB12H
3FD 008404H        90                  NOP
*
*PRINT CYCLES NEWEST 10
EXEC ADR  BUS ADR   DATA   STATUS  CLKS FRAME   TIME          LEVEL  UNIT 0
x         b 0083FC d 9090 s 0054 CF c    f 3EF
x 0083FA  b        d      s          c    f 3E0  403.0 microsecs   0
x         b 0083FE d 9090 s 0054 CF c    f 3F1
x 0083FB  b        d      s          c    f 3F2  403.6 microsecs   0
x 0083FC  b        d      s          c    f 3F3  404.2 microsecs   0
x 0083FD  b        d      s          c    f 3F4  404.8 microsecs   0
x         b 008400 d 068A s 0054 CF c    f 3F5
x 0083FE  b        d      s          c    f 3F6  405.4 microsecs   0
x 0083FF  b        d      s          c    f 3F7  406.0 microsecs   0
x         b 008402 d FC00 s 0054 CF c    f 3F8
x 008400  b        d      s          c    f 3F9  406.6 microsecs   0
x         b 008404 d 9090 s 0054 CF c    f 3FA
x         b 008406 d 9090 s 0054 CF c    f 3FB
x         b 00FC00 d AB12 s 005D DR c    f 3FC
x 008404  b        d      s          c    f 3FD  409.6 microsecs   0
x         b 008408 d 9090 s 0054 CF c    f 3FE
```

Figure 3-8 Sample Trace Buffer Displays in Both Modes for Emulation with the System Register EVENBYTE

There is one memory access. The byte appears on the upper data lines. The following example defines a system register that causes a break when that condition occurs, sets AX to 0, begins emulation, and displays the trace buffer in both INSTRUCTIONS and CYCLES mode (see Figure 3-9).

The following list summarizes how to handle breaks on byte reads from even and odd addresses.

Assume that byte 12 is stored at the even address FC00.	Assume that byte AB is stored at the odd address FC01.
FC00 12 FC01 AB FC02 34 FC03 56	FC00 12 FC01 AB FC02 34 FC03 56
One memory access occurs. The data bus contains AB12.	One memory access occurs. The data bus contains AB12.
To break when the byte 12 is read:	To break when the byte AB is read:
***DEFINE SYSREG evenbyte =** ****READ AT 0FC00 IS 0XX12**	***DEFINE SYSREG oddbyte =** ****READ AT 0FC01 IS 0ABXX**

Moving the User Cable

If you remove the user cable while the I²ICE system is running, the probe's microprocessor loses its system clock. This may require reloading the I²ICE software. At times, however, you may need to remove the user cable, such as when you transfer the user cable from the prototype system to the buffer box.

Use the UNITHOLD command before removing the user cable. Once the user cable is in place, enter any character to continue operation, as shown in the following example:

```
*UNITHOLD
Enter any character to release probes --
```

The UNITHOLD command's effect on the probe hardware is probe-specific as follows:

8086/8088 probe	causes a 3-state condition for all signal lines except ALE.
80186/80188 probe	causes a 3-state condition for all signal lines except RESET and CLK.
80286 probe	causes a 3-state condition for all signal lines except ground.

```
*DEFINE SYSREG oddbyte =
**READ AT 0FC01 IS 0ABXX
*AX=0
*GO FROM 32K USING oddbyte
*Probe 0 stopped at location 0800:0405H because of bus break
  Break register is 0DDBYTE   Trace Buffer 0verflow
*
*PRINT INSTRUCTIONS NEWEST 5
FRAME    ADR        BYTE              MNEMONICS  0PERANDS                UNIT 0
3F4 0083FDH         90                  N0P
3F6 0083FEH         90                  N0P
3F7 0083FFH         90                  N0P
3F9 008400H         8A0601FC            M0V  AL,BYTE PTR 0FC01H
        00FC01H-DR-AB12H
3FD 008404H         90                  N0P
*
*PRINT CYCLES NEWEST 10
EXEC ADR  BUS ADR    DATA   STATUS  CLKS FRAME   TIME          LEVEL  UNIT 0
x         b 0083FC d 9090 s 0054 CF c    f 3EF
x 0083FA  b        d      s         c    f 3E0  403.2 microsecs   0
x         b 0083FE d 9090 s 0054 CF c    f 3F1
x 0083FB  b        d      s         c    f 3F2  403.8 microsecs   0
x 0083FC  b        d      s         c    f 3F3  404.4 microsecs   0
x 0083FD  b        d      s         c    f 3F4  405.0 microsecs   0
x         b 008400 d 068A s 0054 CF c    f 3F5
x 0083FE  b        d      s         c    f 3F6  405.6 microsecs   0
x 0083FF  b        d      s         c    f 3F7  406.2 microsecs   0
x         b 008402 d FC01 s 0054 CF c    f 3F8
x 008400  b        d      s         c    f 3F9  406.8 microsecs   0
x         b 008404 d 9090 s 0054 CF c    f 3FA
x         b 008406 d 9090 s 0054 CF c    f 3FB
x         b 00FC01 d AB12 s 005D DR c    f 3FC
x 008404  b        d      s         c    f 3FD  409.8 microsecs   0
x         b 008408 d 9090 s 0054 CF c    f 3FE
```

Figure 3-9 Sample Trace Buffer Displays in Both Modes for Emulation with the System Register ODDBYTE

4 THE I²ICE™ SYSTEM PERSONALITY MODULES (PROBES)

This chapter introduces the three I²ICE personality modules. (The personality modules are also referred to as probes). The 8086/8088 probe emulates the 8086 and the 8088 microprocessors. The 80186/80188 probe emulates the 80186 and the 80188 microprocessors. The 80286 probe emulates the 80286 microprocessor.

In this chapter, a separate main section is devoted to each of the probes. Within each main section, there is a subsection that explains special considerations that apply to the probe.

The 8086/8088 Probe

The 8086 and 8088 microprocessors feature a large segmented memory space, a versatile instruction set, and instruction pipelining. The iAPX family includes the 8087 coprocessor, which optimizes numeric processing.

With the I²ICE pseudo-variables you can display and modify 8086/8088 registers symbolically. The following example loads the word AB12 into the AX register.

```
*AX = 0AB12
*AX
AB12
```

The following example displays the high and low bytes of the AX register.

```
*AH
AB
*AL
12
```

You can use the Boolean operators AND, OR, and XOR with these registers. The following example sets the trap flag in the FLAGS register while retaining the setting of any previous flags.

```
*FLAGS = FLAGS OR 100H
```

Alternatively, the following example sets the Boolean pseudo-variable representing the trap flag to TRUE.

```
*TFL = TRUE
```

The I²ICE system also provides pseudo-variables to display and modify 8087 registers. Refer to Chapter 5 for more information about manipulating 8087 registers.

Hardware and Software Considerations for the 8086/8088 Probe

This section describes the unique characteristics of the 8086/8088 probe. You should be aware of these characteristics when designing prototype hardware and software and when emulating your prototype.

Separate subsections are provided on the following topics:

- Address Wrap-around
- Break Information
  - — Slipping Past Instruction Breakpoints
  - — Slipping Past Breakpoints on Combined Instructions
  - — Breaking in the Middle of an Instruction
- READY Signal Set-Up Time
- Request/Grant Line
- Non-Maskable Interrupt Line and Interrupt Line
- Non-Maskable Interrupts and Program Stepping
- Synchronization between the Prototype and the Probe
- User-Accessible Test Points
  - — The SYNC START/ Test Point
  - — The 87 INT Test Point
- Coprocessor Considerations
- Inability to Break When RESET Is Asserted
- Getting a User NMI While in Emulation Mode
- Using the I²ICE™ System as a Signal Generator
- 10-MHz 8086 Probe MAX Mode Operation
- Probe MIN Mode Operation
- Address/Data Bus Float

Address Wrap-Around

The 8086/8088 microprocessor represents a virtual memory address as a selector:offset pair. The selector and the offset are each 16 bits long. The 8086/8088 microprocessor then translates that virtual address into a 20-bit physical address. A memory address in the break/trace board is 20 bits long.

As shown in Table 4-1, the difference between the way the 8086/8088 microprocessor and the break/trace board handle memory addresses causes discrepancies when wrap-arounds occur.

Table 4-1 8086/8088 Segment Boundary Increments

	8086/8088 Microprocessor	Break/Trace Board
Starting address	0:FFFFH	0FFFFH
Address incremented by 1 (next sequential address)	0:0000H (wrap-around)	10000H (no wrap-around)

Wrap-arounds do not affect bus information, but they can make break and trace information hard to follow. Address wrap-arounds can occur only when the offset is incremented past FFFF. This is not a recommended practice for any 8086/8088 application.

Break Information

A break normally occurs immediately after the target instruction executes. The following sections describe the three cases when a break is not recognized until one or more instructions after the breakpoint.

Slipping Past Instruction Breakpoints

A break sometimes slips past the specified breakpoint because the probe emulates at full processor speed and I²ICE probe hardware cannot always break on the exact instruction specified. In general, the greater the number of cycles required to execute an instruction, the lower the chances of slipping. The newest trace frame contains the last instruction executed. The break message contains the address of the next instruction to be executed.

Slipping Past Breakpoints on Combined Instructions

Although you can specify a breakpoint between parts of combined instructions, the I²ICE hardware never sees it. The following combined instructions cause slipping:

- Repeat prefixes
- LOCK prefixes
- Segment override prefixes
- MOV to a segment register
- POP a segment register
- FWAIT prefix on an 8087 instruction

Breaking in the Middle of an Instruction

In two cases the I²ICE probe can break in the middle of an instruction: WAIT and repeated string instructions. These instructions contain primitive operations or wait test cycles which can be recognized, incorrectly, as a breakpoint.

READY Signal Set-Up Time

The BTHRDY (both READY) pseudo-variable ANDs the user's processor READY signal with the 8086/8088 probe's ready signal. When BTHRDY is TRUE, the 8086/8088 probe's READY signal must be set up .3 nanoseconds before the rising edge of T2, as shown in Figure 4-1.

If the probe processor's READY signal is not set up .3 nanoseconds before the rising edge of T2, the signal is missed and the result of the logical AND is false. This causes an additional wait-state in a normally not-READY system and no wait-states in a normally READY system. Set-up time is normal when BTHRDY is FALSE.

Request/Grant Line

The internal 8087 coprocessor uses the request/grant (RQ/GT1) line. You cannot connect bus masters in a daisy chain on the RQ/GT1 line when you use an internal 8087.

Non-Maskable Interrupt Line and Interrupt Line

If a non-maskable interrupt (NMI) and an interrupt (INTR) are asserted at the same time, the I²ICE system starts to service the INTR first. This results in additional latency while the stack operations and interrupt acknowledge cycles occur. The NMI is serviced after the INTR vector and initial stack activity are complete. The INTR service is completed after the NMI is serviced.

Non-Maskable Interrupts and Program Stepping

The 8086/8088 probe ignores NMIs when stepping through a user program with the ISTEP command. The following GO command, which steps through 10 consecutive break locations, enables you to step through programs while recognizing NMIs.

```
COUNT 10
      GO TIL 0XXXX
END
```

Synchronization between the Prototype and the Probe

When the probe is executing code from high-speed (HS) memory but the user prototype expects memory with a different access time in the same address space, the user's bus control logic can get out of synchronization with the probe. A solution is to set the BTHRDY pseudo-variable to TRUE.

If the user prototype expects slow memory, another solution is to insert an appropriate number of wait-states into the HS memory accesses.

User-Accessible Test Points

The top of the buffer box has two user-accessible user test points. They are labeled SYNC START/ and 87 INT.

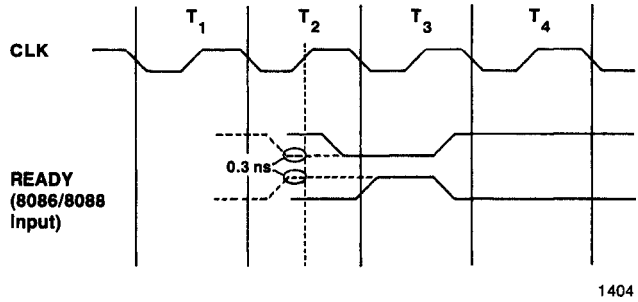


Figure 4-1 Ready Signal Set-Up Time

The SYNC START/ Test Point

The SYNC START/ test point is a TTL input that is normally high. When SYNC START/ is low and the probe enters emulation, READY is held low, and the probe undergoes a READY hang after the first instruction fetch. You can cause this hang by holding SYNC START/ low.

To synchronize the probes in a multi-probe system, first set MEMRDY to FALSE in each probe. This prevents a memory time-out from occurring during the first instruction fetch. Then, keep SYNC START/ on each probe low. Ensure that SYNC START/ for each probe goes high at the same time. This raises READY, gets rid of the READY hang, and ensures that each probe enters emulation at the same time.

A typical application is to connect all the SYNC START/ test points to one of the clipsout lines. This allows you to control the state of the SYNC START/ signal from the I²ICE console with the CLIPSOUT command.

NOTE

If, while SYNC START/ is low, a coprocessor requests the address and data buses, the probe's microprocessor will not acknowledge that request until you raise SYNC START/. This is important if the coprocessor is performing a time-critical operation.

You may also find SYNC START/ useful in a single probe system because it gives you hardware control over when the probe enters emulation. SYNC START/ must be high for emulation to begin.

The 87 INT Test Point

The 87 INT test point is a TTL output. An internal 8087 coprocessor asserts this signal. When high, this signal indicates that an unmasked exception has occurred during internal 8087 instruction execution when 8087 interrupts are enabled.

Coprocessor Considerations

When using a coprocessor with the 8086/8088 probe, be aware of the following:

- During emulation, a two-clock delay precedes each $\overline{RQ}$, $\overline{GT}$, and release pulse in MAX mode and each HOLD and HLDA assertion in MIN mode. During emulation, a user's $\overline{RQ}$ and release pulse will not be seen by the probe processor until two clock cycles after they have occurred.
- Normal MIN mode coprocessor protocol requires that HLDA become inactive before asserting the next hold.
- You can choose to run the external coprocessor only during emulation or during both emulation and interrogation. (The CPMODE pseudo-variable selects coprocessor mode.) When the coprocessor runs during interrogation mode, it may have as much as a one-microsecond delay in addition to the two-clock delay.
- The PICE system ignores a coprocessor when the probe is in the reset state. If a coprocessor asserts $\overline{RQ}$ during this time, the $\overline{RQ}/\overline{GT}$ sequence may get out of synchronization. The probe is reset while the PICE host software loads PICE probe software.
- When the CPMODE pseudo-variable is 1 (coprocessor activity in emulation mode only) and the PICE system is in interrogation mode, the 8086/8088 probe latches any pending coprocessor requests. If you do not want these pending requests honored, you must reset the external coprocessor and then the 8086/8088 probe (with the RESET UNIT command).
- Coprocessor activity is not traced when the 8086/8088 probe is in MIN mode.

Inability to Break When RESET Is Asserted

If a break occurs while RESET is high, the PICE system does not recognize the break. The break does not occur even after RESET goes low. You must reset the PICE system with the RESET UNIT command to exit this condition.

Getting a User NMI while in Emulation Mode

If a user NMI arrives while the PICE system is in interrogation mode, that user NMI is not serviced. Rather, it is latched by the PICE system and will be serviced when emulation is resumed. If you do not wish to service these latched NMIs, they can be cleared with a RESET UNIT or a UNITHOLD command before resuming emulation with the GO command.

Using the PICE™ System as a Signal Generator

You can use the 8086/8088 probe as a signal generator by connecting it to prototype hardware and mapping both memory and I/O to the emulator's internal resources. In this mode, all the control signals, status lines, address lines, and write data lines are valid even though the prototype hardware may not function correctly. The emulator runs by accessing correct data from the emulator resources and ignoring data from the prototype system. This enables you to perform microprocessor functions that the current prototype hardware does not support.

10-MHz 8086 Probe MAX Mode Operation

For the 8086 probe running at 10 MHz and in MAX mode, the user must supply a clock with a minimum low time of 60 nanoseconds. Less clock low time may cause a wrong address to be latched by ALE. If you wish to use less clock low time, delay ALE in the user system by an amount of time greater than or equal to the difference between 60 nanoseconds and the reduced clock low time that you are using. This can be done by adding a buffer to ALE in the user system.

Probe MIN Mode Operation

When performing consecutive reads to program memory, the DT/$\overline{R}$ line of the probe microprocessor (at the end of the user cable) goes high for a short time between reads. The 8086/8088 microprocessors keep DT/$\overline{R}$ low between consecutive reads. When performing consecutive I/O cycles (i.e., word I/O to an odd address), the M/IO line goes high for a short time during T4. The 8086/8088 microprocessors keep M/IO low between consecutive I/O cycles.

Address/Data Bus Float

The address/data (AD) bus is floated during T4 as follows:

Read cycle	The AD bus is floated in T4 as long as MRDC is active in T4.
INTA cycle	The AD bus is floated in T4 as long as INTA is active in T4.
Write cycle	The AD bus is never floated in T4.

NOTE

Proper use of the DEN and DT/$\overline{R}$ signals result in the normal operation (i.e., no bus contention) of both the emulator and the prototype hardware.

The 80186/80188 Probe

The 80186 and 80188 microprocessors extend the capabilities of the 8086 and 8088 microprocessors and are upwardly compatible with iAPX 86 and 88 software. They add instructions for fast index calculation, subroutine linkage, I/O data transfers, and program error detection.

The 80186 integrates a number of the most common iAPX 86 components onto a single chip. These include two independent high-speed DMA channels, a programmable interrupt controller, three programmable 16-bit timers, chip selects, peripheral chip-select logic, a programmable wait-state generator, a clock generator, and a local bus controller.

The 80186/80188 contains four interrupt pins. The I²ICE system assumes that these pins are configured either as all inputs (for fully nested mode) or as two inputs and two outputs (for cascade mode and special fully-nested mode). The outputs are interrupt-acknowledge signals, one for each of the remaining interrupt input lines.

The 80186/80188 has another alternate signal set. During reset, the 80186/80188 samples $\overline{RD}$/QSMD to determine whether it will run in standard or queue status mode. In queue status mode, the 80186/80188 provides queue status signals in place of ALE and $\overline{WR}$.

The I²ICE pseudo-variable QSTAT determines whether the 80186 probe runs in standard or queue status mode. The default is FALSE.

QSTAT = TRUE The 80186 probe runs in queue status mode.

QSTAT = FALSE The 80186 probe runs in standard mode.

With the I²ICE pseudo-variables you can display and modify 80186/80188 registers and flags. The following example displays the DX register.

```
*DX
AB12
```

Some 80186/80188 registers are represented as offsets into an internal register map. The following example displays the lower memory chip select (LMCS) register.

```
*CSCTRL(2)
38
```

The 80186/80188 probe provides debugging support for the 8087 coprocessor. The 8087 must be an external coprocessor, and the prototype must contain the 82188 coprocessor interface chip. Refer to Chapter 5 for more information about manipulating 8087 registers.

A coprocessor memory violation can occur if a HOLD/HLDA sequence occurs and the S0-S2 lines present a non-idle state with a non-USER mapped address on the bus. To prevent the coprocessor memory violation, all memory should be mapped to USER.

Hardware and Software Considerations for the 80186/80188 Probe

This section describes the unique characteristics of the 80186/80188 probe. You should be aware of these characteristics when you design prototype hardware and software and when you emulate your prototype.

Separate subsections are provided on the following topics:

- Address Wrap-around
- Break Information
  - — Slipping Past Instruction Breakpoints
  - — Slipping Past Breakpoints on Combined Instructions
  - — Breaking in the Middle of an Instruction
- Mapping Considerations for the 80186/80188 Probe
- Synchronization between the Prototype and the Probe
- User-accessible Test Points
- User Socket
- ALE Pulse Stretching

Address Wrap-Around

The 80186/80188 microprocessor represents a memory address as a selector:offset pair. The selector and the offset are each 16 bits long. A memory address in the break/trace board is a 20-bit address.

As shown in Table 4-2, the difference in memory address lengths causes discrepancies when wrap-arounds occur.

Wrap-arounds do not affect bus information, but they can make break and trace information hard to follow. Avoid wrap-arounds by not executing instructions near segment boundaries.

Break Information

A break normally occurs immediately after the target instruction executes. The following sections describe the three cases where a break is not recognized until one or more instructions after the breakpoint.

Slipping Past Instruction Breakpoints

A break sometimes slips past the specified breakpoint because the probe emulates at full processor speed and I²ICE probe hardware cannot always break on the exact instruction specified.

An extra instruction is executed when the number of bytes in the target instruction and the number of cycles required to execute that instruction match. For instance, a two-byte instruction that executes in two bus cycles causes the I²ICE probe to slip. Other combinations of instruction bytes and bus cycles can also cause a breakpoint to slip. In general, the greater the number of cycles required to execute an instruction, the lower the chances of slipping.

The newest trace frame contains the last instruction executed. The break message contains the address of the next instruction to be executed.

Slipping Past Breakpoints on Combined Instructions

Although you can specify a breakpoint between parts of combined instructions, the I²ICE hardware never detects it. The following combined instructions cause slipping:

- Repeat prefixes
- LOCK prefixes
- Segment override prefixes
- MOV to a segment register
- POP a segment register

Breaking in the Middle of an Instruction

In two cases the I²ICE probe can break in the middle of an instruction: with the WAIT command and with repeated string command instructions. These commands contain wait test cycles or primitive operations that can be recognized, incorrectly, as a breakpoint.

Table 4-2 80186/80188 Segment Boundary Increments

	80186/80188 Microprocessor	Break/Trace Board
Starting address	0:FFFFH	0FFFFH
Address incremented by 1 (next sequential address)	0:0000H (wrap-around)	10000H (no wrap-around)

Mapping Considerations for the 80186/80188 Probe

The I²ICE system can get out of synchronization with the 80186/80188 probe when the prototype system borrows slow memory or I/O from the I²ICE system and the user program directs the 80186/80188 microprocessor (through the probe) to ignore external READY (refer to the Chip Select/Ready Generation Logic specification in the chip literature). Consequently, data presented at the wrong time is incorrect.

CAUTION

Use extreme caution when mapping memory and I/O locations to any I²ICE system resource if your program ignores external READYs.

Take the following precautions if the 80186/80188 microprocessor has been programmed to ignore external READYs:

- Do not map I/O addresses programmed to ignore external READYs to ICE.
- Do not map memory ranges programmed to ignore external READYs to MULTIBUS (MB) memory.
- Use high-speed (HS) or optional high-speed (OHS) memory in a memory range that ignores external READYs only if WAITSTATE = 0.

CAUTION

User program references to an 80186/80188 internal peripheral control register cause the 80186/80188 I²ICE probe processor to complete bus cycles without wait-states and with external READY ignored, even when the 80186/80188 I²ICE probe has wait-states set. If locations that ignore external READYs are not mapped to USER or HS, the system may hang.

CAUTION

Programming 80186/80188 internal peripheral control registers can enable the 80186/80188 probe processor to complete bus cycles with an internally generated READY signal while ignoring external READY. Bus cycles may be terminated with less wait-states than allowed with external READY or set by the 80186/80188 probe. The system may hang if locations that ignore external READY are not mapped to memory with the corresponding number of wait-states.

Synchronization Between the Prototype and the Probe

When the probe is executing code from HS memory, but the user prototype expects this memory to exhibit different access time, the user's bus control logic can get out of synchronization with the probe. One solution is to set the BTHRDY pseudo-variable to TRUE.

If the user prototype expects slow memory, another solution is to insert an appropriate number of wait-states into the HS memory accesses.

User-Accessible Test Points

The top of the buffer box has two user-accessible user test points, one labeled SYNC START/ and the other TP.

The TP test point is an output that is active high when the probe is emulating.

SYNC START/ is a TTL input. The SYNC START/ test point is normally high. When SYNC START/ is low and the probe enters emulation, READY is low and the probe undergoes a READY hang when fetching the first instruction. You can cause this hang by holding SYNC START/ low.

NOTE

SYNC START/ has no effect if the address of the first instruction fetch is within a range that has been programmed through the peripheral control registers to ignore external READY.

To synchronize the probes in a multi-probe system, first set MEMRDY to FALSE in each probe. This prevents a memory time-out from occurring during the first instruction fetch. Then, keep SYNC START/ low on each probe. Ensure that SYNC START/ for each probe goes high at the same time. This raises READY, eliminates the READY hang, and ensures that each probe enters emulation at the same time.

NOTE

If, while SYNC START/ is low, a coprocessor asserts HOLD, the probe's microprocessor will not assert HLDA until you raise SYNC START/. This is important if the coprocessor is performing a time-critical operation.

NOTE

If the probe's microprocessor is waiting for READY from the user prototype, that READY must still be there when SYNC START/ goes high. When the prototype asserts its READY, the prototype must not assume that the probe's microprocessor receives that READY after a set time. Rather, the prototype must hold the READY asserted until it has determined that the probe's microprocessor has acknowledged the READY.

SYNC START/ is also useful in a single-probe system because it gives you hardware control over when the probe enters emulation. SYNC START/ must be high for emulation to begin.

User Socket

Your prototype system contains a socket into which you will connect the user cable. Intel recommends using the Textool/3M socket 268-5400. See Appendix A for instructions on connecting the 80186/80188 probe to the user prototype system.

ALE Pulse Stretching

The 80186/80188 probe generates ALE before T1. It accomplishes this by decoding the status lines during T4 and T_i. Note that generating the ALE pulse during an idle state can produce a stretched ALE pulse for the entire period when idles states are being generated.

For example, A DIV instruction can last longer than 80 idle clock cycles. The corresponding ALE pulse for the instruction following the division will be greater than 80 clock cycles.

In most cases, the long ALE pulse does not cause a problem in a user prototype system. If, however, your prototype system uses the rising edge of ALE to trigger a sequence of events or to generate logic signals, a problem can arise. The triggered logic or event will be distorted in length or time.

The 80286 Probe

The 80286 features multitasking, a large address space with four levels of protection, and high-speed compatibility with previous Intel iAPX microprocessors.

The 80286 runs in either real mode or protected mode. In real mode, the 80286 acts like an 8086. It can address up to 1M byte plus 64K of physical address space. In protected mode, the 80286 can address up to 1G byte of virtual memory per task and up to 16M bytes of physical memory.

When running in real address mode, programs developed for the iAPX 86 and iAPX 88 require minimal modification. The advantage is that these programs run up to six times faster than on the 8086. When running in protected mode, iAPX 86 and iAPX 88 programs may require slight modifications.

This section contains the following subsections that provide information on the 80286 microprocessor and the 80286 probe.

- Address Translation
  - — 8086 Address Translation
  - — 80286 Address Translation
- Multitasking
- Interrupts
- Address Protection
  - — Real Mode and PCHECK
  - — Protected Mode and PCHECK
- Memory Mapping for the 80286 Probe
- Support for Processor Extensions
- Displaying 80286 Registers and Flags
  - — Real Mode and PCHECK = TRUE
  - — Real Mode and PCHECK = FALSE
  - — Protected Mode and PCHECK = TRUE
  - — Protected Mode and PCHECK = FALSE
- Hardware and Software Considerations for the 80286 Probe

Address Translation

The 80286 probe performs 8086 address translation or 80286 address translation. When the 80286 probe is emulating in real mode, it performs 8086 translation. When the 80286 probe is emulating in protected mode, it performs 80286 translation. When the probe is not emulating, the I²ICE pseudo-variable SEL286 determines what address translation takes place.

SEL286 = TRUE 80286 address translation takes place.

SEL286 = FALSE 8086 address translation takes place.

When you load a program file that is in 80286 object module format (OMF), the I²ICE system automatically sets SEL286 to TRUE. When you load a program file that is in 8086 OMF, the I²ICE system sets SEL286 to FALSE.

8086 Address Translation

A virtual address is represented as a selector:offset pair. The selector and the offset are each 16 bits long. Each can be an expression. The selector:offset pair is of memory type POINTER.

The I²ICE system forms a physical address by shifting the selector value left by four bits and then adding the offset. The result is a 20-bit real address. With 20 bits, you can address 1M byte of memory.

If you specify a physical address, you can use 24 bits even though the 8086/8088 microprocessor addresses 1M byte of memory. With a 24-bit physical address, the I²ICE system can address 16M bytes in program memory. If you specify a pointer address, the I²ICE system constructs a 20-bit physical address.

80286 Address Translation

A virtual address is represented as a selector:offset pair. The selector and the offset are each 16 bits long. Of the 16 selector bits, 14 contain address information. The other two bits contain protection information. The complete virtual address contains 30 address bits. With 30 bits, you can address 1G byte of memory.

The 80286 address translation uses the global descriptor table (GDT) and the local descriptor table (LDT) to construct a physical address. There is only one GDT, but many possible LDTs. Both kinds of tables reside within the virtual memory space.

NOTE

The GDT cannot be indexed with a value greater than 255.

The global table descriptor register (GDTR) contains a GDT descriptor. This GDT descriptor contains the base address of the GDT. The local table descriptor register (LDTR) contains an LDT selector that is an offset into the GDT and points to an LDT descriptor.

The LDTR contains an explicit cache. When you load the selector part of the LDTR (called RLDT), the 80286 loads the LDT descriptor (pointed to by RLDT) into the LDTR cache. The LDTR is actually 64 bits long, 16 for the selector and 48 for the explicit cache.

Figure 4-2 shows the relationship of the two descriptor tables (the GDT and the LDT) and the two registers (the LDTR and the GDTR).

Of the 14 address bits in the virtual address's selector, one bit identifies either the LDT or the GDT, and the other 13 represent an index into the selected table. This index points to a segment descriptor in the descriptor table. The segment descriptor contains access rights, a base address, and the segment limit. The final physical address is the sum of this base address and the virtual address's offset.

Figure 4-3 illustrates 80286 virtual address translation.

When programming the 80286, you can either specify the selector explicity or use a segment register. The 80286 contains four segment registers. Each segment register contains a selector and an explicit cache. When you load a segment register with a selector, the 80286 also loads the explicit cache with the segment descriptor. A segment register is 64 bits long, 16 for the selector and 48 for the explicit cache. As long as the selector does not change, the 80286 does not have to access a descriptor table.

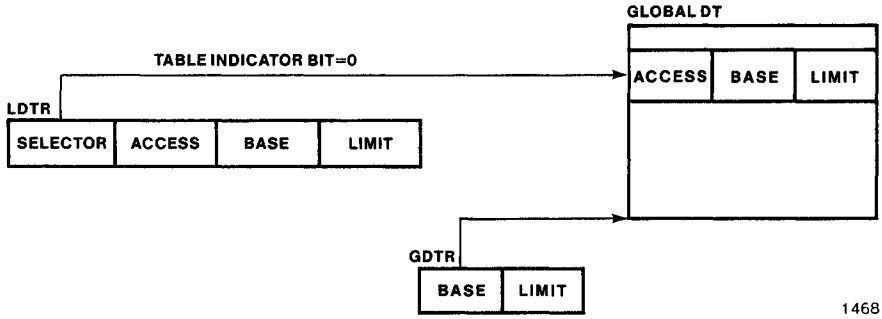


Figure 4-2 The GDT and the LDT

Figure 4-4 shows the relationship of the two descriptor tables (the GDT and the LDT) and the two registers (the LDTR and the GDTR).

When using I²ICE commands, you can represent an 80286 virtual address symbolically or by specifying the selector and offset. You can optionally include an LDT selector, as illustrated in the following syntax.

[expression-for-LDT-selector:]expression-for-selector:expression-for-offset

The brackets around the *expression-for-LDT-selector* indicate that it is optional. If you leave out the *expression-for-LDT-selector*, the I²ICE system assumes the selector in the current LDTR.

The LDT-selector:selector:offset triplet is of memory type POINTER. Debug variables defined as a triplet are displayed as a triplet. Debug variables defined as a selector:offset pair are displayed as a pair.

When you apply the POINTER memory template to program memory locations, these locations are displayed as selector:offset pairs.

The final physical address is limited to 24 bits. With 24 bits, you can address 16M bytes of memory. The virtual address is still 30 bits. The 80286 has a virtual memory space larger than its physical memory space. A bit in the segment descriptor (part of the access field) identifies whether the virtual address currently exists in physical memory.

For more information on 80286 addressing, see the entries for Address, Address protection, and Address translation in the *I²ICE™ System Reference Manual*.

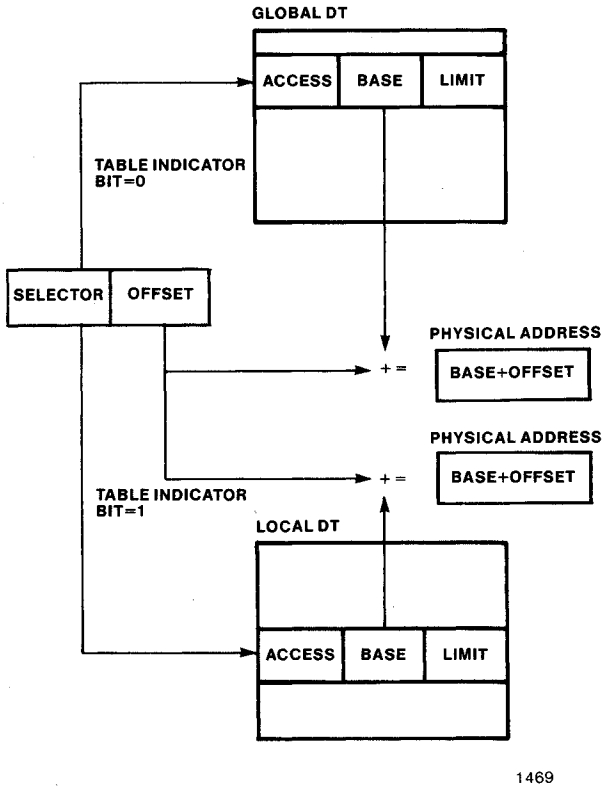


Figure 4-3 80286 Virtual Address Translation

Multitasking

The 80286 provides built-in multitasking support. A task switch operation saves the entire 80286 execution state (registers, address space, and a link to the previous task), loads a new execution state, and begins execution on the new task. This operation makes use of a task-state segment.

The GDT contains a task-state segment descriptor (TSSD). In addition to the base address of the task-state segment (TSS), the TSSD contains an access field and a limit field. The task register (TR) contains a task selector (an offset into the GDT) that points to the TSSD.

The following syntax displays the TSS.

TSS [(*expression-for-selector*)]

The TSS command without an operand displays the task-state segment whose task selector is in TR. You can override the selector currently stored in the TR by including a selector with the

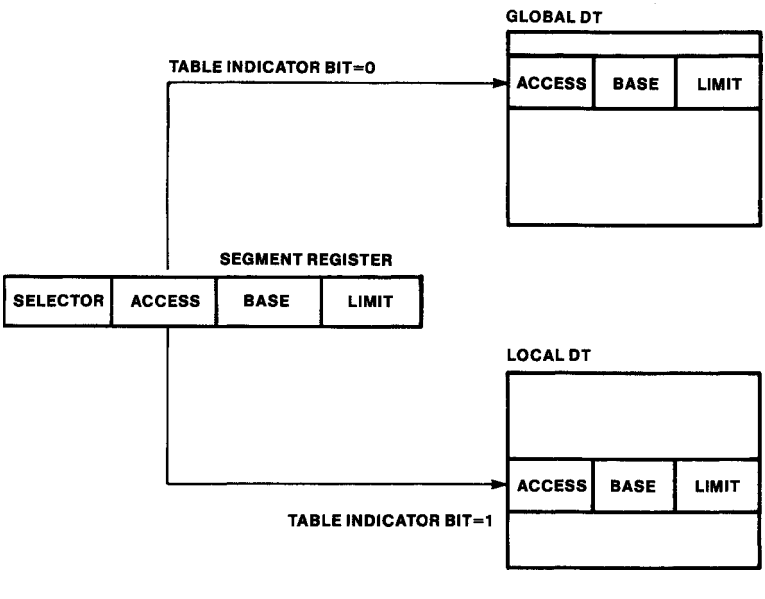


Figure 4-4 The Segment Register and the Descriptor Tables

TSS command. The I²ICE system returns an error if you choose a selector beyond the range of the GDT or one that points to an entry in the GDT that is not a TSSD.

Interrupts

The 80286 contains an interrupt descriptor table (IDT) that defines up to 256 interrupts. The interrupt descriptor table register (IDTR) contains the base address and the limit of the IDT. The relationship between the IDT and the IDTR is the same as the relationship between the GDT and the GDTR.

Address Protection

The LDT descriptors and the segment descriptors contain access bits. Two of these bits identify the descriptor protection level (the DPL). There are four protection levels: 0, 1, 2, 3. Level 0 has the most privilege, 3 the least.

The I²ICE pseudo-variable PCHECK determines whether the I²ICE system operates with 80286 protection checking on or off. The default for PCHECK is TRUE.

PCHECK = TRUE	the I²ICE system observes the 80286 protection rules when viewing and modifying 80286 registers and accepting memory addresses.
PCHECK = FALSE	the I²ICE system ignores the 80286 protection rules as much as possible.

Real Mode and PCHECK

When the 8086 is in real mode and PCHECK is TRUE, you can neither display nor alter the segment caches, the LDTR, the GDTR, the TR, and the IDTR. When the 8086 is in real mode and PCHECK is FALSE, you can display and alter the segment caches, the LDTR, the GDTR, the TR, and the IDTR.

Protected Mode and PCHECK

When the user program executes in protected mode and PCHECK is FALSE, the I²ICE system ignores the 80286 protection rules when loading the 80286 registers and accessing memory. You can display and alter the segment caches, the LDTR, the GDTR, the TR, and the IDTR.

When the user program executes in protected mode and PCHECK is TRUE, the I²ICE system obeys the 80286 protection rules when loading the 80286 registers. You can display but not alter the segment caches, the LDTR, the GDTR, the TR, and the IDTR. You must use a virtual address to access memory. The I²ICE system obeys the 80286 protection rules.

For more information on address protection and PCHECK, see the Address protection and PCHECK entries in the *I²ICE™ System Reference Manual*.

Memory Mapping for the 80286 Probe

The I²ICE system with an 80286 probe supports the standard I²ICE memory mapping features plus the GRANULARITY pseudo-variable.

With the 8086/8088 and the 80186/80188 probes, you must map HS and MB memory in blocks of 1K bytes. If you try to map a block of less than 1K bytes, the I²ICE system rounds up to the nearest 1K-byte block and prints a message telling you that it rounded the value. With the 80286 probe, however, you can map 1024 blocks of memory in either 1K blocks or 16K blocks. When GRANULARITY = 1K, the upper four address bits are ignored by mapping logic; thus, only the lower megabyte of memory is mappable. When GRANULARITY = 16K, the entire address space is available. Choose the block size by setting the GRANULARITY pseudo-variable. To change the granularity, you must have all program memory mapped to either USER or GUARDED.

The I²ICE system always maps OHS memory in blocks of 16K, even if GRANULARITY is 1K.

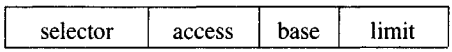
Support for Processor Extensions

The I²ICE system with an 80286 probe supports the 80287 numeric processor extension with debugging commands. With the I²ICE pseudo-variables you can display and modify 80287 registers in much the same way you display and modify 8087 registers with the 8086/8088 probe.

The 80287 requires the PEREQ and $\overline{\text{PEACK}}$ lines. The COREQ pseudo-variable determines whether the 80286 probe recognizes its PEREQ and $\overline{\text{PEACK}}$ pins.

Displaying 80286 Registers and Flags

You can access 80286 registers and flags with I²ICE pseudo-variables. This section uses the local descriptor table register as an example. The format of this register is as follows:



You can access each field separately with the following pseudo-varibles.

LDTSEL	The selector field
LDTAR	The access rights field
LDTBAS	The base address field
LDTLIM	The limit field

These pseudo-variables operate differently depending on the operating mode (real or protected mode) and the setting of the PCHECK pseudo-variable. Similar pseudo-variables exist for the other 80286 registers.

Real Mode and PCHECK = TRUE

When the 80286 probe is in real mode and PCHECK is TRUE, you cannot display or modify any of the LDTR fields. The LDTR has no meaning in real mode.

Real Mode and PCHECK = FALSE

When the 80286 probe is in real mode and PCHECK is FALSE, you can display and modify each LDTR field. The LDT has no meaning in real mode, but you can access it with I²ICE commands. Changing the selector field with the LDT pseudo-variable causes the 80286 probe to update the LDTR's explicit cache. Changing the selector field with LDTSEL does not cause the 80286 probe to update the LDTR's explicit cache.

Protected Mode and PCHECK = TRUE

When the 80286 probe is in protected mode and PCHECK is TRUE, you can display each LDTR field. The LDT has meaning when the 80286 is in protected mode. You cannot modify

the LDTR fields directly. Changing the selector field with the LDT pseudo-variable causes the 80286 probe to update the LDTR's explicit cache.

Protected Mode and PCHECK = FALSE

When the 80286 probe is in protected mode and PCHECK is FALSE, you can display and modify each LDTR field. The LDT has meaning when the 80286 is in protected mode. Changing the selector field with the LDT pseudo-variable causes the 80286 probe to update the LDTR's explicit cache. Changing the selector field with LDTSEL does not cause the 80286 probe to update the LDTR's explicit cache.

Hardware and Software Considerations for the 80286 Probe

This section describes the unique characteristics of the 80286 probe. You should be aware of these characteristics when designing prototype hardware and software and when emulating your prototype. Separate subsections are provided on the following topics:

- Hardware Slipping Past a Breakpoint
- High-Address Bits Override
- Issuing a RESET Command when an 80287 Is Present
  - — The RESET UNIT Command
  - — The RESET REGs Command
  - — The RESET ICE Command
- Resetting the 80286 Chip and the 80286 Probe
- Timing Differences between iAPX 286 and the I²ICE™ 80286 Probe
- User Substrate Capicator and +5 Volt Source
- Tracing Considerations
- User Socket
- Synchronizing Emulation to an External Event
- Using the Initialization Segment
- Reading from and Mapping to Mapped Memory or I/O
- Pascal-286 and FORTRAN-286 array size

Hardware Slipping Past a Breakpoint

The are several considerations to note regarding breakpoint slipping:

- Hardware slipping beyond an execution breakpoint occurs in one instance. If the instruction immediately preceding the instruction that causes the break results in an exception that occurs late in the execution cycle, the break may occur after the first instruction in the exception handling routine rather than after the expected instruction. You see a break at an incorrect address. The trace buffer, however, will reveal the path taken by the microprocessor.
- The 80286 probe can only break on instruction boundaries. Therefore, when the I²ICE system recognizes a data breakpoint, the currently executing instruction must complete before a break can occur.
- When the I²ICE system is emulating and a breakpoint is encountered, the trace information may include information about the next instruction after the specified break. This can be verified by examining the CS:IP registers. These registers will disagree with the last instruction in the trace buffer. The trace information is incorrect; the processor actually does break in accord with the criteria specified in the pre-emulation session.
- Unlike the 8086/8088 and 80186/80188 probes, the 80286 probe does not slip beyond a breakpoint. However, it may break prematurely if an execution breakpoint is set to occur on the instruction after a HLT instruction is executed. The break occurs at the completion of the specified instruction.
- Do not set a breakpoint on the instruction after a HLT instruction because the message displaying the cause of the break will be invalid if time-outs are enabled.

High-Address Bits Override

When you reset the 80286 microprocessor, the upper four address bits <A23-A20> remain high until the code-segment register (CS) is modified. When you set breakpoints, you may want to specify these address bits as high. Do that by preceding the address with an asterisk (*).

For example, the following commands set a breakpoint at the same address.

***GO TIL 0FFFFF0**

This command specifies a 24-bit absolute address. (The leading zero is necessary to distinguish the number from a symbol when the first digit is a letter.)

You can use a 24-bit absolute address in the following two cases:

When SEL286 = TRUE and PCHECK = FALSE
When SEL286 = FALSE

How the I²ICE commands access memory does not depend on the setting of the protection-enabled flag in the MSW.

***GO TIL *0FFFF:0**

This command specifies a virtual address.

If SEL286 = FALSE, the 80286 probe performs 8086 address translation. This results in a 20-bit physical address. The upper four address bits (<A23-A20>) are normally zero. The asterisk forces these bits high. If SEL286 = TRUE, the 80286 probe performs 80286 address translation. The result is a 24-bit physical address. The asterisk forces the upper four address bits (<A23-A20>) high.

When SEL286 = TRUE, you can also represent an address as an LDT-selector:selector:offset triplet. The asterisk forces the upper four address bits (<A23-A20>) high.

***GO TIL *RESET_VECTOR**

This command specifies a symbolic address.

Assume that the user program defines this symbolic address as 0FFFF:0 in real mode. Ordinarily, this results in the 20-bit physical address FFFF0. Address bits 23-20 are zero. The asterisk before the symbolic address forces the upper four address bits (<A23-A20>) high. The reset vector for the 80286 is FFFFF0.

Issuing a RESET Command When an 80287 Is Present

Each of the following I²ICE commands activates the RESET pin on the probe's microprocessor:

RESET UNIT
RESET REGS
RESET ICE

RESET REGS and RESET ICE always return the probe's microprocessor to real mode. Under certain circumstances (as explained in the next section), RESET UNIT may also return the probe's microprocessor to real mode.

None of these commands resets the 80287 numeric processor extension. Consequently, none of them returns the 80287 to real mode.

Running your prototype with the 80286 and 80287 in different address modes will result in invalid address translation. To return the 80287 to real mode, assert its RESET line. This must be performed by prototype hardware.

The RESET UNIT Command

When you break emulation, the I²ICE system copies the probe's registers into a register buffer. When issued from interrogation mode (from the * prompt), the I²ICE command RESET UNIT resets the probe's microprocessor but does not affect the register copies. When you

resume emulation, the I²ICE system reloads the probe's registers with the values in the register buffer. Consequently, a RESET UNIT issued from interrogation mode does not appear to have any effect on a user program.

When issued from interrogation mode, a RESET UNIT does not return the probe's microprocessor to real mode.

When issued from emulation mode (from the ? prompt), a RESET UNIT causes a break and returns the registers and their copies to the values they had just before the probe began emulating.

Figure 4-5 shows returning the probe's microprocessor to real mode.

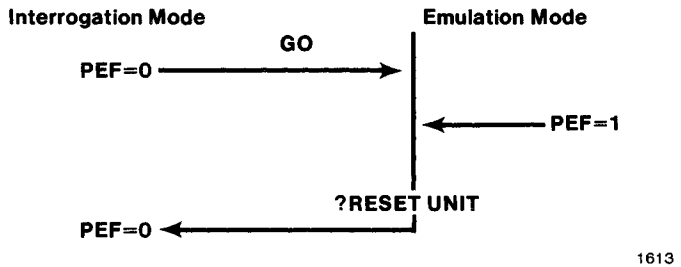


Figure 4-5 Returning the Probe's Microprocessor to Real Mode

The RESET REGS Command

The I²ICE command RESET REGS asserts the RESET pin on the probe's microprocessor. The RESET REGS command also clears the segment registers, clears the MSW, and loads the CS:IP with the reset vector. It does this both for the probe's actual registers and for their copies in the register buffer. Hence, entering the RESET REGS command returns the probe microprocessor to real mode.

The RESET ICE Command

The I²ICE command RESET ICE asserts the RESET pin on the probe's microprocessor and reloads probe software. The RESET ICE command also clears the segment registers, resets the pseudo-variables displayed by the STATUS command (BTHRDY, BUSACT, COENAB, COREQ, CPMODE, IORDY, MEMRDY, PCHECK, RSTEN, SEL286, TRCBUS) to their default values, clears the MSW, and loads the CS:IP with the reset vector. It does this for the probe's actual registers and for their copies in the register buffer. Hence, entering the RESET ICE command returns the probe microprocessor to real mode.

The RESET ICE command also returns the AX, BX, CX, and DX registers, the stack and base pointers, and the source and destination indexes to the same values that the 80286 registers have after reset.

Resetting the 80286 Chip and the 80286 Probe

The system clock provides the fundamental timing for the 80286 system. It is divided by two inside the 80286 microprocessor to generate the processor clock (PCLK). The 80286 probe also generates a signal called PCLK, as may the target system. The 80286 microprocessor's internal divide-by-two circuitry can be synchronized to an external clock generator by a low-to-high transition on the RESET input to the 80286 microprocessor. Asynchronous PCLK phasing may cause bus contention between the target system and the I²ICE system. The following three conditions can cause the phase of PCLK in the 80286 probe to differ from PCLK in the target system.

- Reset from the host (RESET ICE command, RESET UNIT command, UNITHOLD command, and when an auto reset is requested).
- User hardware reset while the probe is in interrogation mode or when the RSTEN pseudo-variable is FALSE.
- User system power-on reset.

If PCLK synchronization is a problem, use one of the following solutions.

- Design the PCLK generation circuit in the user system to be synchronized with status bits rather than RESET. (The 82284 clock generator uses the status lines $\overline{S0}$ and $\overline{S1}$ to synchronize its PCLK output.)
- Before starting a debug session, enter emualtion and then reset the target system to synchronize the PCLKs. The PCLKs will remain synchronous until a RESET UNIT command, a RESET ICE command, or a target system reset with the probe in interrogation mode. If one of these conditions occurs, enter emulation and reset the target system to restore synchronization.

When the 80286 probe is reset, the address and data buses are in a 3-state condition for the full duration of the reset.

Timing Differences Between the iAPX 286 and the 80286 Probe

There are timing differences between the ways that the I²ICE 80286 probe and the 80286 chip handle RESET and HOLD/HLDA. See the I²ICE data sheet for an explanation of these differences.

User Substrate Capacitor and +5 Volt Source

The prototype hardware need not supply a substrate capacitor or +5 volts to the 80286 probe.

Tracing Considerations

If the 80286 probe releases the memory bus to a coprocessor (the ADMA 82258), it does not record that coprocessor's bus activity in the trace buffer. Because the 80287 processor extension uses the memory protection capabilities of the 80286, the 80286 probe does record 80287 instruction execution and bus activity in the trace buffer.

Cascade interrupt addresses are not recorded in the trace buffer. Connect the chips module to the bus in the target system if you need this data.

User Socket

Your prototype system contains a socket into which you will connect the user cable. Intel recommends using the Textool/3M socket 268-5400. See Appendix A for instructions on connecting the 80286 probe to the user-prototype system.

Synchronizing Emulation to an External Event

The top of the buffer box has a user-accessible test point labeled SYNC START/. SYNC START/ is useful because it gives you hardware control over when the probe enters emulation.

The SYNC START/ test point is a TTL input that is pulled up with a 1K resistor. The SYNC START/ input is normally high. When SYNC START/ is driven low and the GO command is executed, the probe treats SYNC START/ as an auxiliary hold input (HLDA will not be activated). HOLD is asserted and the 80286 no longer has access to the address and data buses.

To synchronize the probes in a multiple-probe system, first set the BUSACT and MEMRDY pseudo-variables to FALSE in each probe. This prevents a bus time-out from occurring when the 80286 microprocessor loses control of the address and data buses. Then, keep SYNC START/ on each probe low and ensure that SYNC START/ for each probe goes high at the same time. This lowers HOLD (de-asserts it) and returns the address and data buses to the 80286 microprocessor, allowing emulation to proceed. In this way you can ensure that each probe enters emulation at the same time.

Because SYNC START/ gives you hardware control over when the probe enters emulation, it is also useful in a single-probe system.

Using the Initialization Segment

The 80286 addresses memory with 24 address lines. After a reset, the 80286 is in real mode and the most significant four bits of the address is high. The 80286 then addresses a 64K-byte segment (called the initialization segment) at the top of the 16M-byte physical address space. After the first long jump (which modifies the CS register), the most significant four bits go low.

The result is that right after a reset, the 80286 temporarily has access to memory locations in addition to the 1M byte provided in real mode.

A problem arises if you try to load the initialization segment with the I²ICE system 8086 loader (your program is an 8086 OMF). Protected-mode users should construct 80286 OMFs and hence use the 80286 loader.

If you still need to load the initialization segment with the 8086 loader, one method is to first map the initialization segment to where you want the initialization code to reside. Then load the initialization code into HS memory mapped within the lower 1M byte. Move your initialization code with the BYTE command. The following example assumes that your initialization code takes up no more than 1K byte.

```
*MAP 0 LENGTH 1K HS
*MAP 1023K LENGTH 1K USER
*LOAD init.86
*BYTE 1023K LENGTH 1K = BYTE 0 LENGTH 1K
```

Another method is as follows. If the granularity is 1K, you can load the initialization segment by mapping the 64K-byte segment at the top of the 1M-byte address space to HS and then entering the LOAD command. When the granularity is 1K, the I²ICE memory map has address wrap-around. Each physical memory location responds to 16 different addresses. If you map the 64K-byte segment from 1M – 1 to 1M – 64K to HS, you are also mapping the 64K-byte segment from 16M – 1 to 16M – 64K (the initialization segment) to HS, the 64K-byte segment from 15M – 1 to 15M – 64K to HS, etc., to the same physical locations.

Reading from and Mapping to Mapped Memory or I/O

When you map memory to HS or MB memory or when you map I/O ports to ICE, the bus to the prototype is also active.

For example, assume that you map the first 32K bytes of memory to HS. When you read from one of these memory locations, you also read from user memory. The I²ICE system, however, ignores the read data. When you write to one of these memory locations, you write to both the HS location and the USER location. Even though the memory locations are mapped to I²ICE memory, the user memory corresponding to those locations does not remain unchanged.

The same holds true for I/O ports. If you write to an I/O port mapped to ICE, the write data appears on the screen or as an I²ICE variable in a debug procedure as well as at the user I/O port (if it exists).

Pascal-286 and FORTRAN-286 Array Size

The I²ICE system does not support Pascal-286 and FORTRAN-286 array size greater than 64K.

5 COPROCESSOR SUPPORT

A coprocessor is a microprocessor that enhances the functions of the CPU.

The 8087 and 80287 numeric data processors (NDP) perform arithmetic and comparison operations on a variety of numeric data types as well as executing numerous built-in transcendental functions.

The 8089 input/output processor (IOP) performs input and output operations for the CPU. An IOP combines the attributes of a CPU with those of a flexible DMA controller.

The distinction is sometimes made between a coprocessor and a processor extension. A coprocessor can gain control of the memory and data buses, plus it uses the HOLD and HLDA signals. A processor extension uses the PEREQ and $\overline{\text{PEACK}}$ signals. That makes the 80287 a numeric processor extension, while the 8087 and 8089 are coprocessors.

Mapping Restrictions When Using Coprocessors

When the 8087 is used as an external coprocessor (with the 8086/8088 and 80186/80188 probes), memory accessed by the external 8087 must be mapped to USER. For the 8089, all program memory accessed by the 8089 must be mapped to USER.

The PHANG Pseudo-Variable

Use the PHANG pseudo-variable to detect 8086/8088 and 80186/80188 coprocessor hangs (the PHANG pseudo-variable is not supported on the 80286 probe). Setting PHANG to TRUE causes a system time-out when the coprocessor has been granted the bus and remains inactive for longer than one second. The default is TRUE.

The 8087/80287 Numeric Data Processors

The 8086/8088 and the 80186/80188 probes provide debugging support for a prototype system containing the 8087. The 8086/8088 probe supports an 8087 as an internal or an external coprocessor; the 80186/80188 supports an 8087 as an external coprocessor. When an 8087 is used with the 80186/80188 probe, it requires the 82188 coprocessor interface chip.

The 80286 probe processor provides debugging support for a prototype system containing the 80287.

The I²ICE commands provide access to the 8087's and 80287's stack, status registers, and flags. The I²ICE system's disassembly and trace features extend to 8087 and 80287 instructions and data types.

How the I²ICE system treats the 8087 depends on the two pseudo-variables COENAB and CPMODE and on the GET87 command. How the I²ICE system treats the 80287 depends on the pseudo-variables CPMODE and COREQ.

The following sections introduce the COENAB, CPMODE, and COREQ pseudo-variables and the GET87 command. Table 5-1 summarizes the interactions between the COENAB, CPMODE, and COREQ pseudo-variables. The following subsections describe further the use of the COENAB and CPMODE commands. See also the *I²ICE™ System Reference Manual* (and the section in Chapter 4 of this manual entitled "Issuing a RESET Command when an 80287 Is Present") for further information on the coprocessor commands.

Table 5-1 Coprocessor Pseudo-variable Interaction

	CPMODE = 1	**CPMODE = 2**
COREQ = T	PEREQ/$\overline{\text{PEACK}}$ are recognized only during emulation.	PEREQ/$\overline{\text{PEACK}}$ are recognized during both emulation and interrogation.
COREQ = F	PEREQ/$\overline{\text{PEACK}}$ off.	PEREQ/$\overline{\text{PEACK}}$ off.
COENAB = T	HOLD/HLDA or $\overline{\text{RQ}}$/$\overline{\text{GT}}$ are recognized only during emulation.	HOLD/HLDA or $\overline{\text{RQ}}$/$\overline{\text{GT}}$ are recognized during both emulation and interrogation.
COENAB = F	HOLD/HLDA or $\overline{\text{RQ}}$/$\overline{\text{GT}}$ off.	HOLD/HLDA or $\overline{\text{RQ}}$/$\overline{\text{GT}}$ off.

The COENAB Pseudo-Variable

The coprocessor enable (COENAB) pseudo-variable enables or disables an external coprocessor. The COENAB pseudo-variable has the value TRUE or FALSE. Setting COENAB to TRUE enables an external coprocessor, and the probes respond in the following ways.

8086/8088 probe	recognizes its request/grant lines (MAX mode) or hold/hold acknowledge lines (MIN mode).
80186/80188 probe	recognizes its hold/hold acknowledge lines.
80286 probe	recognizes its hold/hold acknowledge lines.

Setting COENAB to FALSE disables an external coprocessor, and the probes ignore their request/grant or hold/hold acknowledge lines.

For the 8086/8088 and 80186/80188 probes, COENAB's default value is TRUE. For the 80286 probe, COENAB's default value is FALSE. Note that, for the 8086/8088 and 80186/80188 probes, when the I²ICE software is invoked, it checks the request/grant or hold/hold acknowledge lines before the COENAB pseudo-variable's default value is set.

COENAB and an External Coprocessor

For an 8086/8088 or 80186/80188 probe, if COENAB is FALSE and your program contains a coprocessor instruction, the I²ICE system will hang. The coprocessor does not run, but the coprocessor instruction still causes the I²ICE probe to wait for an acknowledgement. If the I²ICE system hangs in this way, recover by resetting first the coprocessor (with the hardware reset button) and then the probe (with the RESET UNIT command).

COENAB and an Internal Coprocessor

If you use an internal coprocessor with your 8086/8088 probe, always set COENAB to TRUE. If COENAB is FALSE, the internal coprocessor still runs, but the coprocessor may get out of synchronization with the probe.

The CPMODE Pseudo-Variable

Use the CPMODE pseudo-variable to select external coprocessor modes before starting emulation; CPMODE has no effect on an internal coprocessor. The CPMODE pseudo-variable has the value 1 or 2. For the 8086/8088 and 80186/80188 probes, the default is 1. For the 80286 probe, the default is 2.

When CPMODE is 1 and COENAB is TRUE, the probe's microprocessor recognizes its request/grant or hold/hold acknowledge lines only during emulation. For the 8086/8088 and 80186/80188 probes, if a request is issued while the probe is not emulating, the request is stored and the grant occurs as soon as emulation begins. When CPMODE is 2 and COENAB is TRUE, the probe's microprocessor recognizes its request/grant or hold/hold acknowledge lines at any time.

For the 8086/8088 and 80186/80188 probes, CPMODE also determines how the I²ICE system treats modifications to the 8087 registers made while the I²ICE system is in interrogation mode. After you enter the GET87 command, the I²ICE system maintains copies of the 8087 registers in a register buffer. When you read and write 8087 registers, you read and write the copies in this buffer.

If you resume emulation after modifying one or more 8087 registers and CPMODE is 2, the I²ICE system loads the actual 8087 registers with the modified values from the register buffer (which is external emulation memory). If you resume emulation after modifying 8087 registers and CPMODE is 1, the I²ICE system does not load the actual 8087 registers with the modified values from the register buffer. Emulation resumes with the old values.

The COREQ Pseudo-Variable

The COREQ pseudo-variable enables or disables an external numeric extension and is specific to the 80286 probe. When COREQ = TRUE, the 80286 probe recognizes its PEREQ and PEACK lines. Setting COREQ = FALSE disables the external numeric extension, and the 80286 probes does not recognized the PEREQ and PEACK lines.

The GET87 Command

The GET87 command is specific to the 8086/8088 and 80186/80188 probes. You must enter the GET87 command to tell the I²ICE system that an 8087 coprocessor is present. The 8087 must be enabled (COENAB must be TRUE) for the I²ICE system to execute the GET87 command.

The following restrictions apply when you use the GET87 command with an external coprocessor:

- The probe must not be emulating and CPMODE must be 2.
- Include with the GET87 command the starting address of a 110-byte buffer. The I²ICE system uses this area as an intermediate buffer in saving and restoring the 8087 register data. The original contents of this buffer are preserved between save and restore operations. User's data is restored in the buffer after the GET87 command is entered. This buffer must be mapped to USER and not specified as read-only.

The following restrictions apply when using the GET87 command with an internal coprocessor (8086/8088 probe only):

- The probe must not be emulating. The value of CPMODE is not significant.
- Do not include an address with the GET87 command. The I²ICE system uses reserved system memory rather than USER memory for the register buffer. If you do include an address, the I²ICE system ignores it.
- The I²ICE system always loads the actual 8087 registers with the values from the buffer when you resume emulation after modifying one or more 8087 registers.

6 MULTIPLE-PROBE SYSTEMS

This chapter describes the operation of a multiple-probe I²ICE system. It introduces the I²ICE commands that control the system break and trace lines and explains how to arm the I²ICE system, assert the system break and trace lines, enable a unit, and write debug procedures for use with multiple probes

I²ICE™ System Units

Up to four I²ICE chassis may be daisy-chained together in an I²ICE system. Each chassis is called a unit and has a unit number from 0 to 3. Each unit may have its own probe and can communicate with one or more prototype systems and with any other unit.

By default, the current unit is the unit closest to the host that has a probe attached. (An I²ICE chassis may have the iLTA without a probe.) The unit numbers increase as you move away from the host. The next chassis is unit 1, then unit 2, and finally unit 3.

The I²ICE system commands operate on the current unit. The following example changes the current unit with the UNIT pseudo-variable.

```
*UNIT = 2
```

You can temporarily change the current unit for the duration of a command with a backslash. The following example displays the modules on unit 1 and on unit 0 (the current unit).

```
*\1 dir module; dir module
```

Units communicate using the system break (SYSTRIG) and system trace lines (SYSTRACE). When the SYSTRIG line enters a unit, it becomes the local signal SYSBREAKIN. When the SYSTRACE line enters a unit, it becomes the local signal SYSTRACEIN. Figure 6-1 illustrates the system break and trace lines in a multiple-probe configuration.

For a unit to assert SYSTRIG, the I²ICE system must first be armed. The I²ICE system does not need to be armed to assert SYSTRACE. For a unit to recognize an assertion of either of the system lines (allow SYSBREAKIN or SYSTRACEIN to be asserted), that unit must be enabled.

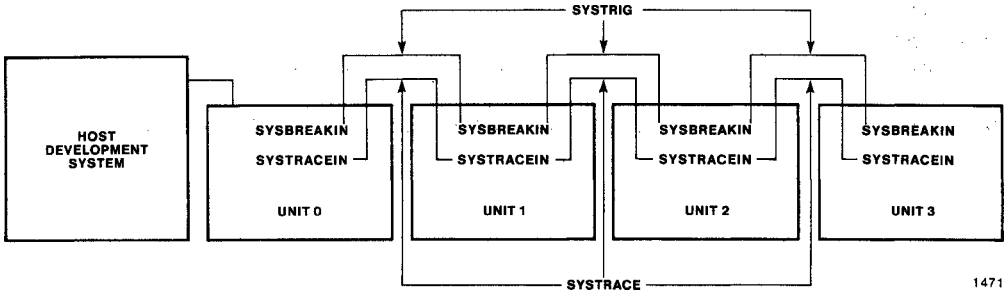


Figure 6-1 A Multiple-Probe I²ICE™ System

Arming the I²ICE™ System

The I²ICE system is armed by default. You can arm and disarm the I²ICE system with the following:

- The SYSTEM ARM/DISARM command.
- An event specification—you can specify the keyword SYSARM in an event register or in the GO command.
- A system specification—you can specify the keyword SYSARM in a system register or in the GO command.

You cannot arm and disarm the system with a break specification. You arm the I²ICE system, not individual probes. When the system is armed, all probes are armed.

The following example shows how to disarm the I²ICE system with the SYSTEM ARM/DISARM command.

```
*SYSTEM DISARM
```

The following example shows how to arm the I²ICE system with a system specification that is stored in a system register called arm0.

```
*DEFINE SYSREG arm0 = SYSARM AT :cmaker#12
*SYSTEM DISARM
*GO USING arm0
```

Before beginning emulation with an event or system specification that arms the I²ICE system, disarm the system with the SYSTEM DISARM command.

Arming the I²ICE system is not the same as arming a breakpoint. To arm a breakpoint, use an arm specification. Include the arm specification in an arm register or as part of the GO command. Refer to Chapter 3 for a discussion of arm specifications and arm registers.

Asserting the System Break and Trace Lines

You can assert the system break line with an event specification or a system specification. You can specify the keyword SYSTRIG (for asserting the system break line) or the keyword SYSTRACE (for asserting the system trace line) in an event register, system register, or in the GO command.

You can assert the system trace line with a trace specification. You can specify the keyword SYSTRACE in a trace register or as part of the GO command.

The following example shows how to assert the system trigger line with a system specification. This system specification is stored in a system register called trig1.

```
*SYSTEM ARM
*DEFINE SYSREG trig1 = SYSTRIG AT :cmaker#22
*GO USING trig1
```

The emulating unit asserts the system trigger line when the user program begins to execute statement #22. Whether asserting the system trigger line causes a unit to break depends on whether the unit is enabled. If the system trigger line is asserted, then any unit that has its system break line enabled will break.

When you use SYSTRACE in a multiprobe environment with various probe frequencies, the slower probes may miss the system trace event for one instruction. Therefore, specify a range of addresses, such as one of the following:

SYSTRACE AT OUTSIDE *address-start* LENGTH 50

SYSTRACE AT X0X110XY

Enabling I²ICE™ System Units

By default, each unit is enabled for both the system break line and the system trace line. You can enable and disable each line separately. First set the current unit to the unit you want enabled and then enter the ENABLE command. The following example shows how to enable the system break line for unit 1.

```
*UNIT = 1
*ENABLE SYSBREAKIN
```

The local signal corresponding to SYSTRIG is SYSBREAKIN.

The previous example changed the current unit to 1. The current unit remains 1 until you explicitly change it back. With the backslash, you can change the current unit only for the duration of the command.

The following example deals with two units. Unit 0 arms the system when its program executes statement #12. Unit 1 triggers the system when its program executes statement #22. Unit 0 and unit 1 recognize the system trigger line and break when it is asserted.

```
*\0 ENABLE SYSBREAKIN
*\1 ENABLE SYSBREAKIN
*DEFINE SYSREG arm0 = SYSARM AT :cmaker#12
*DEFINE SYSREG trig1 = SYSTRIG AT :dmaker#22
*\0 GO USING arm0
*\1 GO USING trig1
```

Symbolic Support for Multiple Probes

The I²ICE system provides symbolic support for each of its probes in a multiple-probe system. You can refer to program variables by name in each unit. The program symbol table resides in the host development system.

The following example loads unit 0 with a program called prog0 and unit 1 with a program called prog1.

```
*UNIT = 0
*LOAD :F1:prog0
*\1 LOAD :F1:prog1
```

Assume that prog0 contains a program variable called pvar0 and that prog1 contains a program variable called pvar1. The following example sets pvar0 equal to pvar1.

```
*\0 :prog0.pvar0 = \1 :prog1.pvar1
```

The following example shows that if the current unit is 0, you need specify only the \1, not the \0.

```
*:prog0.pvar0 = \1 :prog1.pvar1                    /*current unit is 0*/
```

If the current unit is 1, you must specify both the \0 and the \1 or verify that the first variable belongs to the current unit. For example, when the I²ICE system sees the following assignment:

```
*\0 :prog0.pvar0 = :prog1.pvar1
```

the I²ICE system looks for both program variables on unit 0. The following two assignments are equivalent.

```
*UNIT
001
*\0 :prog0.pvar0 = \1 :prog1.pvar1
*:prog1.pvar1 = \0 :prog0.pvar0
```

The following example illustrates inter-probe communication. Assume that unit 0 and unit 1 are emulating a program (prog0 for unit 0 and prog1 for unit 1) and that you want unit 0 to break when unit 1 writes a variable called pvar1. You do not want unit 1 to break.

```
*SYSTEM ARM
*\0 ENABLE SYSBREAKIN
*\1 DISABLE SYSBREAKIN
*\0 GO FOREVER
*\1 GO TIL SYSTRIG WRITE AT .pvar1
```

Writing Debug Procedures for Multiple Probe Systems

Debug procedures that handle multiple probe systems must know which probes are emulating. The WAIT function returns the unit number of the probe breaking emulation. Once the break has occurred and if no probe is emulating, the WAIT function returns the value 255T.

The following example assumes that probe 0 and probe 1 are both emulating. Probe 0 breaks. WAIT then returns the value 0. WAIT continues to return 0 until probe 1 breaks. WAIT returns the value 1 when probe 1 breaks; once the break has occurred, WAIT returns the value 255T because no probe is emulating.

```
*UNIT = 0
*\1 GO USING sys1
*GO USING sys1
?WAIT
```

Note that the prompt is determined by the current unit. The prompt is ? when the current unit is emulating. When unit 0 breaks, the I²ICE system returns its unit number.

```
0
*WAIT
```

When the probe (unit 1) breaks, the I²ICE system returns its unit number. Entering WAIT returns the value 255, indicating that emulation is not occurring.

```
1
*WAIT
255
```

The following debug procedure sets the current unit to 0, then sends unit 0 into emulation. Unit 0 will emulate until the user program writes location 0031:0010H. The procedure then sets the current unit to 1 and waits until unit 0 breaks before sending unit 1 into emulation.

```
*DEFINE PROC bothrun = DO
. *UNIT = 0
. *GO TIL WRITE AT 0031H:0010H
. *UNIT = 1
. *REPEAT
```

```
. . *IF (WAIT = =0) OR (WAIT = =255T) THEN
. . . *GO TIL WRITE AT 0031H:000EH
. . . *END
. . *END
. *END
```

WAIT is also useful in single-probe systems when, for example, the program runs for a long time before breaking. The following debug procedure starts emulation, waits until the break, and then prints out the microprocessor registers.

```
*DEFINE PROC longbreak = DO
. *GO TIL WRITE AT .pvar1 IS 25T
. *REPEAT
. . *UNTIL WAIT = =255T
. . *END
. *REGS
. *END
```

Without the REPEAT loop, the I²ICE system would try to execute the REGS command during emulation and return an error message.

The CAUSE command displays the reason for the last emulation break. The CAUSE message lists the location and reason for the break along with the unit number, the debug register that caused the break, the value of the chips, and trace buffer overflow (if applicable). The following example displays the reason emulation stopped.

```
*CAUSE
Probe 0 stopped at :CMAKER#10 because of guarded access
  Bus address = 008274
```

Synchronization Between Units

Some applications require that different probes begin emulating at the same time. If you start two probes emulating by changing the unit number and entering the GO command, the first probe begins emulating before the second. To synchronize the probes, you need to use the SYNC START/ test point available through a small opening in the side of the probe buffer box. This test point is an input to the probe.

The 8086/8088 and 80186/80188 Probes

SYNC START/ is normally high. When SYNC START/ is low and the probe enters emulation, READY is low, and the probe undergoes a READY hang when fetching the first instruction. You can cause this hang by holding SYNC START/ low.

To synchronize the probes in a multiple-probe system, first set the MEMRDY pseudo-variable to FALSE in each probe. This prevents a memory time-out from occurring during the first instruction fetch. Then, keep SYNC START/ on each probe low. Ensure that SYNC START/ for each probe goes high at the same time by connecting each SYNC START/ test point to one

of the two clipsout lines. These are output from the I²ICE system and are initially low. Then, enter the GO command for each probe. Each probe hangs after the first instruction fetch. Then, set the clipsout line to 1. All the probes enter emulation simultaneously.

The 80286 Probe

SYNC START/ is normally high. When SYNC START/ is low and the probe enters emulation, the probe treats SYNC START/ as an auxiliary hold input (HLDA will not be activated). HOLD is asserted and the 80286 no longer has access to the address and data buses.

To synchronize the probes in a multiple-probe system, first set the BUSACT and MEMRDY pseudo-variables to FALSE in each probe. This prevents a bus time-out from occurring when the 80286 microprocessor loses control of the address and data buses. Then, keep SYNC START/ on each probe low and ensure that SYNC START/ for each probe goes high at the same time. This lowers HOLD (de-asserts it) and returns the address and data buses to the 80286 microprocessor, allowing emulation to proceed. In this way you can ensure that each probe enters emulation at the same time.

A I²ICE™ SYSTEM NON-HOST HARDWARE INSTALLATION

The hardware installation procedure for the I²ICE system depends on the host development system, the number of I²ICE chassis in the system, and the system options. Begin installation of your I²ICE system using the installation instructions in this appendix. The appendix has sections on the following host-independent topics.

- Installing the I²ICE instrumentation chassis
- Installing the emulation base module
- Installing personality modules and user cables
- Installing the emulation clips module
- Installing the iLTA logic timing analyzer option
- Host installation information

The I²ICE™ System Instrumentation Chassis Installation

Refer to Figures A-1, A-2, and A-3 when installing the instrumentation chassis. Perform the following steps for each instrumentation chassis in your I²ICE system.

1. Unpack the chassis and set it in the desired location.

CAUTION

Provide at least six inches (15 centimeters) of clearance on all sides to ensure proper cooling.

2. Set the line voltage switch to the appropriate value for your area: 120 for 90–132 VAC operation or 240 for 180–264 VAC operation. (Use a small screwdriver as a lever.)

WARNING

Changing the power cord involves hazardous voltages and currents. Only qualified personnel should change the power cord.

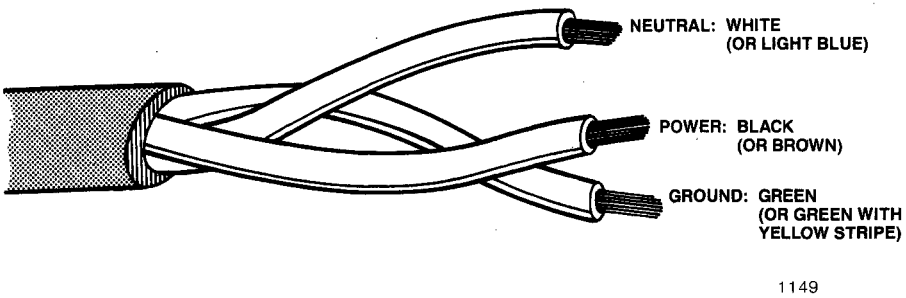


Figure A-1 Power Cable

3. If the system location is in an area where the power outlets do not match the power plug on the instrumentation chassis power cord, remove the power plug and install the appropriate power connector.

   Refer to Figure A-1 when installing a new power connector.

   Do not connect the I²ICE system to the power outlet yet.

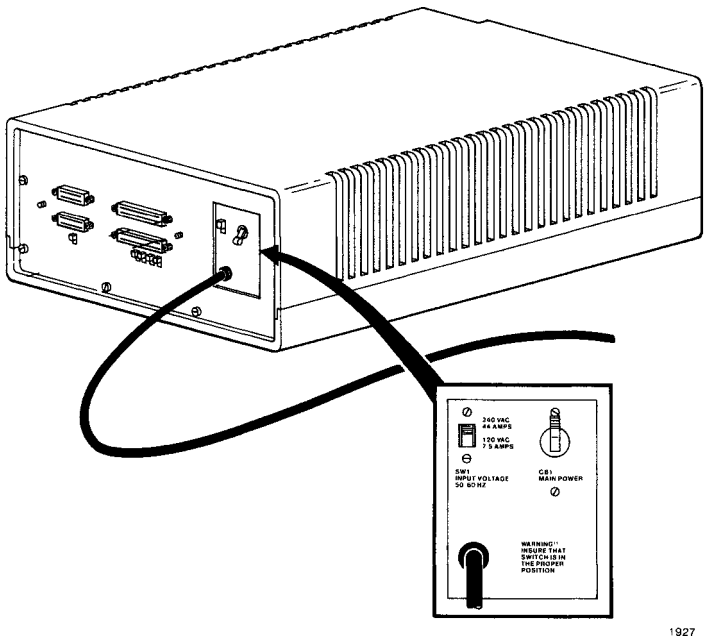


Figure A-2 Circuit Breaker on the Rear Panel of the I²ICE™ System Instrumentation Chassis

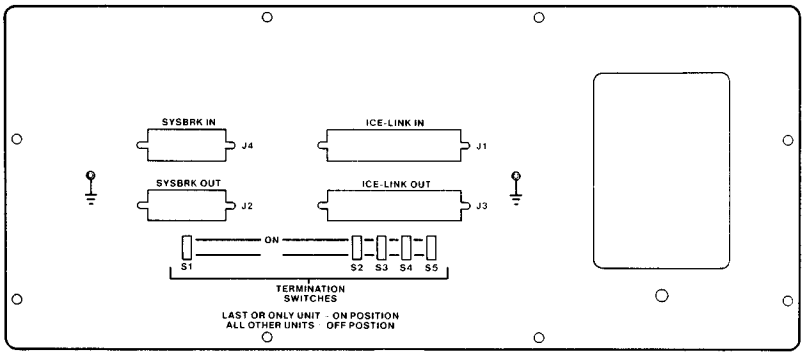


Figure A-3 Termination Switches on the Rear Panel of the I²ICE™ System Instrumentation Chassis

CAUTION

To avoid overloading the line circuit, ensure that each I²ICE instrumentation chassis is on a separate line circuit and that the host development system is on a separate line circuit.

The I²ICE chassis draws a maximum of 12 amps at 90–132 V and 6 amps at 180–264 V. A Series III development system with two double-density disk drives and an expansion chassis draws a maximum of approximately 10.5 amps at 90-132 V.

4. Verify that the circuit breaker CB1 (see Figure A-2) on the instrumentation chassis rear panel is OFF. Connect the line cord to an appropriate power outlet.

5. For single-chassis I²ICE systems, ensure that all five termination switches are ON (up) (see Figure A-3).

   For multiple-chassis I²ICE systems, ensure that the last (the highest-numbered) instrumentation chassis has all five termination switches ON (up). All other chassis in a multiple-chassis I²ICE system must have all termination switches OFF (down) (see Figure A-3).

6. Remove the front panel from the instrumentation chassis and set it aside.

CAUTION

Remove the foam packing behind the front panel (and save it for future storage and shipment). Powering up the I²ICE instrumentation chassis without removing the foam will damage the equipment.

Emulation Base Module Installation

The emulation base module consists of a break/trace board, a map-I/O board, the buffer base assembly, and an emulation clips module. The break/trace board, the map-I/O board, and the buffer base assembly are already installed. Figure A-4 shows the location of the break/trace board and the map-I/O board in the I²ICE instrumentation chassis.

For your reference, the map-I/O board is jumpered at E1-E2 and E3-E4 (see Figure A-5).

In this section one installation task is described: checking buffer box jumpering.

Buffer Base Assembly Jumpering

Ensure that the buffer board is jumpered correctly (see Figure A-6).

E5–E6 For 16-bit emulation (8086, 80186, or 80286 microprocessors).

E4–E5 For 8-bit emulation (8088 or 80188 microprocessors).

If you have a multiple-chassis system, check the jumpering for each buffer board supplied.

Installing Personality Modules and User Cables

There are three kinds of personality modules (also called probes): the 8086/8088 probe, the 80186/80188 probe, and the 80286 probe. Separate subsections are provided to explain installation of each probe type and each probe user cable.

Installing the I²ICE™ System 8086/8088 Emulation Personality Module

The 8086/8088 emulation personality module consists of the 8086/8088 personality board, the user cable, and the buffer box cover. The 8086/8088 emulation personality module connects to the buffer base assembly and configures the generic portion of the I²ICE system to emulate a specific processor.

The emulation personality board contains jumper selections that define the level of 8087 support. The coprocessor is internal when an iSBC 337 MULTIMODULE™ board is installed on the 8086/8088 emulation personality board. When the I²ICE system is running with an internal coprocessor, the LED on the buffer box labeled 8087 lights up. The coprocessor is external when an iSBC 337 MULTIMODULE board is installed in the user system.

In Tables A-1 and A-2, the jumpers appearing in bold type indicate the default jumper configuration for the 8086/8088 emulation personality board. Refer to Tables A-1 and A-2 and Figures A-7, A-8, and A-9 when installing the 8086/8088 emulation personality module. Many of the steps described here may have already been performed; you need only verify them.

1. Two ribbon cables are included with the 8086/8088 personality module. Use these cables to replace the two interconnect cables attached to the 8086/8088 probe buffer board (the probe buffer board is shipped with the emulation base module). The new ribbon cables minimize cross talk. Use Figure A-7 together with the following text to replace the cables.

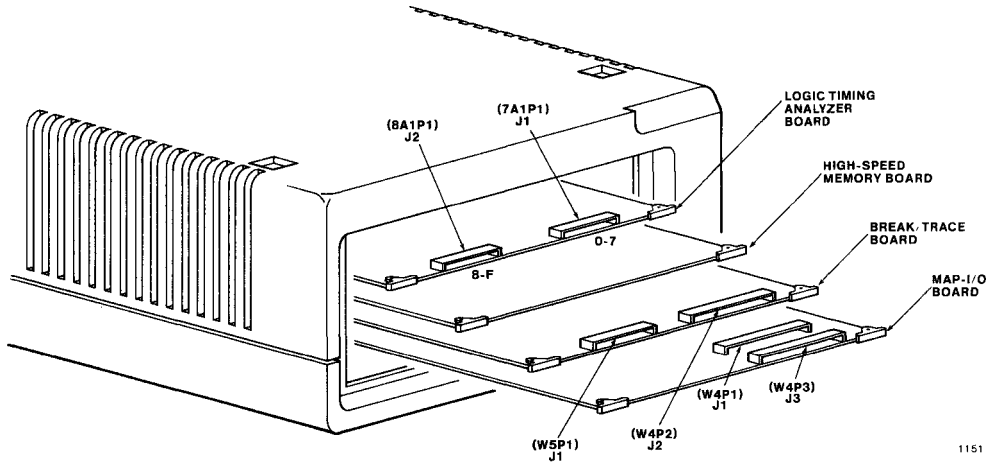


Figure A-4 Instrumentation Chassis Boards

E1-E2 }
E3-E4 } JUMPERED FOR NORMAL OPERATION.

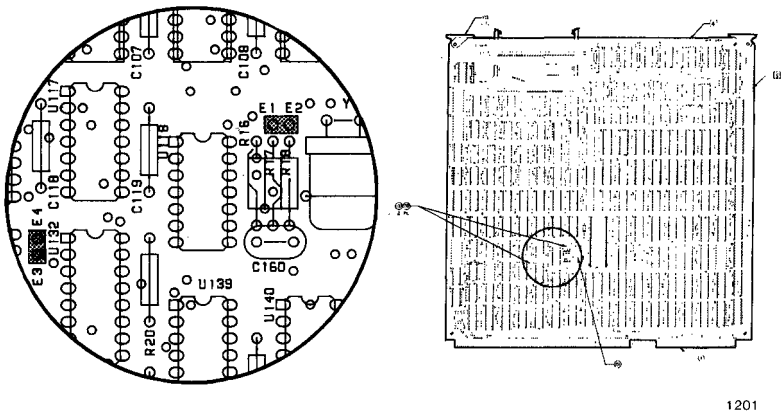


Figure A-5 Jumper Positions on the Map-I/O Board

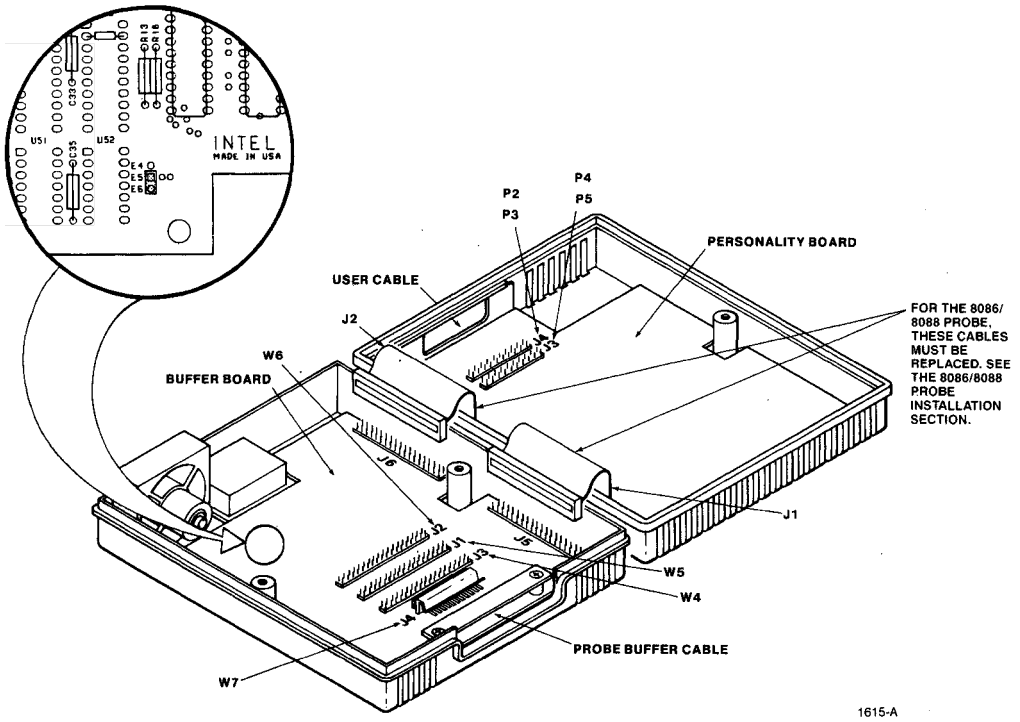


Figure A-6 Jumper Positions on the Buffer Board

Table A-1 8086/8088 Emulation Personality Module Jumper Configurations

Jumper	8086 MAX/MIN	8088 MIN	8088 MAX	Normal*	Piggyback Board 8086/88 MIN	Piggyback Board 8086/88 MAX
W3	**E8-E9**	E7-E8	E7-E8			
W4				**E12-E13**		
W5				**E14-E15**		
W6	**E16-E17**	E17-E18	E17-E18			
W8	**E22-E23**	E23-E24	E23-E24			
W9	**E26-E27**	E25-E26	E26-E27			
W11				**E10-E11**		
W12	**E55-E56**	E56-E57	E56-E57			
					E2-E1	E2-E3
					E5-E4	E5-E6

*Do not change these jumper configurations.

Table A-2 Jumpering for 8087 Support

Coprocessor	Location of User Plug	NMI Source	Jumpers		
			W1	W7	W10
internal coprocessor	user system	from user system	E1-E2	**E19-E20**	E28-E29
		from 8087 INTR	E1-E2	E20-E21	E28-E29
	loopback	none	E1-E2	**E19-E20**	E28-E29
		from 8087 INTR	E1-E2	E20-E21	E28-E29
no internal coprocessor	user system	from user system	**E2-E3**	**E19-E20**	**E29-E30**
	loopback	none	**E2-E3**	**E19-E20**	**E29-E30**

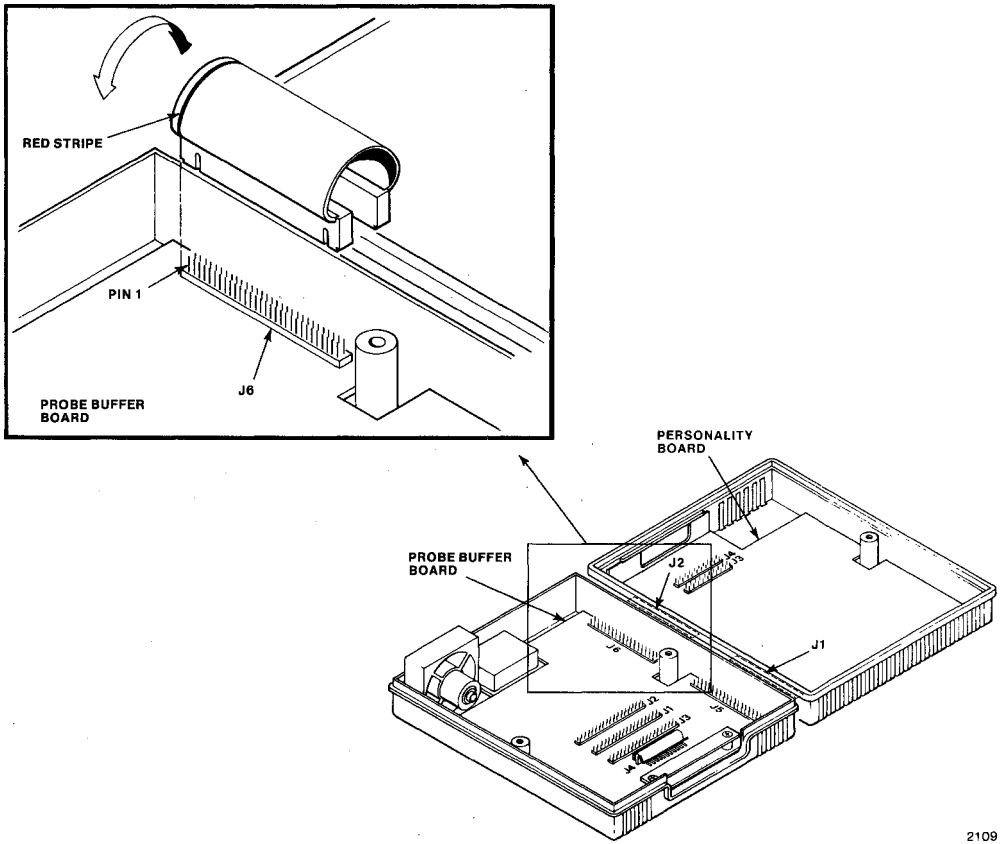


Figure A-7 New Cable Installation for the 8086/8088 Probe

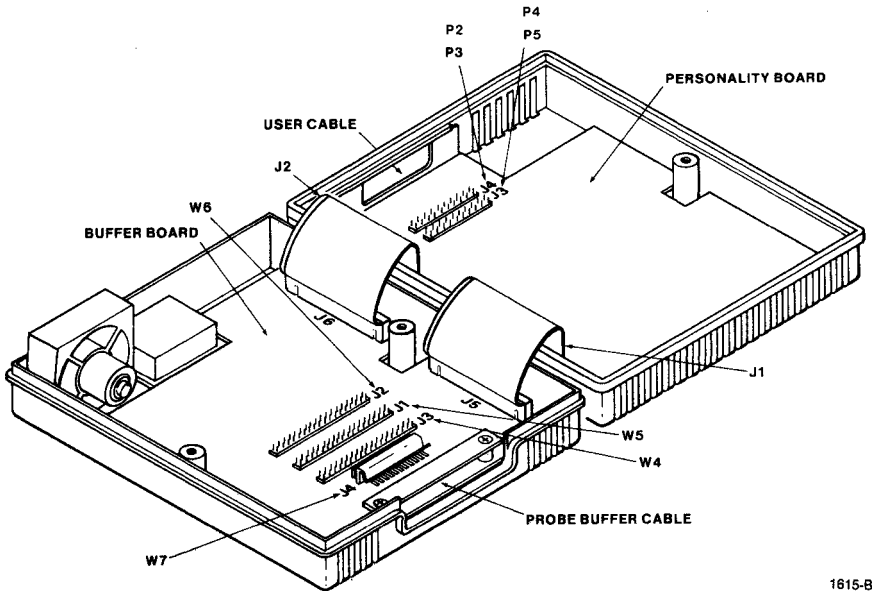


Figure A-8 8086/8088 Emulation Personality Module Installation

The two cables that you will replace connect the two halves of the probe buffer box. One half of the buffer box contains the probe buffer board (the words "PROBE BUFFER" are silk-screened on it); this half of the buffer box is shipped with the emulation base module. The other half of the buffer box contains the personality board (the words "86/8 PERSONALITY" are silk-screened on it); this half of the buffer box is shipped with the 8086/8088 personality module. Jack numbers specified in the following instructions are also silk-screened on the boards.

a) Remove the existing cables connected to J5 and J6 of the probe buffer board.

b) Locate the bag containing the translucent ribbon cables. Each cable is marked with the number 165199-001.

c) Insert one of the new translucent cables at J6 on the probe buffer board, aligning the red stripe on the cable with pin 1 of the J6 connector (pin numbers are also silk-screened on the boards).

IMPORTANT

Ensure that the side of the cable connector from which the cable emerges faces away from the buffer box edge (see Figure A-7).

d) Bend the other end of the cable so that you can insert it at J2 of the personality board.

e) Repeat steps 3 and 4 for the second cable, connecting the cable from J5 of the probe buffer to J1 of the personality board.

2. Ensure that plug W8P2 of the user cable is connected to J3 of the personality board. Ensure that plug W8P3 of the user cable is connected to J4 of the personality board (see Figure A-8).

3. Remove any unnecessary slack. Secure the user cable to the buffer box cover.

4. Refer to Tables A-1 and A-2 and Figure A-9 and install the indicated jumpers on the personality board.

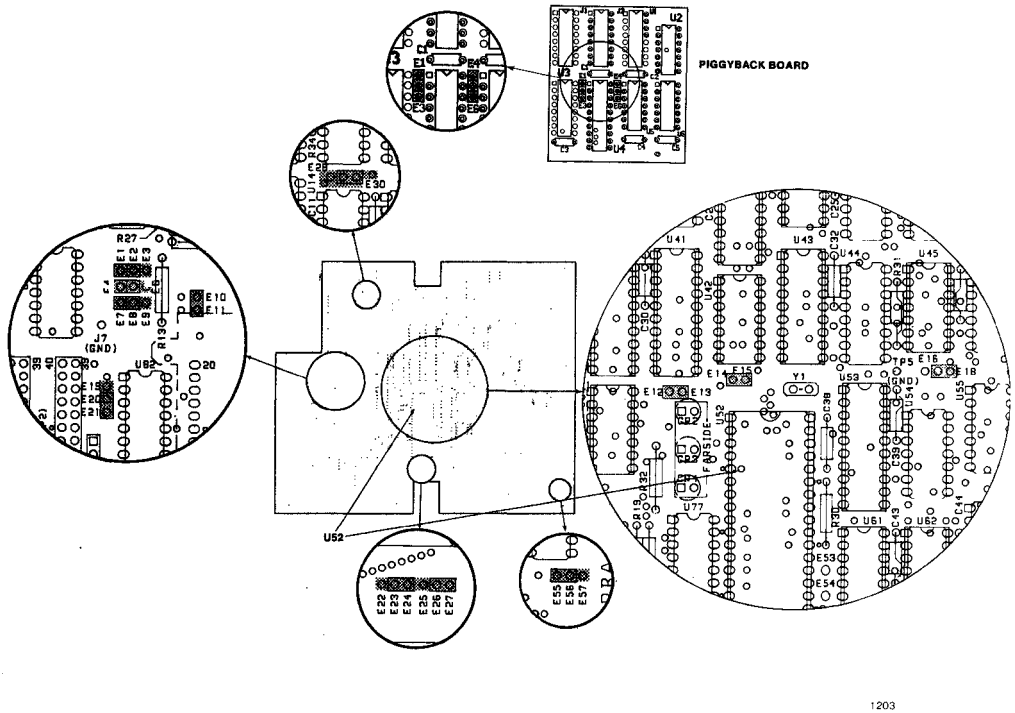


Figure A-9 Jumper Positions on the 8086/8088 Personality Board

5. If your emulator configuration calls for an 8087 processor used as an internal coprocessor, perform the following steps to install the iSBC 337 MULTIMODULE board in the U52 socket of the personality board. See Figure A-9.

   a. Remove the 8086 or 8088 microprocessor from the socket at U52.

   b. Stack three 40-pin sockets (packaged in an accessory kit) in the existing socket at U52.

   c. Stack three single-pin sockets in the existing socket at E53.

   d. Stack the remaining three single-pin sockets in the existing socket at E54.

   e. Install the microprocessor in the P1 socket of the iSBC 337 MULTIMODULE board.

   f. Install the iSBC 337 MULTIMODULE board in sockets U52, E53, and E54.

6. Install the supplied processor chip (8086 or 8088) for your emulator configuration in the U52 socket of the personality board (or P1 socket of the iSBC 337 MULTIMODULE board). This CPU chip is specially selected to work in I²ICE systems.

NOTE

The microprocessors supplied with the probes are special I²ICE components. To avoid timing problems, do not use standard production microprocessors with the I²ICE system.

7. Connect the two halves of the buffer box and secure with two screws, lockwashers, and washers.

The 8086/8088 personality board contains a 74F244 IC installed in a socket at location U30 that causes a 3.2 mA input current (IIL) on the CLK user pin. This load may be excessive in some user applications.

If loading on the CLK pin is critical for applications of 8 MHz and below, replace the 74F244 IC (at location U30 of the 8086/8088 personality board) with the 74S244 packaged with the 8088 microprocessor. This change reduces the input current (IIL) from 3.2 mA to 0.8 mA. In applications between 8 MHz and 10 MHz, the 74F244 IC must remain installed in U30.

Installing the 8086/8088 User Cable

When you are first learning about the I²ICE system, you will want to have the user cable looped back to the top of the buffer box for use with the I²ICE tutorial. Later, when you are ready to connect your probe to your prototype hardware (also called target hardware), return to this section for information on connecting the user cable to prototype hardware.

Figure A-10 shows the 8086/8088 user cable and gives its length and the height of the user plug.

To guard against electrical noise problems, the two ground pins (1 and 20) on the 8086/8088 probe are connected. The connection is at the end of the user cable in the microprocessor carrier. When connecting the user cable to prototype hardware or to the top of the buffer box, take care that pin 1 is in the correct position. Damage may result if the cable is connected

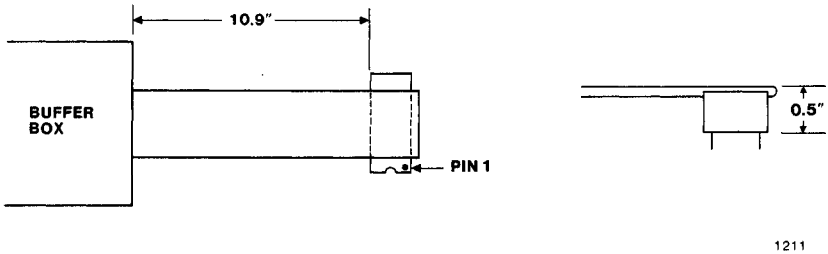


Figure A-10 8086/8088 User Cable Dimensions

incorrectly (see Figure A-11). On some versions a dot identifies the position on the buffer box socket that is intended to contain pin 1 of the user cable. Connect the cable to the top of the buffer box now; note the following caution before you make the connection.

CAUTION

Connecting the user cable to the top of the buffer box in the wrong orientation shorts +5V to ground on the personality board. Connecting the user cable to prototype hardware in the wrong orientation shorts +5V on the prototype to ground on the personality board. Refer to Figure A-11 for the correct orientation of the user cable.

When you want to connect your user cable to prototype hardware, you must determine whether sufficient room exists for placement of the user cable. If the prototype hardware resides in a card cage with a minimum inter-board separation of 0.56 inches (the MULTIBUS card cage), the slot above the prototype board must remain empty to allow access for the 8086/8088 user cable.

If you have multiple I²ICE chassis, install the emulation clips on the other chassis.

This completes installation of the 8086/8088 probe. If you have no other personality modules to install, go now to the emulation clips installation section that follows the section on installing the 80286 user cable.

Installing the I²ICE™ System 80186/80188 Emulation Personality Module

The 80186/80188 emulation personality module consists of the 80186/80188 personality board, the user cable, and the buffer box cover. The 80186/80188 emulation personality module connects to the buffer base assembly and configures the generic portion of the I²ICE system to emulate a specific processor. Refer to Figures A-12 and A-13 when installing the 80186/80188 personality module. Many of the steps described here may have already been performed; you need only verify them.

1. Connect the ribbon cable from J5 of the buffer board to J1 of the personality board (see Figure A-12).

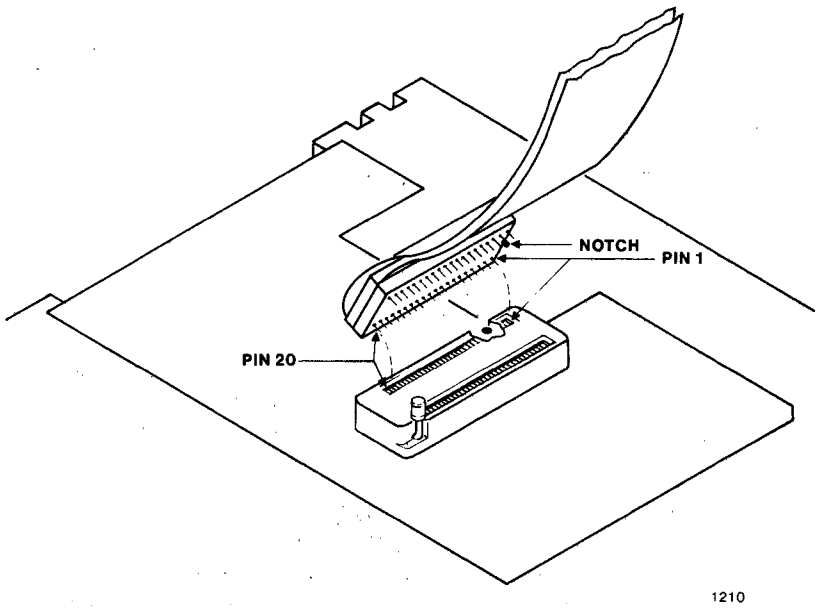


Figure A-11 The Correct Orientation of the 8086/8088 User Cable

2. Connect the ribbon cable from J6 of the buffer board to J2 of the personality board (see Figure A-12).

3. Install the indicated jumpers for the emulator configuration (see Figure A-13):

   E17-E18 for 16-bit emulation (80186)
   E16-E17 for 8-bit emulation (80188)

4. Verify that the proper microprocessor chip (80186 or 80188) for the emulator configuration is in the U19 socket of the personality board. Note that this is a bond-out chip available only from Intel. The U19 socket is on the personality board; the PGA socket on top of the buffer box is reserved for the user cable in loopback mode.

CAUTION

The microprocessors supplied with the probes are special I²ICE components. To avoid problems and potential damage, do not use standard production microprocessors with the I²ICE system.

5. Connect the two halves of the buffer box and secure with two screws, lockwashers, and washers.

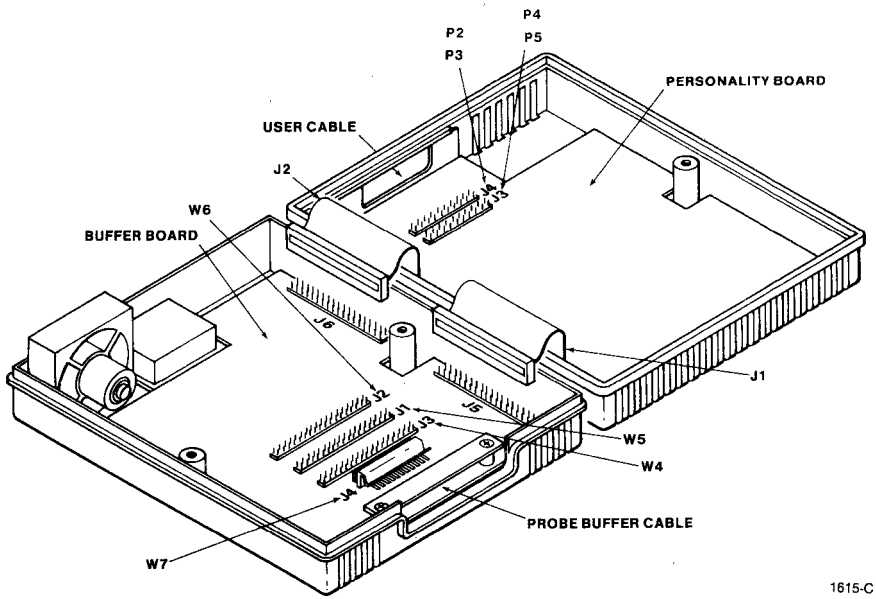


Figure A-12 80186/80188 and 80286 Emulation Personality Module Installation

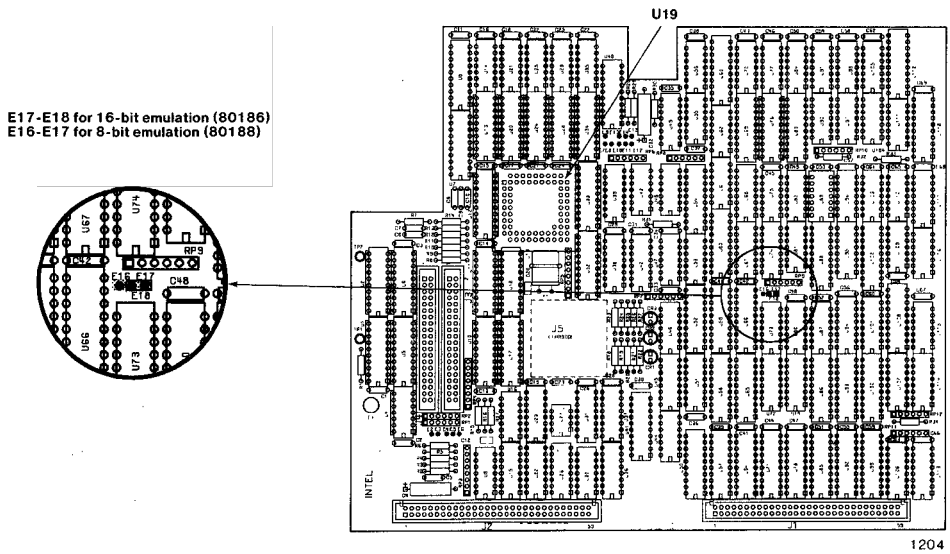


Figure A-13 Jumper Positions on the 80186/80188 Personality Board

Installing the 80186/80188 User Cable

When you are first learning about the I²ICE system, you will want to have the user cable looped back to the top of the buffer box for use with the I²ICE tutorial. Later, when you are ready to connect your probe to your prototype hardware (also called target hardware), return to this section for information on connecting the user cable to prototype hardware.

Figure A-14 shows the 80186/80188 user cable and its pertinent dimensions.

Prepare now to connect the user cable to the top of the buffer box. You must orient the user socket on the prototype board so that pin 1 is toward the edge connectors and on the right when facing the front of the card cage (see Figure A-14). Read the directions in the following caution note.

CAUTION

When you connect the user cable to the socket assembly, be careful not to damage the tab by pin 1. Install the user cable as follows (see Figure A-15):

1. Carefully place the user cable in place while observing that the pin 1 tab is not damaged by the end of the cable.
2. Slide the retaining clip in place.
3. Turn the retaining bail to secure the user cable in place.

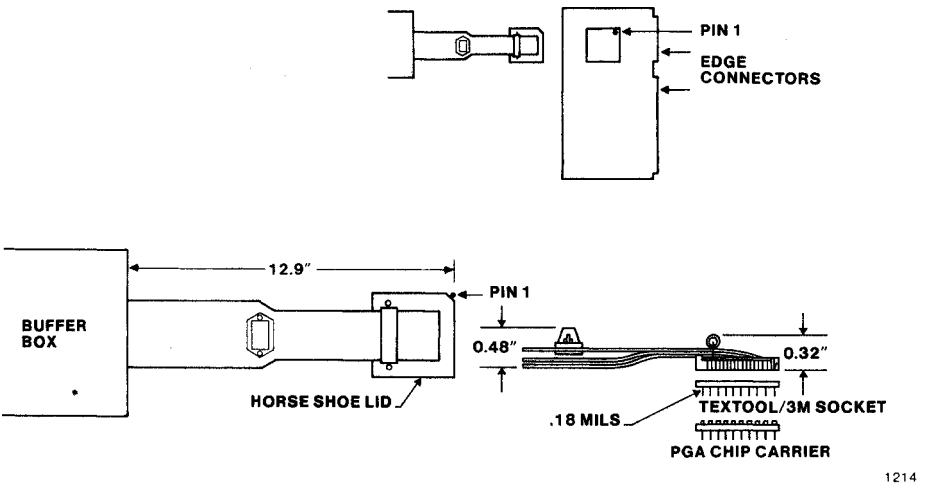


Figure A-14 80186/80188 User Cable Dimensions

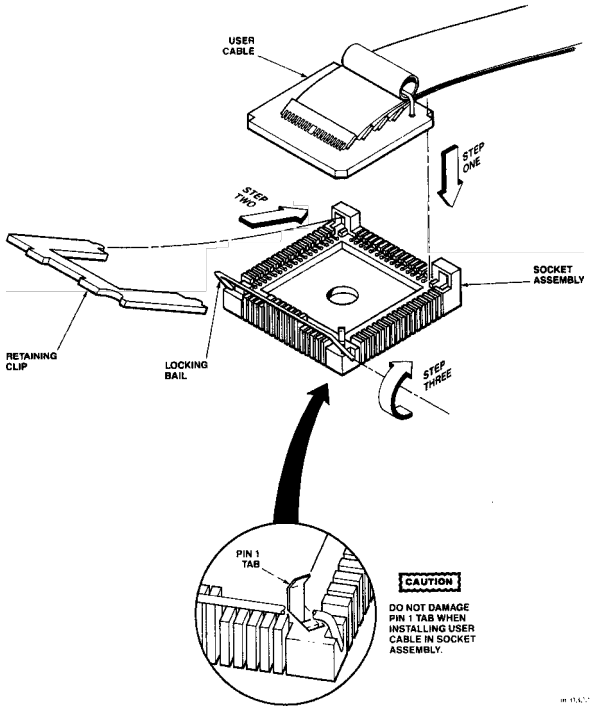


Figure A-15 Connecting the 80186/80188 User Cable

When you want to connect your user cable to prototype hardware, you must determine whether sufficient room exists for placement of the user cable. If the prototype hardware resides in a card cage with a minimum inter-board separation of 0.56 inches (the MULTIBUS card cage), the slot above the prototype board must remain empty to allow access for the 80186/80188 user cable.

NOTE

When you use the 80186/80188 or 80286 probe, Intel recommends that the prototype contain the Textool/3M socket 268-5400.

If you have multiple I²ICE chassis, install the emulation clips on the other chassis.

This completes installation of the 80186/80188 probe. If you have no other personality modules to install, go now to the emulation clips installation section that follows the section on installing the 80286 user cable.

Installing the I²ICE™ System 80286 Emulation Personality Module

The 80286 emulation personality module consists of the two 80286 personality boards, the user and ground clip cables, and the 80286 buffer box cover. The 80286 emulation personality module connects to the buffer base assembly and configures the generic portion of the I²ICE system to emulate a specific processor. Refer to Figure A-12 when installing the 80286 emulation personality module. Many of the steps described here may have already been performed; you need only verify them.

1. Connect the ribbon cable from J5 of the buffer board to J1 of the personality board (see Figure A-12).
2. Connect the ribbon cable from J6 of the buffer board to J2 of the personality board (see Figure A-12).
3. Ensure that plugs P2 and P3 of the user cable are connected to J4 of the personality board. Ensure that plugs P4 and P5 of the user cable are connected to J3 of the personality board (see Figure A-12).
4. Connect the two halves of the buffer box and secure with two screws, lockwashers, and washers.
5. Install the printed circuit board (PCB) located at the end of the user cable into the loopback socket assembly that protrudes from the module assembly plastic. Do not remove the socket loopback assembly from the module assembly.

CAUTION

Do not install anything except the user cable in the loopback socket.

Installing the 80286 User Cable

When you are first learning about the I²ICE system, you will want to have the user cable looped back to the top of the buffer box for use with the I²ICE tutorial. Later, when you are ready to connect your probe to your prototype hardware (also called target hardware), return to this section for information on connecting the user cable to prototype hardware.

Figure A-16 shows the 80286 user cable and its pertinent dimensions.

Connect the cable to the top of the buffer box now.

When you want to connect your user cable to your prototype hardware, consider whether enough room is available for the user cable. If the prototype hardware resides in a card cage with a minimum inter-board separation of 0.56 inches (the MULTIBUS card cage), the slot above the prototype board may need to remain empty to allow access for the 80286 user cable.

The slot above the prototype need not be empty if the user socket on the prototype has pin 1 toward the edge connectors and on the right when facing the front of the card cage (the A

- A ORIENTATION DOES NOT REQUIRE EXTRA SLOT
- B ORIENTATION REQUIRES EXTRA SLOT

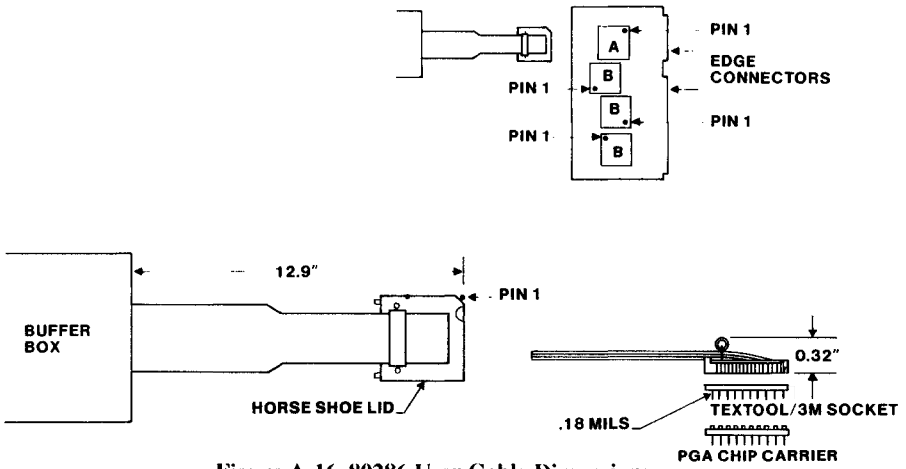


Figure A-16 80286 User Cable Dimensions

1215

orientation). The slot above the prototype must be empty if pin 1 of the user socket is in any other position (the B orientations) because the B orientations require additional room for a bend in the cable (see Figure A-16).

The A and B orientation cautions do not take into account the pin-grid-array (PGA) socket plugged into the leadless-chip-carrier (LCC) socket at the end of the user cable. This socket adds 0.116 inches. If you retain the PGA socket at the end of the cable, even the A orientation may require an empty slot.

NOTE

When you use the 80186/80188 or 80286 probe, Intel recommends that the prototype contain the Textool/3M socket 268-5400.

This completes installation of the 80286 probe. If you have no other personality modules to install, go now to the emulation clips installation section.

Installing the Emulation Clips Module

After your personality modules are installed, perform the following steps to install the emulation clips module.

1. Connect the terminator assembly to the emulation clips module (P2 to J2) (see Figure A-17).
2. Install as many microhooks as needed for the system on the wires of the terminator assembly (see Figure A-17).
3. Feed the W5P1 end of the emulation clips cable through the external strain relief, then through the left-hand slot at the lower front of the instrumentation chassis, through the internal strain relief, and connect W5P1 to jack J1 on the break/trace board (see Figures A-4 and A-18).

If you have multiple I²ICE chassis, install the emulation clips on the other chassis.

This completes installation of the emulation clips module. If you have the iLTA logical timing analyzer option, go to the next section. If you do not have this option, remove any unnecessary slack from the probe cables and any other chassis cables. Secure the cables to the bottom of the instrumentation chassis with the supplied cable clamps; then skip the next section and go to the Host Installation section.

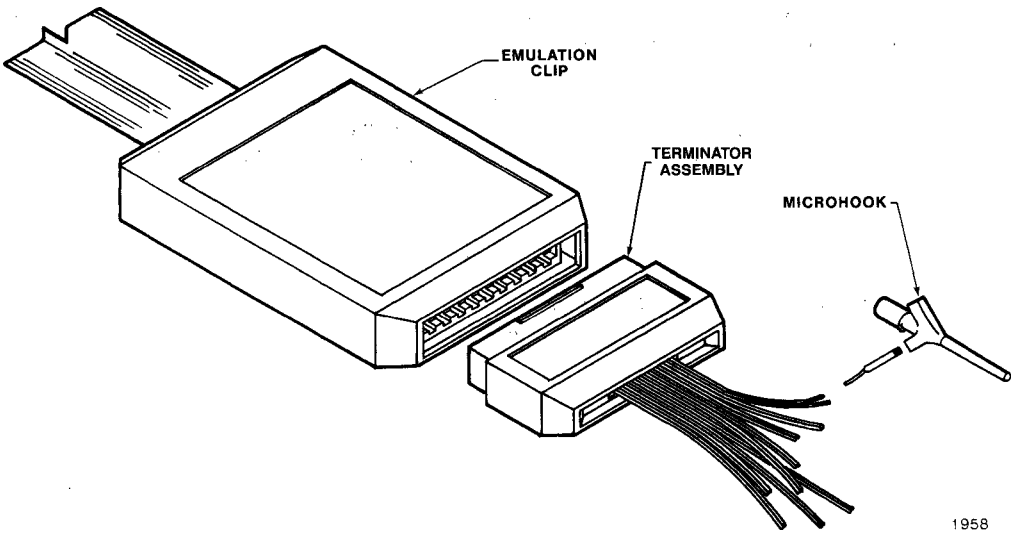


Figure A-17 Assembling the Emulation Clips Module

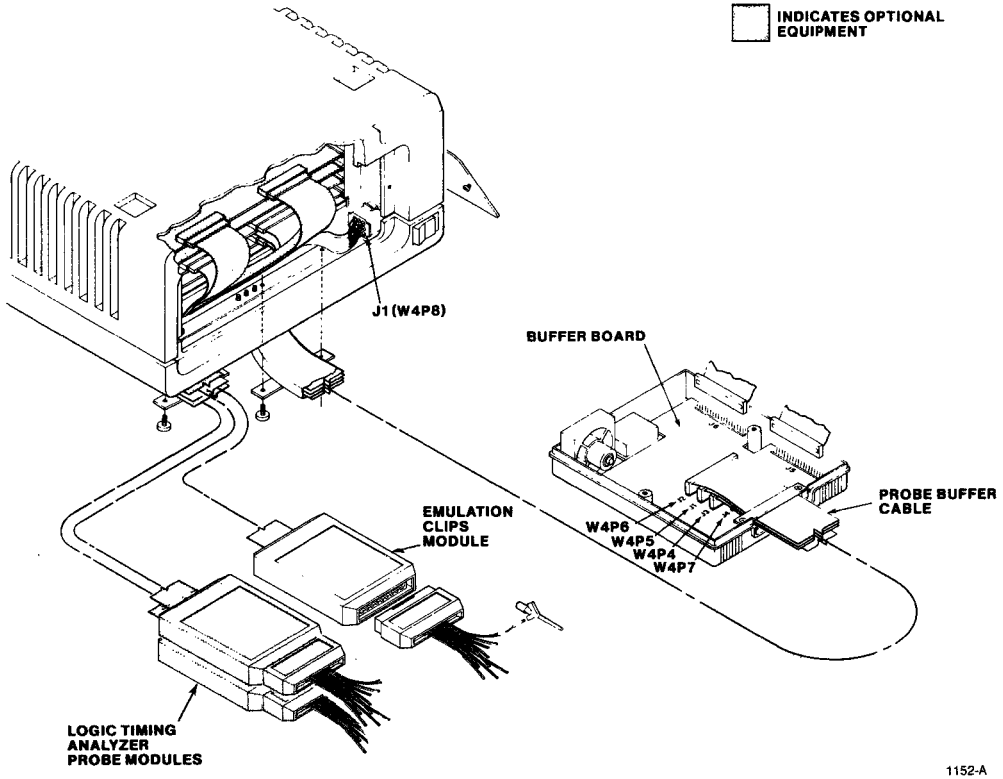


Figure A-18 Instrumentation Chassis Cables

Installing the iLTA Logic Timing Analyzer Option

Each instrumentation chassis can have only one iLTA module. [The iLTA option is not available for Model 800 or IBM PC hosts.] Refer to Figures A-4 and A-18 when you install the iLTA module.

1. Install the iLTA board in the top slot of the PICE instrumentation chassis.

CAUTION

The iLTA board must be installed in the top slot to ensure proper cooling.

2. Route the two iLTA probe cables through the external strain relief on the bottom of the instrumentation chassis and then through the cable slots at the front of the instrumentation chassis base. Route both iLTA probe cables through the left-hand cable slot (together with the emulation clips module cable—see Figure A-18).
3. Connect the 7A1P1 connector of the channel 0-7 probe cable to jack J1 of the iLTA board (see Figures A-4 and A-18).
4. Connect the 8A1P1 connector of the channel 8-F probe cable to jack J2 of the iLTA board (see Figures A-4 and A-18).
5. Remove any unnecessary slack from the probe cables and any other chassis cables. Secure the cables to the bottom of the instrumentation chassis with the supplied cable clamps.
6. Connect a terminator assembly on the end of each of the iLTA probes.
7. Install the microhooks needed for the system on the wires of the terminator assembly.

NOTE

With an iLTA unit installed, there may be some degradation of ESD and AC line noise immunity.

Host Installation Information

You have now completed the installation of host-independent portions of the overall PICE system installation process. Next you must install host-dependent portions.

If your host system has a non-standard terminal, go now to Appendix B for information on configuring your terminal.

If you have an IBM PC host, refer now to Appendix C. If you have an Intel host you did not receive the Appendix C for IBM PC hosts. Instead, you received Appendixes C through G that concern Intel hosts. Table A-3 shows to which appendix (or appendixes) owners of Intel hosts should now refer.

Table A-3 Intel Host Installation Appendixes

Your Host	Hardware Installation Appendix	Software Installation Appendix
Model 800	C	F
Stand-alone Series III	D	E
Series III on a Network	D	F
Series IV	G	G

B CONFIGURING THE I²ICE™ SYSTEM FOR NON-STANDARD HOST TERMINALS

The I²ICE system is designed to run on either an Intel host development system with a standard Intel CRT or on an IBM PC/AT or PC/XT with a standard terminal. The codes expected from the terminal or sent to the terminal are those used by standard Intel or IBM terminals.

You must configure a non-standard terminal for use with the I²ICE system. Do this by creating a CRT file that contains configuration commands that change the terminal codes to those expected by I²ICE software.

If your terminal is one commonly used with Intel equipment, check *The AEDIT User's Guide* (order number 121756) for the listings of some commonly used CRT configuration files.

Include the CRT configuration file when you invoke I²ICE software. If you name the file I2ICE.CRT, I²ICE software automatically configures the terminal to its specifications. Otherwise, you must specify the file in the invocation line. For example, the following command for a stand-alone Series III host invokes I²ICE software and configures the terminal according to the specifications in the CRT file, 1510T.CRT.

```
-RUN :F1:I2ICE CRT(:F2:1510T.CRT)
```

Creating a CRT File

Check the user's manual that comes with the terminal for the codes expected and generated by the terminal. To create a CRT file, compare the terminal's behavior with the following I²ICE software expectations:

- ASCII codes 20H through 7EH display some symbol requiring a one-column space. The carriage return (0DH), linefeed (0AH), and backspace (08H) perform their usual functions.
- There are cursor key output codes and CRT cursor move input codes for the cursor functions down, home, left, right, and up. There are also cursor key input codes for clear screen, clear rest of screen, and clear line.
- The home position is the upper left corner.
- The terminal accepts a blank-out code that blanks out the contents of the screen.
- The CRT has 22 to 25 lines.
- The screen scrolls. When the cursor is on the bottom line of the screen and you press RETURN (or Enter) or the line wraps around from the right margin, the top screen line is deleted and the screen rolls up one line.

- The I²ICE software automatically generates a linefeed each time the carriage return is entered. The terminal should not generate a linefeed with a carriage return. In some terminals, this function can be switched on and off.

Configuration Commands

Configuration commands modify the environment and the communication link between I²ICE software and the keyboard and the screen. There are two types of configuration commands. The A command modifies the way data is presented on the screen. The AF command modifies the codes to and from the terminal.

The format of the A command is as follows:

A*code* = *value*

Where:

A	is the command name.
code	represents a single character specifying which parameter is to be changed.
value	is a hexadecimal number except for the number of lines shown on the display.

The following examples use the A command.

AV = 22	changes the number of lines on the screen to 22.
AB = 7	changes the keyboard BREAK character to CTRL-G (which has a hexadecimal value of 07).

The format of the AF command is as follows:

AF*code* = *value*

Where:

AF	is the command name.
code	is a two-character string specifying the function to be changed.
value	depends on which code is specified. Every character entered after the equal sign is interpreted as part of the value, including spaces.

The following examples use the AF command.

AFMD = 0A	specifies that the code required to move the cursor down the screen is a single linefeed (0AH).
AFER =	specifies that the terminal being used does not have an erase-rest-of-screen code.

Table B-1 lists the valid codes and their values for the A configuration. Table B-2 lists the AF configuration command values.

Table B-1 The A Configuration Command Values

Acode	Series III & IBM PC* Defaults	Series IV Defaults	Description
AB	1B	1B	ESCAPE — a new value should be assigned for terminals that require the ESCAPE key for control sequences.
AO	0	20	The offset value to be added to the row and column numbers following the cursor control sequence of the AF command AFAC. The value must be entered as a single hexadecimal byte. The setting has no effect until cursor addressing mode is set with AFAC.
AR	7F*	7F	RUBOUT — deletes the character to the left of the cursor.
AV	25	25	The number of lines displayed on each screen. The possible values are 22, 23, 24, or 25. Screens smaller than 25 lines have a smaller command area; the text area remains the same at 20 lines. It is entered as a decimal value.
AW	T*	F	T (true) indicates that the terminal generates a carriage return and linefeed if a character is printed in column 80 (a wrapping terminal). F (false) indicates that the terminal is not a wrapping terminal.
AX	T	F	Cursor addressing format. The cursor address is an ordered pair of (x,y) coordinates. The format can be (column,row) or (row,column). T sets the format as (column,row). F sets the format as (row,column).

*The defaults for the IBM PC/AT and PC/XT are the same as those for the Series III, with two exceptions. The IBM PC default values for AR and AW are as follows:

AR = 08
AW = F

Table B-2 The AF Configuration Command Values

AFcode	Series III & IBM PC* Defaults	Series IV Defaults	Description
			The setting has no effect until cursor addressing mode is set with the AF command, AFAC.
AFAC	00*	1B19	Code used as the cursor movement command by the terminal. When the command is given, the coordinates of the new cursor address follow the code. The coordinates are given in the order specified by the AX command and with the offset specified by the AO command.
AFBK	20	20	Code that blanks out a single screen location.
AFCD	1C	88	Cursor down code.
AFCH	1D	81	Cursor home code.
AFCL	1F	89	Cursor left code.
AFCR	14	8A	Cursor right code.
AFCU	1E	87	Cursor up code.
AFDL	(null)*	(null)	Delete line code. Used to speed up the display on the Hazeltine 1510 and similar terminals.
AFEK	1B4B	(null)	Code to erase the entire line.
AFEL	(null)	1B11	Code to erase the rest of the line following the cursor.
AFER	1B4A	1B10	Code to erase the rest of the screen following the cursor.
AFES	1B45	1B05	Code to erase the whole screen.
AFIG	(null)	(null)	A byte to be ignored whenever it is received from the keyboard as input. If AFIG is set to 00, all bytes are accepted from the keyboard. This character is needed on terminals that have multiple character key codes for UP and DOWN, such as the Hazeltine 1510. AFIG should be set to the lead-in (tilde), and UP and DOWN should be set to the second letter of the cursor up or down key code. This avoids problems caused by lack of a typehead buffer.

*The defaults for the IBM PC/AT and PC/XT are the same as those for the Series III, with three exceptions. THe IBM PC default values for AFAC, AFDL, and AFIL are as follows:

AFAC = 1B47
AFDL = 1B49
AFIL = 1B4C

Table B-2 The AF Configuration Command Values (continued)

AF*code*	Series III & IBM PC* Defaults	Series IV Defaults	Description
AFIL	(null)*	(null)	Insert line code. Used for reverse scrolling.
AFMB	0D	0D	Code to move the cursor to the beginning of the line.
AFMD	1C	1B02	Code to move the cursor down.
AFMH	1D	1B08	Code to move the cursor to the home position.
AFML	1F	1B04	Code to move the cursor to the left.
AFMR	14	1B03	Code to move the cursor to the right.
AFMU	1E	1B01	Code to move the cursor up.
AFTM	16	16	CTRL-V — this command is unique to I²ICE systems. It sets the control character that enables you to turn the menu display on and off.
AFXA	1	1	CTRL-A — delete right.
AFXF	6	80	CTRL-F — character delete.
AFXU	15	15	CTRL-U — undo command.
AFXX	18	18	CTRL-X — delete left.
AFXZ	1A	82	CTRL-Z — clear line.

*The defaults for the IBM PC/AT and PC/XT are the same as those for the Series III, with three exceptions. THe IBM PC default values for AFAC, AFDL, and AFIL are as follows:

AFAC = 1B47
AFDL = 1B49
AFIL = 1B4C

GLOSSARY

Address	An address is an unsigned value that corresponds to a location in program memory. The I²ICE system recognizes absolute addresses, virtual addresses, and symbolic references to addresses.
Arm	The arm condition is an optional part of a break/trace sequence in the I²ICE system. A set of arm conditions can be used to ensure that a system break is not possible until all required qualifying conditions are satisfied.
Binary Operator	A binary operator acts on two operands to create a single operand. In addition to the normal Boolean, relational, and arithmetic binary operators, the I²ICE system also recognizes a pointer operator (:) that creates an address pointer using two 16-bit values.
Breakpoint	Breakpoints are specific points in a sequence of events or in the flow of a program that are used to stop emulation.
Emulation	The I²ICE system is in emulation when a user program is running.
Event	An event in the I²ICE system is any condition that can be described with I²ICE command syntax. Typical events are the execution of an instruction or the accessing of a memory location. The I²ICE command language uses events to describe conditions that specify such functions as breaking emulation and enabling trace collection.
Event Machines	The event machines implement single or multiple event recognition and cause a trigger when all defined event conditions are recognized.
Execution	There are two types of execution in the I²ICE system: I²ICE command execution and user program execution. I²ICE command execution is complete when the prompt (* or ?) is returned at the system console. User program execution is under control of the I²ICE system and is complete when the user program encounters a break or the HALT command is issued.
Expression	A series of operands and operators that yields a numeric, Boolean, or string value.

Forcing Character	The forcing character in the I²ICE command syntax is the double-quote (") unary operator. When the forcing character precedes a keyword, the I²ICE system interprets the keyword as a user program symbol.
History Buffer	A buffer that stores recent commands. The commands can be recalled using the up-arrow key.
Interrogation Mode	The I²ICE system is in interrogation mode whenever it is not in emulation mode, that is, it is in interrogation mode whenever the asterisk prompt (*) appears. In emulation mode, the prompt is a question mark (?).
Keywords	Keywords have reserved definitions within the I²ICE command language. See the Keywords entry in the *I²ICE™ System Reference Manual*.
Map	The I²ICE system uses a memory map to direct processor address space to physical memory locations and to control access to mapped program memory during emulation.
Partition	A partition in I²ICE command syntax is a range of addresses.
Probe	Probe refers to the emulation personality module of the I²ICE system. The probe consists of the hardware and software required to make the I²ICE system emulate a particular processor in the prototype system.
Probe Microprocessor	The probe microprocessor is the CPU chip installed in the emulation personality module of the I²ICE system. For example, using the supplied 8088 CPU as the probe processor, the emulation personality module is tailored to emulate 8088 processors.
Pseudo-variable	A pseudo-variable is an I²ICE-system-defined variable that cannot be removed by the user.
Real time	The term real time in I²ICE system emulation means that the prototype processor is operating at the design speed and no extra wait-states are added for mapped memory accesses.
Strings	In the I²ICE command language, a string is one or more characters enclosed in apostrophes (single quotes). Strings are stored as 8-bit ASCII values.
Symbolic References	In the I²ICE command language, symbolic references are user-defined strings that correspond to program addresses or to variables.
Syntax	The I²ICE command syntax is a formal set of rules that defines the requirements for command entry.

Syntax Menu	A menu at the bottom of the screen that indicates what command elements are legal during command entry.
Tracing	The I²ICE system keeps a record of trace information each time it enters emulation.
Unary Operator	A unary operator is an operator that acts on a single operand. The I²ICE system recognizes the NOT, +, −, ", and . unary operators.
Unit	Each configured chassis in the I²ICE system is a unit of that system.

WE'D LIKE YOUR OPINION

Please use this form to help us evaluate the effectiveness of this manual and improve the quality of future documents

To order publications, contact the Intel Literature Department (see page ii of this manual).

Fill in the squares below with a rating of 1 through 10:

POOR				AVERAGE					EXCELLENT
1	2	3	4	5	6	7	8	9	10

- ☐ Readability
- ☐ Technical depth
- ☐ Technical accuracy
- ☐ Usefulness of material for your needs
- ☐ Comprehensibility of material
- ☐ OVERALL QUALITY OF THIS MANUAL

If you gave a 4 or less (in any category), please explain here:

_____

_____

_____

What suggestions would you have for improving this manual:

_____

_____

★ ★ ★ ATTENTION ★ ★ ★

Receive 50% off on the next Intel publication you buy. Send us your comments, and we'll send you a 50%-off certificate.

If you would like us to call you for more specifics about this book, provide the following information. Please print clearly.

Name _____

Phone Number (_____)_____

Address _____

_____

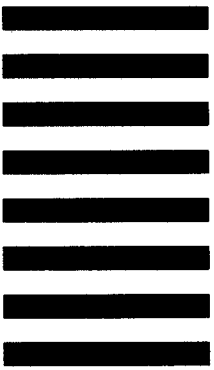
_____

Thanks for taking the time to fill out this form.

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR 95051

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 N.E. Elam Young Pkwy.
Hillsboro, OR 97124-6497

DSHO Technical Publications

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

Printed in U.S.A.

Instrumentation

0843/3K/1185/OSPS/AD