intel®

THE I²ICE™ INTEGRATED INSTRUMENTATION AND IN-CIRCUIT EMULATION SYSTEM REFERENCE MANUAL

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

Order Number: 166302-001

THE I²ICE™ INTEGRATED INSTRUMENTATION AND IN-CIRCUIT EMULATION SYSTEM REFERENCE MANUAL

Order Number: 166302-001

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	im	iPSC	OpenNET
COMMputer	iMDDX	iRMX	Plug-A-Bubble
CREDIT	iMMX	iSBC	PROMPT
Data Pipeline	Insite	iSBX	Promware
GENIUS	Intel	iSDM	QueX
Δ	intel	iSXM	QUEST
i	intelBOS	Library Manager	Ripplemode
i	Intelevision	MCS	RMX/80
I^2ICE	inteligent Identifier	Megachassis	RUPI
ICE	inteligent Programming	MICROMAINFRAME	Seamless
iCEL	Intellec	MULTIBUS	SLD
iCS	Intellink	MULTICHANNEL	UPI
iDBP	iOSP	MULTIMODULE	VLSiCEL
iDIS			

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

TABLES

FIGURES

FIGURES (continued)

The *I²ICE™ System Reference Manual* is the master reference manual in the I²ICE™ publications library. Refer to this manual for detailed operating information on I²ICE commands, topics, and error messages. The *I²ICE™ System Reference Manual* is divided into two chapters:

Chapter 1 is the command encyclopedia for the I²ICE command language. Each command and topic in the I²ICE command language is presented in alphabetical order. Each command entry contains the command syntax, a detailed description of the command, one or more verified examples, and cross-references to related commands and topics.

Chapter 2 describes the I²ICE error codes.

Notational Conventions

Chapter 1 is a detailed encyclopedia of the I²ICE system commands and topics in alphabetical order. Each command entry follows the same format. The following sections briefly describe a sample command entry.

Encyclopedia Commands and Topics

The two types of encyclopedia entries are topics and commands. The name of the command or topic discussed in each section is printed in red on the outside corner of each page in that section. Commands are printed entirely in uppercase letters (e.g., ACTIVE), while only the first letter of each topic is capitalized (e.g., System specification).

Topic Entries

A topic entry expands a subject or consolidates common command syntax for easy reference. A topic entry does not follow a pattern.

Command Entries

Most encyclopedia entries are I²ICE command keywords. The encyclopedia contains the commands that work with all I²ICE probe types, as well as commands that are probe-specific.

The following example describes the information found in a typical command entry.

COMMAND NAME

Purpose statement

Syntax

The command syntax shows how to construct a legal I²ICE command. (Syntax notation is explained in the following section.)

Where:

This section briefly explains each part of the command, including command options and initial and legal values.

Default

This section indicates the default value (if any) for the command.

Discussion

The discussion section details how commands are used. It augments the general information found in the *I²ICE™ System User's Guide* and contains information about why and when commands are most useful.

Examples

Each example uses the command in context. Examples begin with an explanation of how the command is used, what it is used for, and any assumptions the example makes. User input is shown in a shaded field, and system output is printed in a special typeface. For example:

```
*COMMAND
system response
```

Cross-References

Cross-reference items are commands and topics related to the encyclopedia entry.

Syntax Notation

The following syntax notation is used throughout this manual:

COMMANDS	Command keywords appear in all uppercase letters. (You may enter commands in either uppercase or lowercase letters.)
elements	Items for which you must substitute a value, expression, file name, etc., are shown 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 one or more 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 keyboard has two apostrophes (or single quotes), determine which one the I²ICE system accepts in command syntax. Do this by entering one of them. If the apostrophe you chose is not accepted by the I²ICE system, the message line will display "syntax error".
CTRL	CTRL denotes the terminal's control key. For example, CTRL-C means enter C while pressing the control key. (Note: Some keyboards use CNTL rather than CTRL to indicate the control key.)

NOTE

Entering CTRL-D invokes an internal debugger, used for debugging 8086 software, that runs on the host development system. Do not use this debugger when the I²ICE software is running. If you do enter CTRL-D, enter a G followed by a carriage return to return to the I²ICE software.

CTRL-D does not terminate the I²ICE command line. The G returns you to the same line at the point where you left. For example, assume you enter CTRL-D after entering EX. After returning to the I²ICE software, you can complete the EXIT command by entering IT, as shown in the following example.

```
*EX
013A:15A8       RET         :SHORT
PROCESSING ABORTED
*G
IT
 I2ICE terminated
```

Related Publications

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

I²ICE™ Integrated Instrumentation and In-Circuit Emulation System (data sheet), order number 210469

I²ICE™ System User's Guide, order number 166298

PSCOPE-86 High-Level Program Debugger User's Guide, order number 121790

AEDIT Text Editor User's Guide, order number 121756

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

SERVICE AND REPAIR ASSISTANCE

The best possible service for your Intel product is provided by an Intel Customer Engineer. 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.

The Intel Customer Engineer 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 the 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 informa-

tion 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

Encyclopedia

This chapter contains the I²ICE™ system commands and topics in alphabetical order. Table 1-1 groups the commands by function.

Table 1-1 I²ICE™ System Commands Grouped by Function

Function	Command	Description
Address	GRANULAITY	Determines the block size used for 80286 probe memory mapping.
	SEL286	Determines whether the 80286 probe performs 8086 address translation or 80286 address translation.
	TSS	Displays the current task state segment for the 80286 probe when in protected mode.
Arm	ARMREG	Defines or modifies a debug register that contains arm, trigger, and disarm or delay sequences.
	SYSTEM	Sets the initial state of the system arming functions.
Block Commands	COUNT	Groups and executes commands a specified maximum number of times.
	DO	Groups and executes commands.
	IF	Groups and conditionally executes commands.
	REPEAT	Groups and executes commands forever or until an exit condition is met.
Break	BRKREG	Defines a register that contains break specifications.
	ENABLE	Conditions the unit to accept system-level breaks and traces.
Coprocessor	COENAB	Enables or disables coprocessor functions.
	COREQ	Enables or disables external numeric extension activity for the 80286 probe.
	CPMODE	Displays or changes the external coprocessor mode.
	GET87	Defines register handling conditions for the 8087 coprocessor.
	PHANG	Enables and disables system timeout (for the 8086/8088 and 80186/80188 probes) based on coprocessor activity.
Counter	TIMEBASE	Sets the counter source and the increment, and formats the trace buffer timetag.

Table 1-1 I²ICE™ System Commands Grouped by Function (continued)

Function	Command	Description
Debug Procedures	ARMREG	Defines or modifies a debug procedure that contains arm, trigger, and disarm or debug sequences.
	BRKREG	Defines a procedure that contains break specifications.
	EVTREG	Defines a procedure that controls the event machine.
	PROC	Defines, displays, or executes a debug procedure.
	REMOVE	Deletes all user program symbols or specified debug object definitions.
	SYSREG	Defines a procedure that contains system break specifications.
	TRCREG	Defines a procedure that contains user program tracing specifications.
Directory	DIR	Displays program symbols and debug objects.
Editor	EDIT	Invokes the I²ICE system editor.
Emulation	CAUSE	Displays the reason emulation stopped.
	EXIT	Ends emulation.
	GO	Starts emulation and controls break and trace functions.
	HALT	Breaks emulation from the terminal.
	I2ICE	Invokes the I²ICE software.
	LOAD	Copies a program from a file into mapped program memory.
	RESET	Reinitializes specified functions of the I²ICE system.
	WAIT	Suspends command execution during emulation.
Error Messages	ERROR	Controls the amount of error information displayed.
Event Machines	EVTREG	Defines a register that controls the event machine.
	SCTR	Assigns a value to the system event machine counter.
	XCTR	Assigns a value to the execution event machine counter.
Event Register	EVTREG	Defines a register that controls the event machine.
Execution Point	$	Displays or changes the current execution point.
	NAMESCOPE	Displays or sets the current NAMESCOPE for symbolic references.
Expressions	EVAL	Calculates and displays the result of an expression.

Table 1-1 I²ICE™ System Commands Grouped by Function (continued)

Function	Command	Description
Files	APPEND	Saves definitions of debug objects to a file.
	INCLUDE	Retrieves command definitions from a system
	LIST	Opens or closes a log file.
	PUT	Creates and saves system file contents from memory to a file.
	SAVE	Saves the current memory image to a file.
Functions	F2XM1	2^x-1 function.
	FLDL2E	Constant $\log_2(e)$.
	FLDL2T	Constant $\log_2(10)$.
	FLDLG2	Constant $\log_{10}(2)$.
	FLDLN2	Constant $\log_e(2)$.
	FLDPI	Pi.
	FPATAN	Partial arctangent.
	FPTAN	Partial tangent.
	FSQRT	Square root.
	FYL2X	$Y * \log_2(x)$.
	FYL2XP1	$Y * \log_2(x + 1)$.
Help	CAUSE	Displays the reason emulation stopped.
	HELP	Provides on-line operating assistance.
	MENU	Enables and disables the I²ICE syntax menu.
I/O Ports	HOLDIO	Suspends I/O requests to ICE-mapped ports.
	PORT	Displays or changes the contents of byte-wide I/O ports.
	RELEASEIO	Resumes emulation after the HOLDIO command.
	WPORT	Displays or changes the contents of word-wide I/O ports.
Logic Clips	CLIPSIN	Displays the current state of the emulation logic clips.
	CLIPSOUT	Sets the two output lines on the emulation logic clips.
Memory Types	ADDRESS	Displays or changes memory as 16-bit unsigned values.
	ASM	Displays memory as assembler mnemonics.
	BCD	Displays or changes memory as 80-bit packed decimal values.
	BOOLEAN	Displays or changes memory as Boolean TRUE or FALSE values.
	BYTE	Displays or changes memory as an 8-bit unsigned value.
	CHAR	Displays or changes memory as ASCII characters.

Table 1-1 I²ICE™ System Commands Grouped by Function (continued)

Function	Command	Description
Memory Types (continued)	DWORD	Displays or changes memory as 32-bit unsigned values.
	EXTINT	Displays or changes memory as 64-bit signed values.
	INTEGER	Displays or changes memory as 16-bit signed values.
	LONGINT	Displays or changes memory as 32-bit signed values.
	LONGREAL	Displays or changes memory as 64-bit floating point values.
	MAP	Displays or sets physical locations for program memory.
	MAPIO	Displays or sets physical locations for I/O ports.
	POINTER	Displays or changes memory as selector:offset address pointers.
	REAL	Displays or changes memory as 32-bit floating point values.
	SELECTOR	Displays or changes memory as 16-bit unsigned values.
	SHORTINT	Displays or changes memory as 8-bit signed values.
	TEMPREAL	Displays or changes memory as 80-bit floating point values.
	WORD	Displays or changes memory as 16-bit unsigned values.
Number base	BASE	Displays or changes the number base.
Pointer	OFFSETOF	Returns the offset of a pointer value.
	POINTER	Displays or changes memory as selector:offset address pointers.
	SELECTOROF	Returns the selector or segment portion of a pointer.
Probe Microprocessor Signals	BTHRDY	Represents the source of the probe processor READY signal.
	PCHECK	Requests I²ICE protection checking (80286 probe specific)
	PINS	Displays the state of selected microprocessor signals.
	QSTAT	Selects 80186/80188 probe configuration mode.
	RSTEN	Enables the prototype to reset the probe processor.
Registers	ARMREG	Defines or modifies a debug register that contains arm, trigger, and disarm or debug sequences.

Table 1-1 I²ICE™ System Commands Grouped by Function (continued)

Function	Command	Description
Registers (continued)	BRKREG	Defines a register that contains break specifications.
	EVTREG	Defines a register that controls the event machines.
	REGS	Displays selected microprocessor registers in the current unit.
	SYSREG	Defines a register that contains system break specifications.
	TRCREG	Defines a register that contains user program tracing specifications.
Single-line Assembler	SASM	Loads memory with assembled mnemonics.
Stack	CALLSTACK	Displays the return address of procedures on the stack.
	STACK	Displays elements from the top of the stack.
Status	ACTIVE	Reports whether a variable exists at the current execution point.
	STATUS	Displays the current setting of selected debug environment conditions.
Stepping	ISTEP	Single-steps through user programs by machine-language instructions.
	LSTEP	Single-steps sequentially through user programs by high-level language instructions.
	PSTEP	Single-steps through user programs by high-level language instructions, treating procedures as one step.
Strings	CONCAT	Creates and displays a new string by concatenating.
	INSTR	Returns the index of a substring within a given string.
	LITERALLY	Defines, modifies, displays, or removes a name that the I²ICE system interprets as a previously-defined character string.
	NUMTOSTR	Converts an expression into ASCII code.
	STRLEN	Returns the number of characters in a string.
	STRTONUM	Converts a string to a numeric value.
	SUBSTR	Substring function.
	WRITE	Displays and formats character strings and numerical expressions.
Terminal Screen Control	CI	Allows a character to be read from the system terminal.
	CLEAREOL	Clears the screen from the cursor to the end of the line.

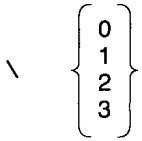
Table 1-1 I²ICE™ System Commands Grouped by Function (continued)

Function	Command	Description
Terminal Screen Control (continued)	CLEAREOS	Clears the screen from the cursor to the end of the screen.
	CURHOME	Moves the cursor to the upper left-hand corner of the screen.
	CURX	Displays the column number or moves the cursor to column x.
	CURY	Displays the row number or moves the cursor to row y.
	Paging	Controls the terminal display speed.
Time-out	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 based on memory access time.
Trace	ENABLE	Conditions the unit to accept system-level breaks and traces.
	PRINT	Formats and displays the contents of the trace buffer.
	SYMBOLIC	Enables or disables trace buffer symbolic display.
	TIMEBASE	Sets the counter source and the increment and formats the trace buffer timetag.
	TRCBUS	Controls the collection of bus information in the trace buffer.
	TRCREG	Defines a register that contains user program tracing specifications.
Unit Commands	\	Overrides the current default unit.
	UNIT	Displays or changes the current default unit.
	UNITHOLD	Causes the I²ICE system to pause while the user cable is moved.
	VERSION	Displays host version number and probe version numbers.
Wait-states	WAITSTATE	Specifies the number of memory wait-states inserted by the I²ICE system.

\

Overrides the current default unit

Syntax



Discussion

With the unit override command, you can override the default unit number for one command; it does not change the default unit (use the UNIT command to change the default unit). The unit override command remains in effect until another backslash or a carriage return is encountered in the command.

Block commands are the only I²ICE commands that cannot be preceded with a backslash. The unit override command cannot operate on the whole block because a block command contains other commands, and the backslash operates on only one command.

The unit number (0, 1, 2, 3) is in the current radix.

Example

1. Add the variable var__2 from unit 1 to the variable var in unit 2 (the default unit):

```
*UNIT = 2
*EVAL var + \ 1 var__2
```

Cross-Reference

UNIT

$

Pseudo-variable that displays or changes the current execution point

Syntax

$ [= *address*]

Where:

$	displays the register pair code-segment:instruction-pointer (CS:IP), which is the current execution point.
address	changes the current execution point by assigning the $ pseudo-variable an address, in either symbolic or numeric notation.

Discussion

The dollar sign ($) represents the program counter or fetch address of the next instruction. The dollar sign is a shorthand way of referring to the CS:IP registers.

Use the dollar sign as follows:

- to display the current execution point
- to change the current execution point
- to save the current execution point

CAUTION

If your program used the stack during the previous emulation, changing the execution point may cause incorrect operation when emulation resumes.

Examples

1. Display the current execution point:

```
*$
0200:05BAH
*CS:IP
0200:05BAH
```

2. Modify the current execution point:

```
*$ = coldstart + 4BH
*$ = $ - 1
*$ = 0ABH                    /*Absolute addressing is not recommended; see the */
                             /*Address entry */
*CS = SELECTOROF (warmstart)
*IP = OFFSET (warmstart)     /* Must change registers one at a */
                             /* time */
```

3. Save the current execution point as a variable:

```
*prog_start = $
*GO FROM prog_start
```

Cross-Reference

Address

ACTIVE

Reports whether a program variable is active at the current execution point

Syntax

ACTIVE (*name*)

Where:

name	is a program variable name.

Discussion

A static variable is always active; a dynamic variable is active only when the current execution point ($) is in the program block that contains the dynamic variable. Use the ACTIVE command to determine whether dynamic, stack-resident variables (such as parameters) have or have not been allocated at the current execution point. The ACTIVE command returns TRUE if the program variable named in *name* is active in the current program block; otherwise, the ACTIVE command returns an error message. For example, suppose a PL/M-86 program contains the following procedure:

```
avg: PROCEDURE (x, y) REAL;
     DECLARE (x, y) REAL;
     RETURN (x+y)/2.0;
END avg;
```

When the execution point is not within this procedure, the variables x and y are not active.

Changing the current execution point (by reassigning $) can cause inactive variables to become active (and vice versa). Keep in mind that the procedure prologue must be executed before its dynamic variables are active. Even though the ACTIVE command returns TRUE, if the prologue of a procedure that contains a dynamic variable has not been executed, accessing it produces undefined results.

The symbolic reference to the variable must be fully qualified unless the variable is within the current name scope. Changing the name scope does not affect whether a variable is active. However, changing the name scope can affect the amount of qualification needed to reference the variable.

When defining breakpoints and trace controls, you can refer to variables that are not active because the value of the variable is not accessed when defined.

Example

1. The following example defines a debug procedure that checks whether a variable is active before you try to access it.

```
*DEFINE PROC is_X_active = DO
. **IF ACTIVE (:util.avg.x) THEN
. . *WRITE 'x = ', :util.avg.x
. . *ELSE
. . *WRITE 'x not active'
. . *ENDIF
. *END
```

   Note that :util.avg.x is the fully qualified reference to the variable x.

Cross-References

Name
NAMESCOPE

ADDRESS

Displays or changes memory
as 16-bit unsigned values

Syntax

ADDRESS *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

ADDRESS *partition*	displays the contents of memory at that location as an address in the current base. An address is a 16-bit unsigned value.
partition	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to a 16-bit unsigned value for ADDRESS.
mtype	is any of the memory types except ASM. The Mtype entry in this encyclopedia lists mtypes.

Discussion

The ADDRESS command interprets the contents of memory as 16-bit unsigned values, overriding any type associated with the memory contents. Thus, ADDRESS .var1 displays the first word at the address of var1, regardless of the type of var1.

The information displayed by the ADDRESS command is identical to that displayed by the WORD and SELECTOR commands. However, when the memory type WORD is used as a data type in a program, it is interpreted as a 16-bit unsigned value. Both the ADDRESS and SELECTOR types, in that context, are interpreted as segments of address pointers.

Examples

The following examples assume a hexadecimal base.

1. Display a single value at the current execution point:

```
*ADDRESS $
0020:0004H 2EFA
```

2. Display several adjacent values:

```
*ADDRESS $ LENGTH 10
0020:0004H 2EFA 168E 0000 72BC 2E00 1E8E 0002 00EA 2101 0000 0814 0400 0814
0020:001EH 0400 0815 0400
```

3. Set a single value of type ADDRESS:

```
*ADDRESS 40:4 = 34AF
```

4. Set several contiguous values:

```
*ADDRESS 40:4 = 10FA, 3045, 107F
```

Display the values set:

```
*ADDRESS 40:4 LENGTH 3
0040:0004H 10FA 3045 107F
```

5. Set a range of locations to the same value (block set):

```
*ADDRESS 40:4 LENGTH 10 = 0
```

6. Set a repeating sequence of values:

```
*ADDRESS 40:4 LENGTH 10 = 1234, 5678, 9ABC, DEF0
```

Display the values set:

```
*ADDRESS 40:4 LENGTH 10
0040:0004H 1234 5678 9ABC DEF0 1234 5678 9ABC DEF0 1234 5678 9ABC DEF0 1234
0040:001EH 5678 0040:0020H 9ABC DEF0
```

7. Copy a value from one memory location to another:

```
*ADDRESS 40:4 = ADDRESS $
```

8. Copy several values (block move):

```
*ADDRESS 40:4 = ADDRESS $ LENGTH 10
```

9. Copy values with type conversion:

```
*ADDRESS 40:4 = BYTE .var2
```

An error message is displayed if the type on the right side of the equal sign cannot be converted to the type on the left. (Refer to the Expression entry in this encyclopedia for rules concerning type conversions.)

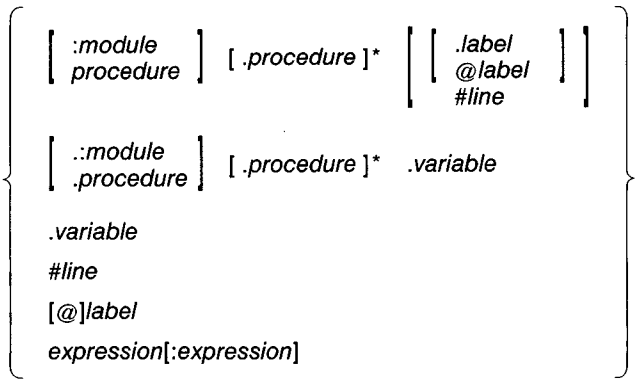
Cross-References

Expression
Mtype
Partition

Address

References program locations

Syntax



Where:

module	is the name of a module.
procedure	is the name of a procedure.
label	is the name of a program label. Use the @ sign when referring to a numeric label.
.label	is the name of a program label with the period (.) delimiter. Use the period delimiter unless the label is numeric.
@label	is the name of a program label with the at-sign (@) delimiter. Use the at-sign delimiter only when the label is a numeric label.
line	is a program line number.
variable	is the name of a program variable.
expression	is an expression that evaluates to an absolute address or a selector or an offset of a virtual address.

:	is the pointer operator (refer to the Expression entry in this encyclopedia for more information).
.	is the dot operator (refer to the Expression entry in this encyclopedia for more information).

Discussion

Addresses can be either virtual or absolute. A virtual address is a symbolic reference or a pointer expression (e.g., selector:offset). An absolute address is a numeric expression (e.g., 045ABH). Do not mix absolute and virtual addresses in the same expression.

Address specification depends on the number base. For instance, when the base is decimal and you specify a hexadecimal address, the H override character must appear following both the segment and offset portions of the expression. Note that address display conventions differ from address entry requirements. When a pointer address is displayed in hexadecimal, only one H appears after the entire expression (e.g., 458:0AFH).

A pointer value consists of a 16-bit selector component and a 16-bit offset component. The selector and offset are used to calculate the effective address. The exact method of calculation is processor-specific. In the 8086 processor, for example, the selector is shifted left four bits, then added to the offset to produce the effective address. (See the following section for information on 80286 probe addresses.) Regardless of the method, there is exactly one effective address corresponding to a given selector:offset pair. There are, however, numerous combinations of selectors and offsets that can result in a given effective address.

Several I²ICE commands require an *address* value. The following examples show command syntax containing *address* entries.

$ = *address*

NAMESCOPE = *address*

MAP *address* LENGTH *number-of-bytes* USER

When the system is expecting an *address*, the entry is converted to a pointer value if necessary. An expression used as an address is converted to a pointer according to the rules for type combination and assignment described in the Mtype entry in this encyclopedia.

Addresses for the 80286 Probe

The following subsections explain special aspects of 80286 addressing.

Virtual Addresses

Virtual addresses are symbolic addresses, selector:offset pairs, or LDT-selector:selector:offset triplets (LDT stands for local descriptor table). You must use a virtual address if the pseudo-variables SEL286 and PCHECK are both TRUE.

The Selector:Offset Pair for the 80286 Probe

When the 80286 probe performs 8086 address translation (SEL286 = FALSE), a virtual address is a selector:offset pair. The 80286 probe constructs the physical address by shifting the selector left by four bits and adding the offset. The physical address can be up to 20 bits long. If you specified the physical address directly, the address can be up to 24 bits long.

When the 80286 probe performs 80286 address translation (SEL286 = TRUE), the virtual address is a selector:offset pair or an LDT-selector:selector:offset triplet.

The selector is an offset into either a local descriptor table (LDT) or a global descriptor table (GDT). It points to a segment descriptor that contains a base address. The addition of this base address and the offset is the final physical address.

The current LDT is identified by the contents of the local descriptor table register (LDTR). The LDTR contains the LDT selector, which is an offset into the GDT that points to an LDT descriptor. The LDT descriptor contains the base address of the LDT.

Similarly, the GDT is identified by the contents of the global descriptor register (GDTR).

The LDT-Selector:Selector:Offset Triplet

You can specify the LDT selector as part of the address. Your specification overrides the LDT selector currently stored in the LDTR, so that the triplet uses a local descriptor table not necessarily currently selected by the LDTR.

Absolute Addresses

An absolute address can be up to 24 bits long. You cannot use an absolute address if the pseudo-variables SEL286 and PCHECK are both TRUE.

Examples

1. An integer entry assumes that the last hexadecimal digit is the offset. For example:

```
*$=20H:6                          /*Integer entry*/
*$
0020:0006H                        /*New address of $ is the integer
                                    converted to a pointer*/
```

2. Symbolic references, pointer expressions (like CS:IP), and POINTER expressions (a memory type) are already pointers and need no conversion. For example:

```
*$ = :mod1                    /*Symbolic reference to the beginning of a module*/

*$ = :mod1.proc5              /*Symbolic reference to the beginning of a procedure*/

*$ = :mod1 #57                /*Symbolic reference to a program line number*/

*$ = 123:4                    /*Pointer expression, no interpretation */
```

Cross-References

Address protection
Address translation
Expression
PCHECK
SEL286

Address protection
80286 probe specific

The 80286 has two modes of operation: real address mode (sometimes called compatible mode) and protected mode.

In real address mode, the 80286's operation is similar to the 8086's operation. There is no virtual memory capability; the physical address space is 1M byte plus 64K. Note that if you try to access a 24-bit address in real address mode, the I²ICE system drops the upper four bits (i.e., zeros them out).

In protected mode, the 80286 allows multitasking, multi-user, virtual memory systems. The virtual address space is 1G byte per task; the physical address space is 16M bytes.

The 80286 powers up in real address mode at address 0FFFFF0H. It enters protected mode when you set the protection enabled flag (PEF) in the machine status word (MSW) to 1.

Privilege Levels

The current privilege level determines what memory locations (code and data) a task can access. The I/O privilege level determines at which current privilege level a task must be executing to execute an I/O instruction. There are four privilege levels: 0, 1, 2, and 3. Level 0 has the most privilege; level 3 has the least privilege. A task can execute at only one of the four levels, called the current privilege level (CPL).

Visibility

A data segment is visible to a task only when the segment's descriptor privilege level (DPL) is equal to or lower (numerically higher) than the CPL. A protection violation occurs if a user program tries to access data belonging to a segment of higher (numerically lower) privilege.

Conforming segments can be read from any privilege level. When a conforming segment is read by tasks of lower (numerically higher) privilege, the CPL remains the same. The conforming segment is executed at a lower (numerically higher) privilege level. An executable, non-conforming segment is visible to a task only when the segment's DPL is equal to the CPL.

The access field of a segment descriptor contains the DPL. The code-segment selector contains the CPL.

The selector pointing to the segment descriptor contains the requested privilege level (RPL). The RPL restricts individual data accesses. If a selector's RPL is numerically larger than the CPL, then the 80286 uses the RPL instead of the CPL when determining the visibility of a segment.

Transferring Control

A task can transfer control either directly or through a call gate.

When the task transfers control directly, the new execution address must be at the same privilege level as the old execution address; that is, the DPL of the new code-segment descriptor must be equal to the CPL.

When the task transfers control through a call gate, the privilege level of the call gate must be equal to or lower (numerically higher) than the CPL; that is, the DPL of the call gate is equal to or greater than the CPL.

The new execution address must be at the same or higher (numerically lower) privilege than the call gate; that is, the DPL of the new code segment descriptor must be equal to or less than the CPL.

Typically, an application program runs with privilege level 3. When the application program requires the use of the operating system, it calls a routine of higher (numerically lower) privilege.

Protection Checking

The I²ICE system's protection checking is distinct from 80286 protection checking, as follows:

- 80286 protection checking — the 80286 must be in protected mode. For example, if a user program running with privilege level 3 tries to access data in a segment of privilege level 2, a protection violation occurs.
- I²ICE protection checking — the I²ICE system does protection checking when the PCHECK pseudo-variable is TRUE. I²ICE protection checking concerns the display and modification of 80286 registers and memory locations with I²ICE commands.

When you access registers, the effect of PCHECK depends on whether the 80286 is in real or protected mode. When you access memory, the effect of PCHECK depends on the setting of the SEL286 pseudo-variable.

Cross-References

80286 flags
80286 registers
PCHECK
SEL286

Address translation
80286 probe specific

The 80286 has a virtual address space of 1G byte per task. The 80286 represents a virtual address as a selector:offset pair. The selector and offset are each 16 bits long. The selector contains 14 address bits; its other two bits define the requested protection level (RPL). With the 16 address bits from the offset, the result is a 30-bit virtual address. With 30 bits, you can address 1G byte of memory.

The 80286 probe performs either 8086 or 80286 address translation. When the probe performs 8086 address translation, it shifts the selector left by four bits and then adds the offset. The result is a 20-bit physical address. With 20 bits, you can address 1M byte of memory.

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 286 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 286 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.

In 80286 address translation, the selector is itself an offset into either the global descriptor table (GDT) or a local descriptor table (LDT). There is only one GDT, but there may be several LDTs. Both the GDT and the LDTs reside within the virtual memory space. Of the 14 address bits in the selector, one bit (the table indicator (TI) bit) selects either the GDT or an LDT. The other 13 bits represent an offset into the selected table. Note: The GDT cannot be indexed with a value greater than 255.

The 13-bit offset points to a segment descriptor. The segment descriptor contains access rights, a base address, and the segment limit. The final physical address is the sum of the base address from the segment descriptor and the offset from the virtual address. Unlike 8086 translation, the selector is not shifted left before the addition. The result is a 24-bit physical address. With 24 bits, you can address 16M bytes of memory. The 80286 has a 1G-byte virtual address space and a 16M-byte physical address space.

Figure 1-1 illustrates the 80286 virtual address translation.

The following 80286 registers are involved with address translation:

GDTR	Global descriptor table register
LDTR	Local descriptor table register
CS	Code segment register
DS	Data segment register
ES	Extra segment register
SS	Stack segment register

The GDTR and the LDTR

The GDTR contains the GDT descriptor. The GDT descriptor locates the GDT in memory. The GDT descriptor contains the GDT's base address and limit. The GDT limit is the range of addresses above the GDT base address that make up the GDT.

The LDTR selector is an offset into the GDT. This offset points to an LDT descriptor. The LDT descriptor, an entry in the GDT, contains the LDT's access rights, base address, and limit.

The LDTR contains an explicit cache. The LDTR selector is 16 bits long, but the register is actually 64 bits long. The other 48 bits belong to the explicit cache. When you load the selector portion of the LDTR, the 80286 copies the specified LDT descriptor from the GDT into the LDTR's explicit cache. Until you change the LDTR selector, the 80286 does not have to access the GDT for a new LDT descriptor.

Address translation (80286) continued

Figure 1-2 shows the relationship of the two descriptor tables (the GDT and the LDT) and the two registers (the LDTR and the GDTR).

The Segment Registers

A segment register identifies a segment descriptor. This segment descriptor is either in the GDT or in the current LDT. The selector portion of a segment register chooses the GDT or the LDT and provides an offset into the selected table. The 80286 multiplies this offset by eight. A descriptor table entry is eight bytes long.

Each of the 80286 segment registers also contains an explicit cache. When you load the selector portion of a segment register, the 80286 copies the specified segment descriptor from the

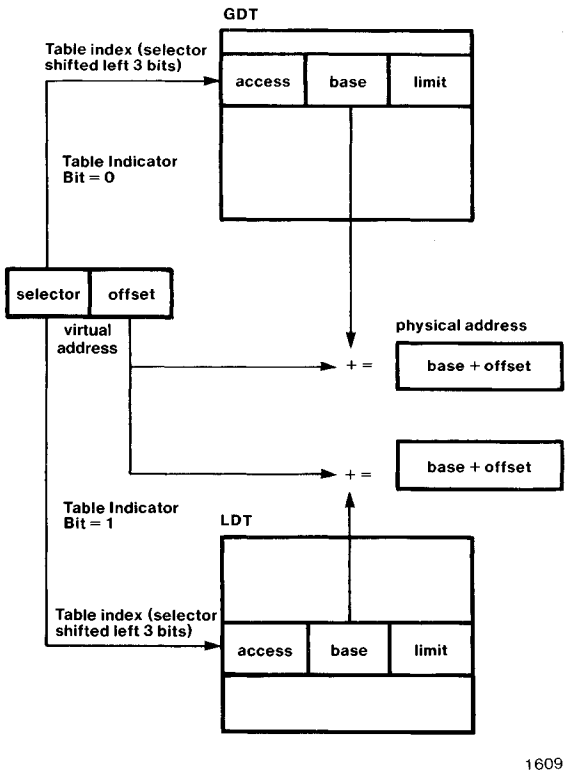


Figure 1-1 80286 Virtual Address Translation

GDT into the explicit cache. Until you change the segment selector, the 80286 does not have to access a descriptor table for a segment descriptor.

Figure 1-3 shows the relationship of the segment registers and the two descriptor tables (the GDT and the LDT).

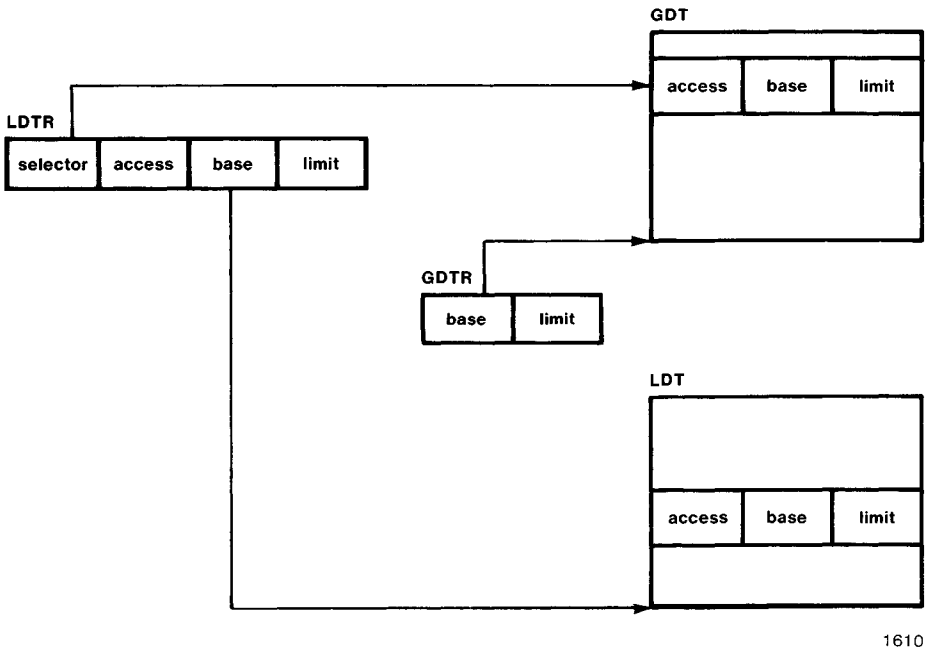


Figure 1-2 The Descriptor Table Registers and the Descriptor Tables

Cross-References

80286 registers
PCHECK
SEL286

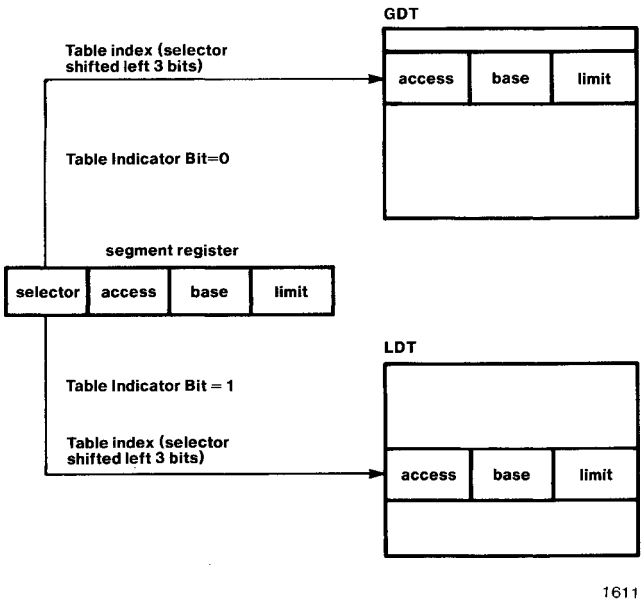
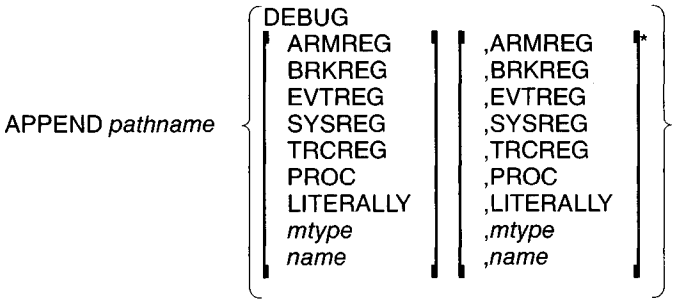


Figure 1-3 The Segment Register and the Descriptor Tables

APPEND

Saves the definitions of debug objects from memory to a file

Syntax



Where:

APPEND *pathname* DEBUG	adds all debug objects currently defined in memory to the file *pathname*.
APPEND *pathname debug-object-type*	adds all debug objects of the specified type (ARMREG, BRKREG, etc.) to the file *pathname*.
APPEND *pathname debug-object-name*	adds the named debug objects to the file pathname. Program memory values are not saved.
pathname	is the fully-qualified reference of the file to which you want to append the debug objects. For further information on *pathname*, see the Pathname entry in the *FICE™ System Reference Manual*.

Discussion

The APPEND command saves the definitions of debug procedures, LITERALLYs, debug memory types, and debug registers to a disk file. The values of debug memory types are not saved.

The APPEND command does not edit the file; it saves information to an existing file. When the named file does not exist, APPEND creates it. Additionally, if a debug object already exists in the APPEND file, both versions are saved but only the most recent definition is restored (with the INCLUDE command).

NOTE

Do not repeat keywords in the command. For example, the following command is incorrect:

```
APPEND :f1:deb.001 PROC, PROC
```

Examples

The following examples assume that the debug objects have been defined and appended to the file :f2:debug.inc and that the base is hexadecimal. (If you have an IBM PC host, disregard the symbol ":f2:". If the file is in your current disk directory, append to the file using the command: APPEND debug.inc DEBUG. If the file is on another drive, replace :f3: with d:, where d is the letter of the file's disk drive.)

```
*DIR DEBUG                          /* List existing debug objects to the terminal */
I      .      .      .      . byte          5
J      .      .      .      . byte          10
K      .      .      .      . byte          1B
SUM    .      .      .      . integer       +344
P      .      .      .      . pointer       0020:0012H
X_VALUE .     .      .      . word          FFC0
BASE_ADDR .   .      .      . literally     'BYTE 1000H:0H'
WHERE  .      .      .      . proc
*APPEND :f2:debug.inc DEBUG
```

1. Create and append additional debug objects to an existing file.

```
*DEFINE DWORD s_factor                  /* Create debug objects */
*DEFINE DWORD r_factor
*APPEND :f2:debug.inc s_factor, r_factor
```

Another way to save s_factor and r_factor is as follows:

```
*DEFINE DWORD s_factor
*DEFINE DWORD r_factor
*APPEND :f2:debug.inc DWORD
```

2. Restore and list the debug objects from the file.

```
*INCLUDE :f2:debug.inc
*define byte I
*define byte J
*define byte K
*define integer SUM
*define pointer P
*define word X_VALUE
*define literally BASE_ADDR = 'BYTE 1000H:0H'
*define proc WHERE = EVAL $ LINE
*define dword S_FACTOR
*define dword R_FACTOR
```

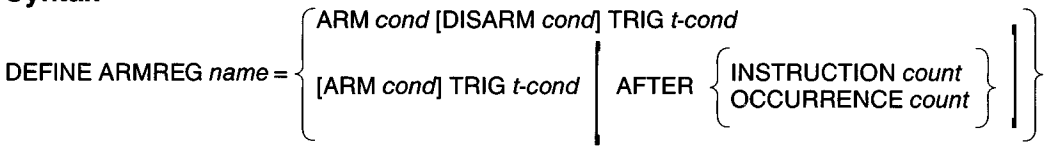
Cross-References

ARMREG
BRKREG
EVTREG
LITERALLY
Mtype
Name
Pathname
PROC
SYSREG
TRCREG

ARMREG

Defines or modifies a debug register that contains arm, trigger, and disarm sequences or delay sequences

Syntax

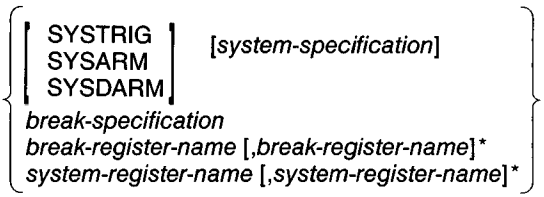


Where:

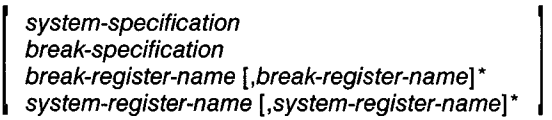
DEFINE ARMREG *name* creates a debug break register called *name*. Follow the equal sign (=) with an arm, trigger, disarm, or delay specification to define the break criteria.

ARM *cond* allows triggering. ARM condition must precede TRIG and DISARM.

t-cond is one of the following:



cond is one of the following:



DISARM *cond* prevents triggering. The DISARM *cond* must be preceded with an ARM *cond*. When the DISARM *cond* is met, the I²ICE system searches for the ARM condition again.

TRIG *t-cond*	triggers a break when TRIG *t-cond* and ARM *cond* (if present) are true. When no ARM *cond* is specified, the I²ICE system immediately searches for the TRIG *t-cond*.
AFTER	qualifies the trigger condition with a delay factor. Triggers without an AFTER *cond* define break conditions. Triggers with an AFTER *cond* define break conditions after a delay. The AFTER and DISARM clauses are mutually exclusive.
SYSTRIG	triggers any enabled I²ICE units and performs the programmed action when the *system-specification* is met. Refer to the ENABLE entry in this encyclopedia for a description of the unit enabling process.
SYSARM	arms I²ICE units that are enabled when the *system-specification* is met, which can then respond to the system trigger line (SYSTRIG).
SYSDARM	disarms any I²ICE units that are enabled when the *system-specification* is met, which then cannot respond to the system trigger line (SYSTRIG).
system-specification	is a bus address, bus data, logic clip information, the buffer full condition, or probe processor status. Complete *system-specification* syntax is in the System specification entry in this encyclopedia.
break-specification	is a numeric or symbolic address (line number, module name, label, or a list of addresses). Complete *break-specification* syntax is in the Break specification entry in this encyclopedia.
break-register-name *system-register-name*	refers to previously defined registers of type BRKREG or SYSREG.
count	is a number or expression that evaluates to a positive whole number in the current base.
INSTRUCTION *count*	breaks emulation after the specified number of machine language instructions have been executed following the trigger.
OCCURRENCE *count*	breaks emulation after the specified number of trigger conditions are met.

Discussion

The ARMREG command sets conditional breakpoints that allow breaking within windows. A break window is opened when an arm condition is encountered and closed when a disarm condition is encountered. There are two ways to stop emulation based on arming sequences. One way is using the GO command; the other is using a debug register called ARMREG (arm register) in the GO command.

Consider using ARMREGs in three cases:

- Use ARMREG to trigger. Used this way, ARMREG operation is identical to BRKREG or SYSREG operation.
- Use ARMREG to ARM a trigger. With ARMREG you can selectively trigger only after an arm qualification is met. Furthermore, you can disarm the trigger. This way, trigger events are screened. The probe recognizes the trigger condition only when armed.
- Use ARMREG to trigger after a delay. The effect of a trigger is specified in the TRIG clause. Arming the system, disarming the system, or triggering a break in emulation are examples of a trigger effect. There are two ways to delay the effect of a trigger. You can tell the I²ICE system how many instructions to execute *after* the trigger point before activating the trigger effect. Alternatively, you can tell the I²ICE system how many triggers must occur *before* the trigger effect.

You can optionally enclose ARMREG specifications in a DO/END block.

How to Specify an ARMREG

Figure 1-4 simplifies the syntax diagram by showing a tree of legal syntax combinations.

Triggering

Triggering (controlled by TRIG) causes a defined action, such as an emulation break, to occur.

Counting

With the AFTER clause you can count events. Events can be the number of instructions executed after the trigger point or the number of occurrences of the trigger condition. The count sequence begins at the first trigger. The break occurs when the count is satisfied.

Manipulating ARMREGs

Manipulate an ARMREG by referring to it by name. You can manipulate ARMREGs in the following ways:

- Create an ARMREG with the DEFINE command
- Delete an ARMREG from memory with the REMOVE command
- List ARMREG names with the DIR command
- Save (or restore) an ARMREGs to (or from) a file with the PUT, APPEND, or INCLUDE commands
- Display an ARMREG with the command ARMREG
- Execute an ARMREG with the GO USING command
- Modify an ARMREG with the editor

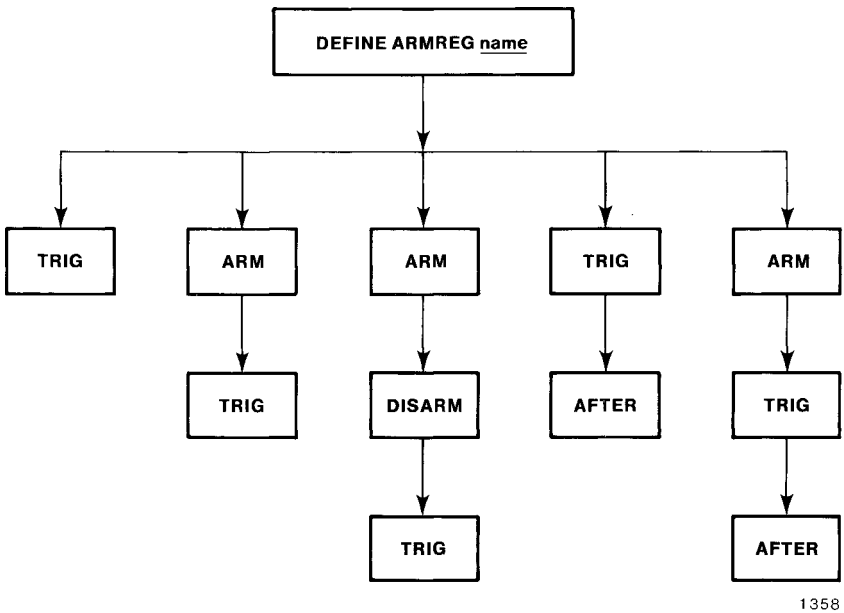


Figure 1-4 Tree of Legal Syntax

NOTE

Defining new break specifications using an old ARMREG name destroys the old definition in memory. An error results if you try to assign the name of an ARMREG to any other type of debug object in memory.

Retrieving a saved ARMREG that has the same name as an existing ARMREG overwrites the one in memory.

An error results when you try to retrieve a saved ARMREG from a file that has the same name as any other debug object in memory.

Using ARMREGs with Multiple Units

The keywords SYSTRIG, SYSARM, and SYSDARM indicate actions caused by arm registers. Other units in the I²ICE system must be enabled to respond to a system action. Refer to the ENABLE and SYSTEM entries in this encyclopedia for details.

Limitations on Arm Specifications in the GO Command

Arm registers can contain any number of specifications. The GO command's ability to execute the specifications is, however, limited by the number of word recognizers available.

Word Recognizers

Word recognizers are the programmable portion of the internal execution state machine that compares user match specifications with conditions on the bus it monitors. When the match occurs, the state machine halts emulation. Refer to the Event machines entry in this encyclopedia for details.

Word recognizer use is governed internally. You cannot know precisely how many word recognizers are used in any given specification. A good rule of thumb is that one- or two-range (partition) specifications or four-location specifications are the upper limit.

The I²ICE system reports an error when the word recognizer limit is exceeded. You can either simplify the specification or use the DEFINE EVTREG construct.

Restrictions

The following restrictions apply when using more than one unit:

- Only one unit is allowed to control system arming (SYSARM and SYSDARM).
- SYSARM, SYSDARM, and SYSTRIG cannot be used with SYSTRACE on the same unit.

Example

1. The following example shows how to trigger a break by specifying an ARMREG that contains an arm, trigger, and disarm sequence.

   The source code contains procedures A, B, and C that access utility procedure X. A problem is discovered when procedure B calls utility X. To trap this particular bug, select the arm register to conditionally break emulation.

   The arm register constructed arms the trigger to the current probe when procedure B is addressed and disarms the trigger whenever the probe addresses any other procedure. The break is triggered by any call to utility procedure X from procedure B. Note that the I²ICE system interprets symbolic references to procedures or modules as partitions.

```
*EVAL :mod_a.utility_x LINE
:mod_a #120
*DEFINE ARMREG xtest = DO
**ARM procedure_b
**DISARM OUTSIDE procedure_b
**TRIG :mod_a #120
**END
*GO USING xtest
Probe 0 stopped at :mod_a #120+3 because of execute break
  Break register is XTEST
```

Cross-References

Break specification
ENABLE
Event machines
Name
SYSTEM
System specification

ASM

Displays memory as assembler mnemonics

Syntax

ASM *partition*

Where:

partition	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-instructions*.

Discussion

The format of the display depends on the number of addresses referenced. A single address reference displays the first instruction at that address. A range of addresses, specified as *address* TO *address*, displays all instructions that start within the range. An instruction is displayed if its first byte is within the partition, even if subsequent bytes are outside. To specify an exact number of instructions to be displayed, use the form *address* LENGTH *number-of instructions*. When *partition* is a symbolic reference to a procedure, ASM disassembles the entire procedure.

Disassembled instructions and comments appear on the terminal in columns. They are, from left to right: address, hexadecimal object values, opcode mnemonics, and operands (if any). Comments appended to the operands provide additional information, such as the types of jumps, calls, and returns, the address of a branch relative to the current execution point ($), and the decimal equivalents of hexadecimal values. Refer to the example section of this command for a sample display.

The disassembly includes symbols and module and line number information when the following three conditions are satisfied:

- If SYMBOLIC = TRUE (refer to the SYMBOLIC pseudo-variable entry in this encyclopedia for details).
- If the segment and offset values can be determined from the *address*.
- If the symbol table contains an exact match to the beginning of an instruction in the *partition*.

When an absolute address is used to specify the partition, the disassembly begins without line number information. If a jump or call instruction is subsequently encountered and the disassembler can determine the true segment and offset values, then the display includes module and line number identification.

Examples

1. Display a single instruction:

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

2. The following example shows the disassembly of several instructions. It shows the format used by the disassembler for absolute addresses. It also shows the addition of module and line number information after a CALL instruction has allowed the disassembler to identify segment and offset values within the range of available line numbers.

```
*$
0020:0006H
*ASM 206H LENGTH 30T
    000206H FA              CLI
    000207H 2E8E160000      MOV  SS,CS:WORD PTR 0000H
    00020CH BC2000          MOV  SP,0020H ;+32T
    00020FH 2E8E1E0200      MOV  DS,CS:WORD PTR 0002H
    000214H EA0A002100      JMP  0021H:000AH
    000219H AA              STOS  ES:BYTE PTR [DI]
    00021AH 8BEC            MOV  BP,SP
    00021CH FB              STI
    00021DH B80C00          MOV  AX,000CH ;+12T
    000220H 8906080D        MOV  WORD PTR 0008H,AX
    000224H 8C1E0A00        MOV  WORD PTR 000AH,DS
    000228H B11E            MOV  CL,1EH ;+30T
    00022AH 51              PUSH  CX
    00022BH B92700          MOV  CX,0027H ;+39T
    00022EH 51              PUSH  CX
    00022FH 1E              PUSH  DS
    000230H 50              PUSH  AX
    000231H E81700          CALL  (TEST2PROC)A=001BH ; $+26
:TEST2#2
 0021:0024H 55            PUSH  BP
 0021:0025H 8BEC          MOV  BP,SP
#4
 0021:0027H 817E040100    CMP  WORD PTR [BP+04H],1
 0021:002CH 7403          JZ  (#5)A=0031H ; $+5
 0021:002EH E90600        JMP  (#6)A=0037H ; $+9
#5
 0021:0031H C7060E000000  MOV  WORD PTR 000EH,0
#6
 0021:0037H 5D            POP  BP
 0021:0038H C20200        RET  2    ; NEAR
```

ASM continued

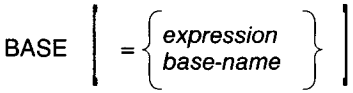
Cross-References

Address
Partition
SYMBOLIC

BASE

A pseudo-variable that display or changes the number base

Syntax



Where:

BASE	displays the current number base. The default base is decimal.
expression	changes the default BASE. The *expression* must evaluate to 2, 10, or 16.
base-name	changes the default base. Names available are BINARY, DECIMAL, and HEX.

Default

DECIMAL

Discussion

The BASE pseudo-variable controls the default number base for terminal input and output. You can use BASE as follows:

- To display the default base (e.g., BASE).
- To change to a new base (e.g., BASE = 2 or BASE = BINARY).
- As a variable in expressions (e.g., *variable* = BASE). The type of the BASE pseudo-variable is BYTE.

When you change the base using an expression, if the expression does not evaluate to 2, 10, or 16 (decimal), the number base does not change and an error results. Unless otherwise specified, all expressions are evaluated in the current base. To override the current base, use an explicit suffix: Y for binary, T for decimal, H for hexadecimal.

NOTE

The BASE variable is always global. When the number base is changed by executing the BASE command, the change happens immediately, even if the change command is within a debug procedure definition, a block command, or in a command line with multiple commands.

Examples

1. Display the current base:

```
*BASE
DECIMAL
```

2. Change to binary radix:

```
*BASE = 2T
*
```

or

```
*BASE = 10Y
*
```

or

```
*BASE = BINARY
*
```

3. Change to decimal radix:

```
*BASE = 101
*
```

or

```
*BASE = DECIMAL
*
```

4. Change to hexadecimal radix:

```
*BASE = 16T
*
```

or

```
*BASE = 10H
*
```

or

```
*BASE = HEX
*
```

5. Use BASE in an expression:

```
*VAR1 = BASE * 2
```

6. The following example shows a command block in which the numbers will be in base two. The block saves the current BASE, switches to BINARY radix for the commands, and then restores the previous BASE.

```
*DO
. *DEFINE BYTE TEMPRADIX = BASE
. *BASE = 2T
. *                                  /* Commands using binary numbers */
. *BASE = TEMPRADIX
. *END
```

Cross-Reference

Expression

BCD

Displays or changes memory as
80-bit packed decimal values

Syntax

BCD *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

BCD *partition*	displays the location specified in *partition* as a binary coded decimal number in decimal.
partition	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to an 80-bit packed decimal value for BCD.
mtype	is any memory type except POINTER, BOOLEAN, CHAR, and ASM.

Discussion

The BCD (binary coded decimal) command interprets the contents of memory as signed 80-bit packed decimal values, overriding any type associated with the memory contents. Thus, BCD .var1 displays the 80-bit packed decimal value that begins at the address of var1, regardless of the type of var1.

Examples

The following examples assume a decimal base. An H is required after both the segment and the offset to specify hexadecimal addresses when the base is decimal.

1. Display a single value:

```
*BCD $
0020:0004H    +73220000169435600
```

2. Display several adjacent values:

```
*BCD $ LENGTH 5
0020:0004H   +73220000169435600   +21010150000224940
0020:000EH   +15040008140400081   +81704000817040O
0020:0022H   +12040008140400081
```

3. Set a single value of type BCD:

```
*BCD 40H:4H = -1234567890
```

4. Set several contiguous values:

```
*BCD 40H:4H = -1234567890,+1000000000
```

Display the values set:

```
*BCD 40H:4H LENGTH 2
0040:0004H -1234567890          +1000000000
```

5. Set a range of locations to the same value (block set):

```
*BCD 40H:4H LENGTH 10 = 0
```

6. Set a repeating sequence of values:

```
*BCD 40H:4H LENGTH 6 = 11111111111, -2222222222
```

Display the values set:

```
*BCD 40H:4H LENGTH 6
0040:0004H +1111111111          -2222222222
0040:0018H +1111111111          -2222222222
0040:002CH +1111111111          -2222222222
```

7. Copy a value from one memory location to another:

```
*BCD 40H:4H = BCD $
```

8. Copy several values (block move):

```
*BCD 40H:4H = BCD $ LENGTH 10T
```

9. Copy values with type conversion:

```
*BCD 40H:4H = BYTE .var2
```

An error message is displayed if the type on the right side of the equal sign cannot be converted to the type on the left. (Refer to the Expression entry in this encyclopedia for the rules concerning type conversions.)

Cross-References

Expression
Mtype
Partition

BOOLEAN

Displays or changes memory as Boolean TRUE or FALSE values

Syntax

BOOLEAN *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

BOOLEAN *partition*	displays the location specified in *partition* as a Boolean value (TRUE or FALSE).
partition	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to a TRUE or FALSE value. Only the least significant bit (LSB) of the result is tested. If the LSB is 1, the BOOLEAN value is TRUE; if the LSB is 0, the BOOLEAN value is FALSE.
mtype	is any of the memory types except POINTER, CHAR, and ASM.

Discussion

The BOOLEAN command interprets the contents of memory as TRUE or FALSE values, overriding any type associated with the memory contents. Thus, BOOLEAN .var1 displays TRUE or FALSE, depending on the LSB of the byte at the address of the program variable var1, regardless of the type of var1.

Examples

The following examples assume a hexadecimal base.

1. Display a single value:

```
*BOOLEAN $
0020:0004H TRUE
```

2. Display several consecutive values:

```
*BOOLEAN $ LENGTH 7
0020:0004H FALSE FALSE FALSE FALSE FALSE TRUE TRUE
```

3. Set a single value of type BOOLEAN:

```
*BOOLEAN 40:4 = FALSE
```

4. Set several contiguous values:

```
*BOOLEAN 40:4 = TRUE, FALSE, FALSE
```

   Display the values set:

```
*BOOLEAN 40:4 LENGTH 3T
0040:0004H TRUE FALSE FALSE
```

5. Set a range of locations to the same value (block set):

```
*BOOLEAN 40:4 LENGTH 10 = TRUE
```

6. Set a repeating sequence of values:

```
*BOOLEAN 40:4 LENGTH 10T = TRUE, FALSE
```

   Display the values set:

```
*BOOLEAN 40:4 LENGTH 10T
0040:0004H TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
TRUE FALSE
```

7. Copy a value from one memory location to another:

```
*BOOLEAN 40:4 = BOOLEAN $
```

8. Copy several values (block move):

```
*BOOLEAN 40:4 = BOOLEAN $ LENGTH 10T
```

9. Copy a value with type conversion:

```
*BOOLEAN 40:4 = BYTE .var2
```

An error message is displayed if the type on the right side of the equal sign cannot be converted to the type on the left. (Refer to the Expression entry in this encyclopedia for the rules concerning type conversions.)

Cross-References

Expression
Mtype
Partition

Boolean condition

A condition that evaluates to a Boolean value

A Boolean condition is either a value of type BOOLEAN (TRUE or FALSE) or an expression that uses one of the following relational operators:

= = equal to

> greater than

< less than

> = greater than or equal to

< = less than or equal to

< > not equal to

Break specification

Defines a break specification

Syntax

[OUTSIDE] *partition* [,[OUTSIDE] *partition*]*

Where:

partition	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*. An address can be virtual or absolute. The value of LENGTH is interpreted as the number of bytes, regardless of the memory type at that location. (Refer to the Address and Partition entries in this encyclopedia for more details.)
OUTSIDE	tells the I²ICE system to recognize only addresses that are not in the partition (a logical NOT function).

Discussion

The term *break-specification* has a special meaning in the syntax of the I²ICE system commands. The execution control commands (i.e., GO, BRKREG, ARMREG, TRCREG, and EVTREG) use this term in their syntax definitions.

Examples

The following examples show four ways to set a break specification in the GO command or a register.

1. A module and procedure name as a single address:

   :initmod.initio

2. A module, line number, and procedure name in a list:

   :initmod #5, :initmod.initio

3. A partition of virtual addresses using the TO form:

   21:0A7 TO 21:0D0

4. An exclusive partition of virtual addresses using the LENGTH form:

   OUTSIDE 21:11 LENGTH 20

Cross-References

Address
Partition

BRKREG

Defines a register that contains break specifications

Syntax

DEFINE BRKREG *name* = *break-specification* [,*break-specification*]* [CALL *dproc*]

Where:

DEFINE BRKREG *name* = *break-specification*	creates a debug break register called *name*. Following the equal sign (=) with a *break-specification* defines the break criteria.
name	is the name of the debug procedure called when the *break-specification* is met.
break-specification	is the address of an executable statement expressed either numerically (e.g., 0465H) or symbolically (e.g., a line number). (The Break specification entry in this encyclopedia describes the syntax in detail.)
CALL *dproc*	calls the debug procedure named when the *break-specification* is met. The debug procedure must return TRUE (meaning a break is to occur) or FALSE (meaning emulation will continue without breaking).

Discussion

Break specifications stop emulation when the target line of code is executed. You can stop emulation using break specifications in two ways. One way is to specify the breakpoint in the GO command; the other is to use a debug register called a BRKREG (break register) in the GO command (with the USING option).

Manipulating BRKREGs

Manipulate a BRKREG by referring to its name. You can manipulate BRKREGs in the following ways:

- Create a BRKREG with the DEFINE command
- Delete a BRKREG from memory with the REMOVE command
- List BRKREG names with the DIR command
- Save a BRKREG on file with the PUT or APPEND commands

- Restore a BRKREG from a file with the INCLUDE command
- Display a BRKREG with the BRKREG command
- Execute a BRKREG with the GO USING command
- Use a BRKREG as part of the DEFINE ARMREG specification
- Modify a BRKREG with the editor

Because BRKREGs are referred to by name, you can reuse break specifications without re-entering them. The GO command allows BRKREG lists. By defining BRKREGs, you can switch breakpoints in a GO statement by changing BRKREG names.

NOTE

Defining new break specifications using an old BRKREG name destroys the old definition in memory. An error results if you try to assign a BRKREG name to any other debug object in memory.

Restoring a saved BRKREG that has the same name as an existing BRKREG overwrites the one in memory.

An error occurs when you try to restore a saved BRKREG that has the same name as any other debug object in memory.

You can optionally enclose BRKREG specifications in a DO/END block.

Using the Optional Call

When a trigger occurs because of a BRKREG that includes a CALL, the CALL transfers control to the named debug procedure. This debug procedure must return a Boolean value (TRUE or FALSE) to the BRKREG. If it returns TRUE, emulation stops and the break message is printed. If it returns FALSE, emulation resumes. A CALL does not execute in real-time.

Emulation halts if a Boolean value is not returned or there is an error in the called debug procedure. An error message indicates that the halt was not caused by a normal execution break.

Restrictions

A BRKREG may contain any number of specifications. The GO command's ability to execute these specifications, however, is limited by the number of word recognizers available.

BRKREG continued

Word recognizers are the programmable portion of the internal execution state machine that compares user match specifications with conditions on the bus it monitors. When a match occurs, the state machine halts emulation. Refer to the Event machines entry in this encyclopedia for details.

Word recognizer use is governed internally. You cannot know precisely how many word recognizers are used in any given specification. A good rule of thumb is that one- or two-range (partition) specifications or four-location specifications are the upper limit. The I²ICE system indicates when the word recognizer limit is exceeded.

Example

1. The following example defines a procedure, a character variable, and a BRKREG. The procedure is named QUERY. The procedure QUERY is called from a BRKREG named THIS_ROUND. The procedure QUERY displays the value of the current probe processor's registers and flags and asks if the user wants to stop emulation. Entering Y returns a TRUE to the calling BRKREG and stops emulation.

```
*DEFINE PROC query = DO
. *REGS
. *WRITE USING ( "Do you want to break?", > )          /*Screen message */
. *DEFINE CHAR ccc = CI                                /*Accept terminal */
                                                       /*input*/
. *WRITE ccc
. *IF ccc = = 'Y' then return true                     /*Test and return Boolean value*/
. . *ELSE return false
. . *ENDIF
. *END
*DEFINE BRKREG this_round = :helpentry CALL query      /* Call query at the
                                                       symbolic addr "helpentry"*/
*GO FROM display USING this_round                      /* Return true to BRKREG */
----- REGISTERS FOR UNIT 0000 ----
AX=4          BX=63A          CX=0          DX=2
CS=5588       DS=188          SS=104        ES=0
IP=46C7       BP=634          SP=624        SI=830
DI=3A2
FLAGS : ZFL PFL
Do you want to break?Y
Probe 0 stopped at :helpentry #3 because of execution break
  Break register is THIS_ROUND
```

Cross-References

Break specification
Event machines
Name

BTHRDY

8086/8088 probe specific

A pseudo-variable that determines the source of the probe processor READY signal

Syntax

```
BTHRDY | =TRUE
       | =FALSE
       | =boolean-expression |
```

Where:

BTHRDY	displays the current setting.
TRUE	uses the logical AND of the prototype READY signal and the I²ICE system READY signal to determine the number of wait-states.
FALSE	uses READY that depends on current mapping. For example, if memory is mapped to USER, the user prototype supplies READY; if memory is mapped to HS, then the I²ICE system supplies READY.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default

FALSE

Discussion

The BTHRDY pseudo-variable controls the source of the READY signal used by the probe's microprocessor while emulating. The possible sources of READY are the following:

- Target system hardware (USER) memory
- High-speed (HS) memory
- Optional high-speed (OHS) memory
- MULTIBUS® (MB) memory [not supported on IBM PC hosts]
- Target system hardware (USER) I/O
- MULTIBUS I/O

If BTHRDY = TRUE and memory is mapped to HS, OHS, or MB memory, the probe's microprocessor waits for both the target system READY and READY from mapped HS, OHS, or MB memory to become valid. (When you are executing from HS or OHS memory, the probe's microprocessor matches actual target execution speed.) The target system must provide a valid READY signal. See Figure 1-5 for timing requirements when BTHRDY is TRUE. Note that the target system must meet these requirements even if memory and I/O are mapped to USER.

If BTHRDY = FALSE and memory is mapped to HS, OHS, or MB memory, target system READY is ignored for those addresses in the range of mapped memory. With this feature you can use the probe as a signal generator for debugging the target system.

Use caution when BTHRDY = FALSE and memory is not mapped to USER. The microprocessor bus cycles in the target system are not terminated by the target system READY but by the READY provided to the probe's microprocessor by the corresponding mapped memory. To prevent bus contention between the target system and the emulator when BTHRDY = FALSE, ensure that the number of wait-states requested by the target system is less than or equal to the number of wait-states specified in the WAITSTATE command.

READY TIMING FOR BTHRDY = TRUE

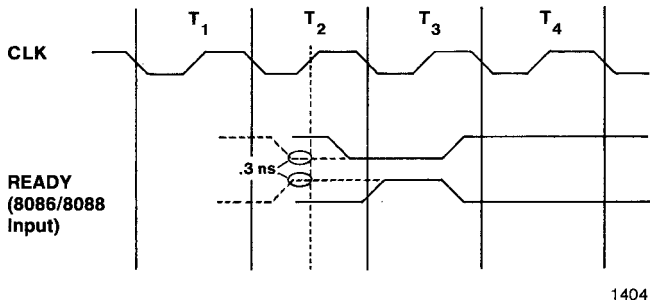


Figure 1-6 Ready Signal Set-up Time with BTHRDY Enabled

Figure 1-5 8086/8088 Probe READY Timing Requirements when BTHRDY = TRUE

BTHRDY (8086/8088) continued

The following example illustrates bus contention when BTHRDY = FALSE, the target system inserts two wait-states but WAITSTATE = 0, and memory is mapped to HS or OHS. Given these conditions, when a program is executed that causes a read cycle followed by a write cycle, the following events occur:

- I²ICE memory returns the data, terminates the cycle in zero wait-states, and starts the write cycle before the target system terminates the read cycle.
- The target system drives read data onto the data bus at the same time the probe is driving write data onto the data bus.

Examples

1. Display the current setting:

```
*BTHRDY
FALSE
```

2. Enable the prototype READY signal:

```
*BTHRDY = TRUE
```

3. Use BTHRDY as a variable:

```
*IF NOT BTHRDY THEN HALT
```

BTHRDY

80186/80188 probe specific

Pseudo-variable that controls the source of the probe microprocessor's READY

Syntax

BTHRDY { =TRUE | =FALSE | =*boolean-expression* }

Where:

BTHRDY	displays the current setting.
TRUE	uses the logical AND of the prototype READY signal and the I²ICE system READY signal to determine the number of wait-states.
FALSE	uses READY that depends on current mapping. For example, if memory is mapped to USER, the user prototype supplies READY; if memory is mapped to HS, then the I²ICE system supplies READY.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default

FALSE

Discussion

The BTHRDY pseudo-variable controls the source of the READY signal used by the probe's microprocessor while emulating. The possible sources of READY are the following:

- Target system hardware (USER) memory
- High-speed (HS) memory
- Optional high-speed (OHS) memory
- MULTIBUS (MB) memory [not supported on IBM PC hosts]
- Target system hardware (USER) I/O
- MULTIBUS I/O

BTHRDY has no effect when memory or I/O is mapped to USER; the target system must provide a valid READY signal.

If BTHRDY = TRUE and memory is mapped to HS, OHS, or MB memory, the probe's microprocessor waits for both the target system READY and READY from mapped HS, OHS, or MB memory to become valid. The target system must provide a valid READY signal.

If BTHRDY = FALSE and memory is mapped to HS, OHS, or MB memory, the target system READY is ignored for those addresses in the range of mapped memory. With this feature you can use your probe as a signal generator for debugging the target system.

Use caution when BTHRDY = FALSE. The microprocessor bus cycles in the target system are terminated by target system READY and not by the READY provided to the probe's microprocessor by the mapped memory. To prevent bus contention between the target system and the emulator when BTHRDY = FALSE, ensure that the number of wait-states requested by the target system is less than or equal to the number of wait-states specified in the WAITSTATE command. If the number of wait-states requested is greater than the number specified in the WAITSTATE command, you can still prevent contention by ensuring the following:

- The target system must not initiate bus cycles for addresses mapped to I²ICE system memory.
- The target system must not drive the data bus during a read cycle to an address mapped to the I²ICE system.

The following example illustrates bus contention when BTHRDY = FALSE, the target system inserts two wait-states but WAITSTATE = 0, and memory is mapped to HS or OHS. Given these conditions, when a program is executed which causes a read cycle followed by a write cycle, the I²ICE memory returns the data, terminates the cycle in zero wait-states, and starts the write cycle before the target system terminates the read cycle. The target system drives read data onto the data bus at the same time the probe is driving write data onto the data bus.

BTHRDY

80286 probe specific

Pseudo-variable that controls the source of the probe microprocessor's READY

Syntax

BTHRDY [= TRUE | = FALSE | = *boolean-expression*]

Where:

BTHRDY	displays the current setting.
TRUE	uses both prototype READY and I²ICE system READY to determine the number of wait-states, unless memory or I/O is mapped to USER. When memory or I/O is mapped to user, BTHRDY has no effect (i.e., the target system READY is used).
FALSE	uses READY that depends on current mapping. For example, if memory is mapped to USER, the user prototype supplies READY; if memory is mapped to HS, then the I²ICE system supplies READY.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default

TRUE

Discussion

The BTHRDY pseudo-variable controls the source of the READY signal used by the probe's microprocessor while emulating. The possible sources of READY are the following:

- Target system hardware (USER) memory
- High-speed (HS) memory
- Optional high-speed (OHS) memory
- MULTIBUS (MB) memory [not supported on IBM PC hosts]
- Target system hardware (USER) I/O
- MULTIBUS I/O

BTHRDY has no effect when memory or I/O is mapped to USER; the target system must provide a valid READY signal.

If BTHRDY = TRUE and memory is mapped to HS, OHS, or MB memory, the probe's microprocessor waits for both the target system READY and READY from mapped HS, OHS, or MB memory to become valid. The target system must provide a valid READY signal.

If BTHRDY = FALSE and memory is mapped to HS, OHS, or MB memory, target system READY is ignored for those addresses in the range of mapped memory. With this feature you can use your probe as a signal generator for debugging the target system.

Use caution when BTHRDY = FALSE. The microprocessor bus cycles in the target system are terminated by target system READY and not by the READY provided to the probe's microprocessor by the mapped memory. To prevent bus contention between the target system and the emulator when BTHRDY = FALSE, ensure that the number of wait-states requested by the target system is less than or equal to the number of wait-states specified in the WAITSTATE command. If the number of wait-states requested is greater than the number specified in the WAITSTATE command, you can still prevent contention by ensuring the following:

- The target system does not initiate bus cycles for addresses mapped to I²ICE system memory (unless the preceding WAITSTATE condition is true).
- The target system does not drive the data bus during a read cycle to an address mapped to the I²ICE system.

The following example illustrates bus contention when BTHRDY = FALSE, the target system inserts two wait-states but WAITSTATE = 0, and memory is mapped to HS or OHS. Given these conditions, when a program is executed which causes a read cycle followed by a write cycle, the I²ICE memory returns the data, terminates the cycle in zero wait-states, and starts the write cycle before the target system terminates the read cycle. The target system drives read data onto the data bus at the same time the probe is driving write data onto the data bus.

BUSACT

A pseudo-variable that allows a system time-out when the microprocessor bus is inactive for more than one second.

Syntax

BUSACT [= TRUE | = FALSE | = *boolean-expression*]

Where:

BUSACT	displays the current setting (TRUE or FALSE).
TRUE	enables bus inactive time-outs.
FALSE	disables bus inactive time-outs.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default

TRUE

Discussion

When BUSACT = TRUE, a time-out occurs when the processor bus is inactive for more than one second. A time-out causes emulation to break.

Examples

1. Display the current setting:

```
*BUSACT
TRUE
```

2. Disable the time-out:

```
*BUSACT = FALSE
```

3. Use BUSACT as a variable:

```
*DEFINE PROC busy = DO
. *IF BUSACT = = FALSE THEN
. . *BUSACT = TRUE
. . *END
. *END
```

BYTE

Displays or changes memory as an 8-bit unsigned value

Syntax

BYTE *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

BYTE *partition*	displays the location specified in *partition* as a byte value in the current base.
partition	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to an 8-bit unsigned value.
mtype	is any memory type except ASM.

Discussion

The BYTE command interprets the contents of memory as 8-bit unsigned values, overriding any type associated with the memory contents. Thus, BYTE .var1 displays the first byte at the address of var1, regardless of the type of var1.

The display includes the corresponding ASCII characters enclosed in apostrophes ('). Non-printing characters are displayed as periods (.).

Examples

The following examples assume the base is hexadecimal.

1. Display a single value:

```
*BYTE $
0020:0004H FA                                    '.'
```

2. Display several contiguous values:

```
*BYTE $ LENGTH 8T
0020:0004H FA 2E 8E 16 00 00 BC 72                '........'
```

3. Set a single value of type BYTE:

```
*BYTE 40:4 = 4AH
```

4. Set several contiguous values:

```
*BYTE 40:4 = 41H, 42H, 43H
```

Display the values set:

```
*BYTE 40:4 LENGTH 3
0040:0004H 41 42 43                                    'ABC'
```

5. Set a range of locations to the same value (block set):

```
*BYTE 40:4 LENGTH 10 = 0
```

6. Set a repeating sequence of values:

```
*BYTE 40:4 LENGTH 10 = 12H, 34H, 56H
```

Display the values set:

```
*BYTE 40:4 LENGTH 0A
0040:0004H 12 34 56 12 34 56 12 34 56 12          '.4V.4V.4V.'
```

7. Copy a value from one memory location to another:

```
*BYTE 40:4 = BYTE $
```

8. Copy several values (block move):

```
*BYTE 40:4 = BYTE $ LENGTH 10
```

9. Copy values with type conversion:

```
*BYTE 40:4 = WORD .var2
```

An error message is displayed if the type on the right side of the equal sign cannot be converted to the type on the left. (See the Expression entry in this encyclopedia for the rules concerning type conversions.)

Cross-References

Expression
Mtype
Partition

CALLSTACK

Displays the names of procedures on the stack

Syntax

CALLSTACK [*n*]

Where:

CALLSTACK	displays the names of the procedures on the stack in order of call (from top to bottom). An asterisk (*) before an element indicates the current debug cursor location.
n	is a number or expression that evaluates to the position of a procedure in the stack. If *n* is negative, the I²ICE system displays the return addresses of the earliest procedures (those on the bottom of the stack). If *n* is positive, the I²ICE system displays the return addresses of the latest procedures (those on the top of the stack). Figure 1-6 illustrates positive and negative *n*.

Discussion

With the CALLSTACK command you can view the dynamic, run-time nesting of the program as opposed to static, lexical nesting. After calls to a procedure, the stack contains the return addresses in order from earliest to most recent. Figure 1-6 illustrates the precedence of procedures in the return stack.

NOTE

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

The CALLSTACK command does not operate correctly if the last executable statement of the main module calls a procedure. The top-level return address must not be within a procedure.

The display format is as follows:

:module-name[.procedure-name] [+ offset]

If the return address is within a procedure, the procedure-name is displayed. The offset in bytes is displayed in the current number base.

CAUTION

The I²ICE system makes certain assumptions about the stack at any given time. Changing the execution point, stack segment, or stack pointer may invalidate these assumptions.

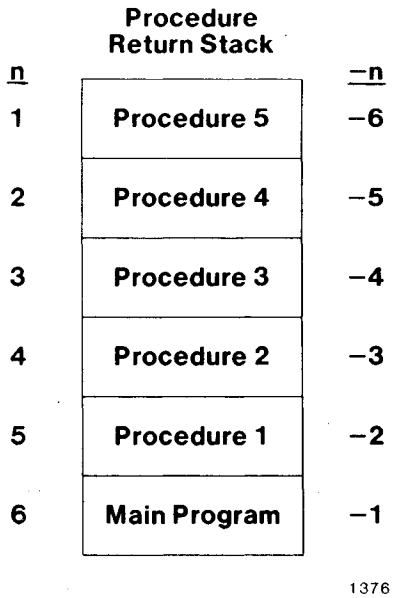


Figure 1-6 Accessing the Procedure Return Stack

Example

1. In this example, when the RETURN at the end of the current procedure is encountered, execution resumes at the address represented by (:tca.prologue + 6). The next RETURN after that returns to (:tca.main + 22).

```
*CALLSTACK
0013:0081H :tca.prologue+6
0021:003AH :tca.main+22
```

Cross-Reference

Expression

CAUSE

Displays the reason emulation stopped

Syntax

CAUSE

Discussion

With the CAUSE command you can display the reason for the last emulation halt. The CAUSE message describes the location and reason for the break. The message contains the debug register that caused the break, value of the clips, and trace buffer overflow (if applicable). The message has the following format:

PROBE *p* stopped at *address* because of *cause*
[BUS ADDRESS = *absolute address*]
[Break register is *name*] [Clips = *cc*] [Trace Buffer Overflow]

The I²ICE system fills in the *underlined* items as described in Table 1-2.

CAUSE is useful when using the ISTEP, LSTEP, PSTEP, and WAIT commands because they do not display a break message unless an error occurs or a breakpoint is stepped through. The message is the same one that the I²ICE system prints when emulation stops because of a programmed breakpoint.

Example

1. The following example displays the reason emulation stopped:

```
*CAUSE
Probe 0 stopped at :CMAKER# 10 because of guarded access
  Bus address=008274
```

Cross-Reference

Expression

Table 1-2 CAUSE Message Variables

Item	Description
p	Unit number (0-3).
address	The value of CS:IP where the unit stopped emulation, displayed in pointer or symbolic notation. (Refer to the Expression entry in this encyclopedia for notation examples.)
cause	One of the following reasons: bus break bus not active coprocessor memory violation coprocessor on bus execute break guarded access halt host I/O access I/O not ready memory not ready no user clock over temperature personality board system break write to ROM
[...]	Brackets indicate that the I²ICE system displays this information only if it is available.
absolute address	Contents of the address bus in absolute format. The number of significant bits is probe-specific.
name	When a debug register specification causes the break, the I²ICE system displays its name.
cc	A two-place hexadecimal number representing the value of the eight input logic clips.

CHAR

Displays or changes memory as ASCII characters

Syntax

CHAR *partition* [=*expression* [, *expression*]* | =*mtype partition*]

Where:

CHAR *partition*	displays the location specified in *partition* as an ASCII character value.
partition	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to an ASCII byte value.
mtype	is one of the following valid types for assignment to CHAR: BYTE, WORD, DWORD, ADDRESS, SELECTOR, CHAR, and ASM.

Discussion

The CHAR command interprets the contents of memory as 8-bit ASCII characters, overriding any type associated with the memory contents. Thus, CHAR .var1 displays the first byte at the address of var1 as an ASCII character, regardless of the type of var1. Non-printing characters and values outside the range of ASCII characters are displayed as periods (.).

Examples

1. Display a single character:

```
*CHAR $
0020:0004H 'A'
```

2. Display several adjacent characters:

```
*CHAR $ LENGTH 25T
0020:0004H 'A.....r.........|.......'
```

3. Assign a single value of type CHAR:

```
*CHAR temp = 'A'
```

4. Assign several adjacent values of type CHAR:

```
*CHAR temp = 'qwerty'
```

Display the values set:

```
*CHAR temp LENGTH 6
0040:0004H 'qwerty'
```

5. Assign several repeating values:

```
*CHAR temp LENGTH 12T = 'GR'
```

Display the values set:

```
*CHAR temp LENGTH 12T
0040:0004 'GRGRGRGRGRGR'
```

Cross-References

Expression
Mtype
Partition

CI

A function that reads
from the system terminal

Syntax

CI

Discussion

With the CI (console input) function you can read one character from the system terminal. The terminal pauses until the character is entered. No prompt is displayed while the system is waiting for the CI character, and the entered character is not echoed to the screen. No carriage return is required after the character has been keyed in.

Example

1. This example defines a procedure, a character variable, and a BRKREG. The procedure is named "query". It is called from a BRKREG named "this_round". Query displays the value of the current probe processor's registers and flags and asks if the user wants to stop emulation. A Y response returns a TRUE to the calling BRKREG and finishes the break.

```
*DEFINE PROC query = DO
. *REGS
. *WRITE USING ( '"Do you want to break?",> ')          /*Screen message */
. *DEFINE CHAR ccc = CI                                 /*Accept terminal input*/
. *WRITE ccc
. *IF ccc = = 'Y' then return true                      /*Test and return Boolean */
. . *ELSE return false
. . *ENDIF
. *END
*DEFINE BRKREG this_round = :helpentry CALL query       /* Call query at */
*GO FROM display USING this_round                       /*symbolic addr "helpentry"*/
----- REGISTERS FOR UNIT 0000 ----
AX=4            BX=63A          CX=0            DX=2
CS=5588         DS=188          SS=104          ES=0
IP=46C7         BP=634          SP=624          SI=830
DI=3A2
FLAGS : ZFL PFL
Do you want to break?Y                                  /* Return true to BRKREG */
Probe 0 stopped at :helpentry#3 because of execution break
  Break register is THIS_ROUND                          /*Probe break message */
```

CLEAREOL

Clears screen from cursor
to end of line

Syntax

CLEAREOL

Discussion

The CLEAREOL command clears the screen display from the cursor's location after the command is entered to the end of the line.

Examples

1. Clear the second line. The following CLEAREOL command clears the second line, because "CLEAREOL" is entered on the first line and then the RETURN (or Enter) key moves the cursor to the second line before CLEAREOL is executed.

```
*CURHOME                    /*Moves cursor to upper left-hand corner of screen*/

*CLEAREOL                   /*Clears the second line*/
```

2. Clear the first line. The following CLEAREOL command clears the first line, because the RETURN (or Enter) that completes the command line moves the cursor to the next line, after which the cursor is moved to the first line by CURHOME--then "CLEAREOL" is executed.

```
*CURHOME; CLEAREOL
```

CLEAREOS

Clears screen from cursor
to end of screen

Syntax

CLEAREOS

Discussion

The CLEAREOS command clears the screen display from the cursor's location after the command is entered to the end of the screen.

Examples

1. Clear the screen from the second line. The following CLEAREOS command clears the screen beginning at the second line, because "CLEAREOS" is entered on the first line and then the RETURN (or Enter) key moves the cursor to the second line before CLEAREOL is executed:

```
*CURHOME                    /*moves cursor to upper left-hand corner of screen*/
*CLEAREOS                                    /*clears entire screen*/
```

2. Clear the screen from the first line. The following CLEAREOS command clears the screen beginning at the first line, because the RETURN (or Enter) that completes the command line moves the cursor to the next line, after which the cursor is moved to the first line by CURHOME--then "CLEAREOS" is executed.

```
*CURHOME; CLEAREOS
```

CLIPSIN

Displays current state of emulator logic clips

Syntax

CLIPSIN

Discussion

The CLIPSIN command displays the current state of the eight input signals on the emulator logic clips in hexadecimal format. Each signal line on the clip pod is numbered. The number of the signal corresponds to the bit number in the byte returned. Table 1-3 lists the colors of the wires corresponding to the input signals.

Table 1-3 Input Clips Signals and Wire Colors

Signal	Wire Color	
CLIP IN 0	Brown	LSB
CLIP IN 1	Red	.
CLIP IN 2	Orange	.
CLIP IN 3	Yellow	.
CLIP IN 4	Green	.
CLIP IN 5	Blue	.
CLIP IN 6	Violet	.
CLIP IN 7	White	MSB

Example

1. Display the input clips in hexadecimal:

```
*BASE=HEX
*CLIPSIN
1FH
```

Cross-References

System specification

I²ICE timing information is provided in the I²ICE data sheet.

CLIPSOUT

Displays and sets the two output lines on the emulator logic clips

Syntax

CLIPSOUT [= *expression*]

Where:

CLIPSOUT	displays the current state of clipsout 0 and 1 on the emulator logic clips probe.
expression	evaluates to a binary 00, 01, 10, or 11. The first digit represents CLIPSOUT 1, and the second digit represents CLIPSOUT 0. A 0 equals low, and a 1 equals high. The initial value is 00.

Default

00

Discussion

The CLIPSOUT command controls two of the four output lines (clipsout 0 and 1) on the emulator logic clips probe. The other two lines, SYS BREAK/ and SYS TRACE/, are connected to the user prototype for breaking and tracing in a multiple unit configuration.

By specifying a number, the controlled lines send a TTL voltage level signal as soon as you press the RETURN (or Enter) key. The signals remain until changed with the CLIPSOUT command or until you perform a power-on reset. The I²ICE system displays the CLIPSOUT display in binary, regardless of the setting of BASE.

Example

1. Set CLIPSOUT 0 to TTL low and CLIPSOUT 1 to TTL high; then display the result.

```
*CLIPSOUT = 10Y
*CLIPSOUT
10Y
```

Cross-References

Expression
System specification

COENAB

8086/8088 probe specific

Pseudo-variable that enables or disables coprocessor functions

Syntax

```
COENAB [ =TRUE
       | =FALSE
       | =boolean-expression ]
```

Where:

COENAB	displays the current setting.
TRUE	enables the coprocessor.
FALSE	disables the coprocessor.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default Value

TRUE

Note that the default value for COENAB is FALSE if the user system's RQ/GT line is tied low (i.e., a shorted signal).

Discussion

The coprocessor enable (COENAB) pseudo-variable enables or disables an external coprocessor. When COENAB = TRUE, the 8086/8088 probe recognizes RQ/GT (MAX mode) or HOLD/HLDA (MIN mode) signals.

When using the coprocessor, you must set the COENAB command before emulation begins. Coprocessor enable remains set until you change it with the COENAB command. Resets to the probe processor and the 8087 device do not affect the setting of COENAB, but the RESET ICE command returns to the default, COENAB = TRUE.

NOTE

RESET ICE does not reset an external coprocessor; it does reset an internal coprocessor.

Although an internal coprocessor operates when COENAB is FALSE, the I²ICE system does not acknowledge trace data, register information, and coprocessor-related breaks when COENAB is FALSE. You can use the 8087 emulator software regardless of the setting of COENAB.

When you use an external coprocessor that is not enabled, any coprocessor instruction executed causes the I²ICE probe to wait indefinitely (hang) for an acknowledge. To correct the hang, manually reset the target coprocessor first and then the probe.

An internal 8087 uses RQ/GT1, leaving RQ/GT0 available for an external coprocessor. If there is no internal 8087, both RQ/GT0 and RQ/GT1 are available for 8087 coprocessors.

Examples

1. Display the current setting of the external coprocessor:

```
*COENAB
TRUE
```

2. Disable an external coprocessor circuit:

```
*COENAB = FALSE
```

Cross-Reference

Expression

COENAB
80186/80188 probe specific

Pseudo-variable that enables
or disables coprocessor functions

Syntax

COENAB [= TRUE | = FALSE | = *boolean-expression*]

Where:

COENAB	displays the current setting.
TRUE	enables the coprocessor.
FALSE	disables the coprocessor.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default Value

TRUE

Discussion

The coprocessor enable (COENAB) pseudo-variable enables or disables an external coprocessor. When COENAB = TRUE, the 80186/80188 probe recognizes its HOLD/HLDA lines.

When using a coprocessor, you must set COENAB to TRUE before emulation begins. Coprocessor enable remains set until it is changed with the COENAB command. Resets to the probe processor do not affect the setting of COENAB, but the RESET ICE command returns to the default, COENAB = TRUE.

When you use an external coprocessor that is not enabled, any coprocessor instruction executed causes the I²ICE probe to wait indefinitely (hang) for an acknowledge. To correct the hang, manually reset the target coprocessor first and then the probe.

Examples

1. Display the current setting of the external coprocessor:

```
*COENAB
TRUE
```

2. Disable an external coprocessor circuit:

```
*COENAB = FALSE
```

Cross-Reference

Expression

COENAB

80286 probe specific

Pseudo-variable that enables or
disables coprocessor functions (HOLD,HLDA)

Syntax

```
COENAB | = TRUE
       | = FALSE
       | = boolean-expression |
```

Where:

COENAB	displays the current setting.
TRUE	indicates that an external coprocessor such as the ADMA 82258 is enabled. If CPMODE is 1, the 80286 probe recognizes the HOLD and HLDA lines only during emulation. If CPMODE is 2, the 80286 probe recognizes the HOLD and HLDA lines during both emulation and interrogation.
FALSE	indicates that an external coprocessor is disabled. The 80286 probe does not recognize HOLD and HLDA lines at any time.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default Value

FALSE

Discussion

The COENAB pseudo-variable enables or disables an external coprocessor. It determines whether the 80286 probe recognizes its HOLD and HLDA lines. (The I²ICE pseudo-variable COREQ controls the PEREQ and PEACK lines.)

When you set COENAB to FALSE, any coprocessor bus request causes the coprocessor to wait indefinitely for an acknowledge (i.e., the I²ICE system may hang). If this happens, first reset the external coprocessor and then the probe (with the RESET UNIT or RESET ICE command).

Resetting the probe microprocessor (activating its RESET pin or entering the RESET UNIT command) does not change the setting of COENAB. The RESET ICE command returns COENAB to FALSE.

Examples

1. Display the current setting of the external coprocessor:

```
*COENAB
TRUE
```

2. Disable an external coprocessor:

```
*COENAB = FALSE
```

Cross-References

CPMODE
Expression

CONCAT

A function that creates and displays
a new string by concatenation

Syntax

CONCAT (*string-reference* [, *string-reference*]*)

Where:

string-reference	is characters enclosed in apostrophes, a string expression using CONCAT, NUMTOSTR, or SUBSTR functions, or a reference to a CHAR type debug variable.

Discussion

The CONCAT command builds strings by concatenating all or parts of old strings to form a new string.

The CONCAT command is used two ways: to display a new message without saving it and to display and save the new message. When the CONCAT command is entered at the prompt, it displays the new message.

Examples

1. Concatenate two strings, the predefined character string msg1 and the literal string 'PROC1':

```
*DEFINE CHAR msg1 = 'Now executing '
*CONCAT (msg1, 'PROC1')
Now executing PROC1
```

2. Concatenate two strings inside a debug variable definition.

```
*DEFINE CHAR msg2 = CONCAT (msg1, 'TEST PROCEDURE')
*msg2
Now executing TEST PROCEDURE
```

Cross-Reference

Strings

Confidence tests

A series of tests that checks I²ICE hardware

Before running the confidence tests, load the appropriate confidence test diskette and plug the user cable into the loopback socket. If you are testing an emulation clips module, connect the logic clips as shown in the *I²ICE™ System User's Guide*.

See the following EXAMPLES section for information on invoking the confidence tests.

NOTE

The 80186/80188 self-test logic (test 20) does not test the following seven pins:

VCC (two pins)
TMROUT 0 and 1 (two pins)
HOLD (one pin)
HLDA (one pin)
$\overline{\text{BHE}}$ (one pin)

NOTE

The 80286 self-test logic (test 20) does not test the following pins:

RESET	$\overline{\text{READY}}$	PEREQ	$\overline{\text{PEACK}}$	NMI
INTR	HOLD	HLDA	$\overline{\text{BUSY}}$	$\overline{\text{ERR}}$
CAP	Vcc	Vss		

Table 1-4 lists the confidence tests.

Table 1-4 The I²ICE™ System Confidence Tests

Test Number	Test Name
0000H	Interface map RAM [test ignored by IBM PC hosts]
0001H	ACK time-out
0002H	System configuration
0003H	ICE-LINK data paths
0004H	Slushware RAM
0005H	Probe initialization
0006H	Probe ID
0007H	Probe start
0008H	Probe address/data
0009H	Host/probe communications
000AH	Slushware loader
000BH	Communications exerciser
000CH	Probe CPU instruction set
000DH	Memory map RAM

Table 1-4 The I²ICE™ System Confidence Tests (continued)

Test Number	Test Name
000EH	I/O map RAM
000FH	High-speed memory map RAM
0010H	High-speed RAM
0011H	Probe memory time-out
0012H	Probe I/O time-out
0013H	Probe bus time-out
0014H	MULTIBUS DMA [test ignored by IBM PC hosts]
0015H	MULTIBUS DMA exerciser [test ignored by IBM PC hosts]
0016H	Software interrupt
0017H	High-speed memory emulation
0018H	MULTIBUS emulation [test ignored by IBM PC hosts]
0019H	Single step
001AH	8086/8088 and 80186/80188: Hardware stack pointer 80286: Hardware register dump area
001BH	Wait-state generator
001CH	Host disk mapping
001DH	Host I/O mapping
001EH	Guarded access mapping
001FH	Read-only mapping
0020H	Probe self-test
0021H	8086/8088: 8087 execution 80186/80188: Internal timer interrupt 80286: Execution state machine RAM
0022H	8086/8088 and 80186/80188: Execution state machine RAM 80286: Execution word recognizer RAM
0023H	8086/8088 and 80186/80188: Execution word recognizer RAM; 80286: Execution word recognizer decoding
0024H	8086/8088 and 80186/80188: Execution word recognizer decoding; 80286: Bus state machine
0025H	8086/8088 and 80186/80188: Bus state machine RAM 80286: Bus word recognizer RAM
0026H	8086/8088 and 80186/80188: Bus word recognizer RAM 80286: Bus word recognizer decoding
0027H	8086/8088 and 80186/80188: Bus word recognizer decoding; 80286: Execution breakpoint
0028H	8086/8088 and 80186/80188: Execution breakpoint 80286: Bus breakpoint
0029H	8086/8088 and 80186/80188: Bus breakpoint 80286: Execution bus breakpoint
002AH	8086/8088 and 80186/80188: Execution bus breakpoint 80286: Trace counter
002BH	8086/8088 and 80186/80188: Trace counter 80286: Trace on/off
002CH	8086/8088 and 80186/80188: Trace on/off 80286: Trace buffer RAM part 1
002DH	8086/8088 and 80186/80188: Trace buffer RAM part 1 80286: Trace buffer RAM part 2
002EH	8086/8088 and 80186/80188: Trace buffer RAM part 2 80286: Trace buffer RAM part 3
002FH	8086/8088 and 80186/80188: Trace buffer RAM part 3 80286: Execution delay counter

Table 1-4 The I²ICE™ System Confidence Tests (continued)

Test Number	Test Name
0030H	8086/8088 and 80186/80188: Execution delay counter 80286: Bus delay counter
0031H	8086/8088 and 80186/80188: Bus delay counter 80286: Time tag counter
0032H	8086/8088 and 80186/80188: Time-tag counter 80286: System bus
0033H	8086/8088 and 80186/80188: System bus 80286: Logic clips
0034H	8086/8088: Coprocessor word recognition 80186/80188: Status word recognition 80286: Local reset
0035H	8086/8088 and 80186/80188: Logic clips
0036H	8086/8088 and 80186/80188: Optional high speed memory
0037H	8086/8088 and 80186/80188: Verify slushware 80286: Optional high speed memory
0038H	8086/8088 and 80186/80188: User interface exerciser*
0039H	8086/8088 and 80186/80188: User emulation*
003AH	8086/8088 and 80186/80188: Host-probe utilities

* The user interface exerciser test and user emulation test assume that the target system has RAM at addresses 0 to 221H.

Examples

The following subsections provide examples for running confidence for the 8086/8088, 80186/80188, and 80286 probes.

Confidence Tests for the 8086/8088 Probe

The following examples assume that the the 8086/8088 diagnostic disk is in drive 1 (or for the IBM PC, drive A) and that you want to run the 8086/8088 confidence tests on I²ICE unit 2.

1. Run the diagnostic tests on the Intellec® Series III:

```
-RUN :F1:ICT086 2
I2ICE 086 Confidence Tests Vx.y
Copyright 1984, Intel Corporation
>TEST
```

Run the diagnostic tests on the Intellec Series IV:

```
>/164609.001/ICT086.86 2
I2ICE 086 Confidence Tests Vx.y
Copyright 1984, Intel Corporation
>TEST
```

Run the diagnostic tests on an IBM PC host. (The prompts shown in the example assume that you have set your PC prompt using the command PROMPT = PG.)

```
C:\ >a: <Enter>
A:\ >ICT086 <Enter>
I2ICE 086 Confidence Tests Vx.y
Copyright 1984, Intel Corporation
>TEST
```

2. Generate a summary of any tests that failed:

```
>SUM EO
```

3. Return to the host operating system by entering

```
>EXIT
```

Confidence Tests for the 80186/80188 Probe

The following examples assume that the 80186/80188 diagnostic diskette is in drive 1 (or drive A on an IBM PC host) and that you want to run the 80186/80188 confidence tests on unit 2.

4. Run the diagnostic tests on the Intellec Series III:

```
-RUN :F1:ICT186 2
I2ICE 186 Confidence Tests Vx.y
Copyright 1984, Intel Corporation
>TEST
```

Run the diagnostic tests on the Intellec Series IV:

```
/164609.001/ICT186.86 2
I2ICE 186 Confidence Tests Vx.y
Copyright 1984, Intel Corporation
>TEST
```

Run the diagnostic tests on an IBM PC host. (The prompts shown in the example assume that you have set the prompt using the command PROMPT = PG.)

```
C:\ >a: <Enter>
A:\ >ICT186 <Enter>
I2ICE 186 Confidence Tests Vx.y
Copyright 1984, Intel Corporation
>TEST
```

5. Generate a summary of any tests that failed:

```
>SUM EO
```

6. Return to the host operating system:

```
>EXIT
```

Confidence Tests for the 80286 Probe

The following examples assume that the 80286 diagnostic diskette is in drive 1 (or drive A on an IBM PC host) and that you want to run the 80286 confidence tests on unit 2.

7. Run the diagnostic tests on the Intellec Series III:

```
-RUN :F1:ICT286.86 2
I2ICE 286 Confidence Tests Vx.y
Copyright 1984, Intel Corporation
>TEST
```

Run the diagnostic tests on the Intellec Series IV:

```
/164609.001/ICT286.86 2
I2ICE 286 Confidence Tests Vx.y
Copyright 1984, Intel Corporation
>TEST
```

Run the diagnostic tests on an IBM PC host. (The prompts shown in the example assume that you have set your PC prompt using the command PROMPT = PG.)

```
C:\ >a: <Enter>
A:\ >ICT286 <Enter>
I2ICE 286 Confidence Tests Vx.y
Copyright 1984, Intel Corporation
>TEST
```

8. Generate a summary of any tests that failed:

```
>SUM EO
```

9. Return to the host operating system.

```
>EXIT
```

Cross-Reference

The *I²ICE™ System User's Guide* has a more information about the I²ICE confidence tests.

COREQ
80286 probe specific

Pseudo-variable that enables or disables
an external numeric extension (PEREQ, PEACK)

Syntax

```
COREQ | =TRUE
      | =FALSE
      | =boolean-expression |
```

Where:

COREQ	displays the current setting.
TRUE	indicates that an external numeric extension such as the 80287 is enabled. When CPMODE is 1, the 80286 probe recognizes the PEREQ and PEACK lines only during emulation. When CPMODE is 2, the 80286 probe recognizes the PEREQ and PEACK lines during both emulation and interrogation.
FALSE	indicates that an external numeric extension is disabled. The 80286 probe does not recognize the PEREQ and PEACK lines at any time.
boolean-expression	is any expression in which the low order bit evaluates to 0 (false) or 1 (true).

Default Value

FALSE

Discussion

The I²ICE pseudo-variable COREQ enables or disables an external numeric extension. It determines whether the 80286 probe recognizes its PEREQ and PEACK lines. (The COENAB pseudo-variable controls the HOLD and HLDA lines.)

When you set COREQ to FALSE, any processor extension data transfer request causes the processor extension to wait indefinitely for an acknowledgement. Under certain conditions when COREQ is TRUE and CPMODE is 1, the probe may also hang. When this happens, first reset the external coprocessor, and then reset the probe (with the RESET command).

Resetting the probe microprocessor (activating its RESET pin or entering the RESET UNIT command) with the RESET UNIT command does not change the setting of COREQ. The RESET ICE command returns COREQ to FALSE.

Example

1. Disable a numeric processor extension.

```
*COREQ = FALSE
```

Cross-References

80287 registers
COENAB
CPMODE

COUNT

Groups and executes commands a specified maximum number of times

Syntax

COUNT *expression*

I²ICE commands

[WHILE *boolean-condition* | UNTIL *boolean-condition*]

END[COUNT]

Where:

COUNT *expression*	specifies the maximum number of times the COUNT command loop executes. The *expression* must evaluate to a positive whole number, less than or equal to 65535T in the current base.
I²ICE commands	executes until the test condition(s) is (are) met or the terminal count is reached. All I²ICE commands are legal except HELP, LOAD, EDIT, and INCLUDE.
WHILE *boolean-condition*	executes the COUNT loop while *boolean-condition* is true. Execution halts when the WHILE condition is false or the terminal count is reached.
UNTIL *boolean-condition*	halts COUNT loop execution when *boolean-condition* is true (unless the terminal count is reached first).
END[COUNT]	terminates the COUNT block. The optional COUNT keyword labels the block type.

Discussion

Unless it is within a procedure definition, a COUNT block is executed immediately after you enter the END statement.

COUNT blocks not containing WHILE or UNTIL clauses are executed at least once. COUNT blocks containing WHILE or UNTIL exit whenever the test condition is satisfied or the count value is reached.

Example

1. The following example shows COUNT used to provide a count from 0 to 4.

```
*DEFINE BYTE b
*b=0
*COUNT 5
.*b
.*b=b+1
*END
0
1
2
3
4
*
```

Cross-References

Boolean condition
Expression

CPMODE

8086/8088 probe specific

Pseudo-variable that displays or changes
the mode of external coprocessor operation

Syntax

```
CPMODE [ =1
       =2
       =expression that evaluates to 1 or 2 ]
```

Where:

CPMODE	displays the current setting.
1	is 1 or an expression that evaluates to 1. Mode 1 allows handshaking during emulation only.
2	is 2 or an expression that evaluates to 2. Mode 2 allows handshaking during both emulation and interrogation.

Default Value

1

Discussion

Select the external coprocessor mode with the CPMODE command before emulation begins.

Mode 1

When CPMODE is 1, the 8086/8088 probe recognizes RQ/GT (MAX mode) or HOLD/HLDA (MIN mode) signals only during emulation.

Mode 1 operation assumes that emulation resumes from the last breakpoint. When this is not the case (e.g., when you use GO FROM), clear the external coprocessor of any pending requests by resetting it. You must reset the external coprocessor because the 8086/8088 probe stores a request from the 8087 coprocessor until the 8086/8088 probe enters emulation, at which time the request is honored.

Mode 2

When CPMODE is 2, the 8086/8088 probe recognizes coprocessor requests at any time. The USER memory is not protected from unauthorized access by the coprocessor. Registers are available for examination and modification.

NOTE

CPMODE operates only on the external coprocessor. It has no effect on the internal 8087 coprocessor.

Cross-References

8086/8088 registers
COENAB

CPMODE

80186/80188 probe specific

Pseudo-variable that displays or changes the mode of external coprocessor operation

Syntax

```
CPMODE | =1                                      |
       | =2                                      |
       | =expression that evaluates to 1 or 2    |
```

Where:

CPMODE	displays the current setting.
1	is 1 or an expression that evaluates to 1. Mode 1 allows handshaking during emulation only.
2	is 2 or an expression that evaluates to 2. Mode 2 allows handshaking during both emulation and interrogation.

Default Value

1

Discussion

Select external coprocessor mode with the CPMODE command before emulation begins.

NOTE

The 80186/80188 probe can have only an external coprocessor; it cannot have an internal coprocessor.

Mode 1

When CPMODE is 1, the 80186/80188 probe recognizes HOLD/HLDA signals only during emulation.

Mode 1 operation assumes that emulation resumes from the last breakpoint. When this is not the case (e.g., when you use GO FROM), clear the external coprocessor of any pending requests by resetting it.

Mode 2

When CPMODE is 2, the 80186/80188 probe recognizes coprocessor requests at any time (even while not emulating). The USER memory is not protected from unauthorized access by the coprocessor.

Cross-References

80186/80188 registers
COENAB

CPMODE

80286 probe specific

Pseudo-variable that displays or changes the mode of coprocessor (and processor extension) operation

Syntax

```
CPMODE | =1                                      |
       | =2                                      |
       | = expression that evaluates to 1 or 2   |
```

Where:

CPMODE	displays the current setting.
1	is 1 or an expression that evaluates to 1. Mode 1 indicates that the coprocessor operates only during emulation.
2	is 2 or an expression that evaluates to 2. Mode 2 indicates that the coprocessor operates during both emulation and interrogation.

Default Value

2

Discussion

When CPMODE is 1, the COENAB and COREQ pseudo-variables have meaning only during emulation. When CPMODE is 1 and COENAB is TRUE, the 80286 probe recognizes the HOLD and HLDA signals only during emulation. When CPMODE is 1 and COREQ is TRUE, the 80286 probe recognizes the the PEREQ and PEACK signals only during emulation.

Mode 1 operation assumes that emulation resumes from the last breakpoint. When emulation resumes from a different location (for example, after you use the GO FROM command), reset the coprocessor to clear any pending requests.

When CPMODE is 2, the COENAB and COREQ pseudo-variables have meaning during both emulation and interrogation. When CPMODE is 2 and COENAB is TRUE, the 80286 probe recognizes the HOLD and HLDA signals during emulation and interrogation. When

CPMODE is 2 and COREQ is TRUE, the 80286 probe recognizes the the PEREQ and PEACK signals during emulation and interrogation.

To access 80287 registers with I²ICE pseudo-variables, CPMODE must be 2 and the 80286 probe must not be emulating.

Cross-References

80287 registers
COENAB
COREQ

CURHOME

Moves cursor to the top left corner
of the display screen

Syntax

CURHOME

Discussion

The CURHOME command moves the cursor to the top left corner of the display screen (coordinates (0,0)).

CURX

A pseudo-variable that displays the column number or moves the cursor to column X

Syntax

CURX [= *expression*]

Where:

CURX	displays the number of column X in the current base.
expression	moves the CRT cursor from its current position to the indicated column. The *expression* must be in the range from 0 to the maximum number of columns on your CRT.

Discussion

The CURX command is typically used with the CURY command to position the cursor on the display screen. Any information written to the screen, after the cursor is moved, is written from the new cursor location. Any characters previously displayed at that location are deleted from the screen as the new characters are written over the old.

Example

1. This example shows cursor movement after the CURX command:

```
CURX = 40T                *
```

Cross-Reference

CURY
Expression

CURY

A pseudo-variable that displays
the row number or
moves the cursor to row Y

Syntax

CURY [= *expression*]

Where:

CURY	displays the number of the Y row in the current base.
expression	moves the CRT's cursor from its previous position to the indicated row. The *expression* must be in the range from 0 to the maximum number of rows on your CRT.

Discussion

The CURY command is usually used with the CURX command to position the cursor on the display screen. Any information written to the screen, after the cursor is moved, is written from the new cursor location. Any characters previously displayed at that location are deleted from the screen as the new characters are written over the old.

Example

1. This example shows cursor movement (from the first row to the fifth) in the Y direction.

```
CURY = 5T


*
```

Cross-Reference

CURX
Expression

Debug registers

Displays debug register contents

Syntax

debug-register name

Where:

debug-register	is one of the following keywords: ARMREG BRKREG EVTREG SYSREG TRCREG
name	is the name of a previously defined debug register.

Discussion

Debug registers contain breakpoint or trace specifications or both.

You can manipulate debug registers in the following ways:

- Create a debug register with the DEFINE command
- Delete a debug register from memory with the REMOVE command
- List debug register names with the DIR command
- Save a debug register to a file with the PUT or APPEND commands
- Retrieve a debug register from a file with the INCLUDE command
- Display a debug register by entering its keyword and name
- Execute a debug register with the GO USING command
- Modify a debug register with the editor

Example

1. This example displays the contents of the ARMREG named trigger_one.

```
*ARMREG trigger_one
DEFINE ARMREG TRIGGER_ONE=TRIG CLIPS 0XXXXXXX1Y AFTER
OCCURRENCE 5
```

Debug registers continued

Cross-Reference

Name

Debug variable

Defines, modifies, or displays a debug variable

Syntax (three forms)

1. Define a debug variable:

   DEFINE [GLOBAL] *mtype debug-variable-name* [= *expression*]

   If you do not enter *expression*, type CHAR is initially null, type BOOLEAN is initially FALSE, and all other memory types (mtypes) are initially 0.

2. Modify a debug variable:

   debug-variable-name = expression

3. Display a debug variable:

   debug-variable-name

Where:

DEFINE *mtype debug-variable-name* [= *expression*]	creates a single value of the specified memory type in host memory space.
GLOBAL	defines variables as global rather than local to any block.
mtype	can be any memory type. (See the Mtype entry in this encyclopedia for a complete list.)
debug-variable-name	displays the value of the named debug variable.
expression	can be any valid combination of values and operations.

Discussion

Debug variables can be local or global. Local variables are known only in their enclosing block and are only visible when that block is executing. Global variables can be accessed at any time.

Debug variable continued

Debug variables are global by being defined outside of a block or by being declared GLOBAL. Local variables are removed automatically after a block has been executed. Global variables are deleted with the REMOVE command.

Debug variables can be defined without a value being assigned. Values are forced to the correct type if possible.

You can change a debug variable by either reassigning its name to a new value or editing the definition.

Examples

1. Define and display a single debug variable:

```
*DEFINE BYTE b                    /*Definition of byte b, no value assigned*/
*b                                /*Command to display b*/
0
*DEFINE BYTE b = 5
*b
5
```

2. Modify and display a previously defined debug variable:

```
*b = 4T + 7T
*b
11
```

Cross-References

Expression
Mtype

Syntax (four forms)

1. To define a LITERALLY expression:

   DEFINE LITERALLY *literally-name* = *'character-string'*

2. To define a debug procedure:

   DEFINE PROC *debug-procedure-name* = DO

   .

   I²ICE commands

   .

   END

3. To define a debug register:

   DEFINE

   { ARMREG *arm-register-name* = *arm-specification*

   BRKREG *break-register-name* = *break-specification*
   [CALL *debug-procedure-name*]

   EVTREG *event-register-name* = DO *event-specification*
   [CALL *debug-procedure-name*]
   END

   SYSREG *system-register-name* = [SYSTRIG | SYSARM | SYSDARM] *system-specification*
   [CALL *debug-procedure-name*]

   TRCREG *trace-register-name* = *trace-specification* }

4. To define a debug variable:

   DEFINE [GLOBAL] *mtype debug-variable-name* [= *expression*]

Discussion

With the DEFINE command you can create LITERALLY definitions, debug procedures, debug registers, and debug variables. Defining debug objects prevents you from having to re-enter them each time you use them. The LITERALLY entry explains how to replace a character string with a specified name. The PROC entry describes defining debug procedures. The ARMREG, BRKREG, EVTREG, SYSREG, and TRCREG entries discuss defining arm, break, event, system, and trace registers, respectively. The Debug variable entry shows how to define debug variables.

Cross-References

ARMREG
Break specification
BRKREG
Debug variable
EVTREG
Expression
LITERALLY
Mtype
Name
PROC
SYSREG
System specification
TRCREG

Syntax

Display descriptors:

```
{ dtable (index)   }
{ dtable[.ALL]     }
{ DT (selector)    }
```

Alter descriptors:

```
{ dtable(index).component [ = expression] }
{ DT(selector).component [ = expression]  }
```

Where:

dtable	represents one of the three descriptor tables. The LDT is the current task's local descriptor table. The GDT is the global descriptor table. The IDT is the interrupt descriptor table.
index	is a number that identifies a descriptor within the descriptor table chosen by *dtable*. The first table entry is 0; the second is 1, etc. Note that *index* is an index and not a selector value.
ALL	specifies that all entries in the specified descriptor table are displayed.
DT	identifies the following argument as a *selector*.
selector	is a 16-bit value that identifies the descriptor table (the TI bit) and the offset into the table.
component	identifies a descriptor field. Not all components apply to every type of descriptor.
expression	resolves to a number to be loaded into the specified descriptor or descriptor field.

Discussion

Table 1-5 lists abbreviations for the 80286 descriptor types. Table 1-6 lists the mnemonics that represent the different descriptor components. Table 1-7 lists the descriptor type associated with each component.

Table 1-5 The 80286 Descriptor Types

Abbreviation	Description	Residence
CALLG	Call gate	GDT, LDT
DSEG	Data segment	GDT, LDT
DTABLE	Descriptor table	GDT
ESEG	Executable segment	GDT. LDT
INTG	Interrupt gate	IDT
TASKG	Task gate	GDT, LDT, IDT
TRAPG	Trap gate	IDT
TSS	Task state segment	GDT

Table 1-6 Mnemonics for the 80286 Descriptor Components

Mnemonic	Description	Size
BASE	Segment or table 24-bit address	3 bytes
LIMIT	Segment or table 16-bit length	1 word
WCNT	Word count for gates	5 bits
SSEL	Segment selector	1 word
SOFF	Segment offset	1 word
IR	Reserved by Intel	1 word
DPL	Descriptor privilege level	2 bits
ED	Expand down (for stack)	1 bit
W	Writable segment	1 bit
A	Accessed	1 bit
C	Conforming	1 bit
R	Readable	1 bit
P	Present	1 bit
B	Busy task	1 bit

Table 1-7 Components Associated with each Descriptor Type

Descriptor type	Component Mnemonics													
	BASE	LIMIT	WCNT	SSEL	SOFF	IR	DPL	ED	W	A	C	R	P	A
Data segment	X	X				X	X	X	X	X			X	
Executable segment	X	X				X	X			X	X	X	X	
Call gate			X	X	X	X	X						X	
Trap gate				X	X	X	X						X	
Interrupt gate				X	X	X	X						X	
Task gate				X		X	X						X	
Task state segment	X	X				X	X						X	X
Descriptor table	X	X				X	X						X	

To display a single descriptor table entry do one of the following.

1. Enter the name of the descriptor table with the entry number in parentheses.
2. Enter DT for descriptor table and put a 16-bit selector value in parentheses. The selector identifies either the LDT or the GDT.

An error message results if you specify an entry beyond the range of the descriptor table. If the entry is within range but you have identified an invalid descriptor, the entry displays in non-decoded form.

To display all the entries in a descriptor table, enter the mnemonic for the descriptor. All entries that identify present objects are displayed. If you append the optional ALL, all entries (even those identifying non-present objects) are displayed.

To set a descriptor table entry to a particular value, first identify the entry and the component you want to change. Then, set that component equal to an expression.

You can change the type of a descriptor by identifying the descriptor entry and setting it equal to one of the descriptor types in Table 1-5.

Examples

1. Display the fourth entry in the LDT:

```
*LDT(4)
LDT (3) DSEG BASE=FF0250 LIMIT=FFF9 P=1 DPL=3 ED=1 W=1 A=0 SR=0000 (SS)
```

2. Display all the descriptor entries in the LDT:

```
*LDT.ALL
LDT (1T) DSEG BASE=000140 LIMIT=00A7 P=1 DPL=0 ED=0 W=1 A=0 SR=0000
LDT (2T) DSEG BASE=000220 LIMIT=0024 P=1 DPL=3 ED=0 W=1 A=0 SR=0000 (SS)
LDT (3T) DSEG BASE=FF0250 LIMIT=FFF9 P=1 DPL=3 ED=1 W=1 A=0 SR=0000 (DS) (ES)
LDT (4T) ESEG BASE=000250 LIMIT=0014 P=1 DPL=3 C=0 R=0 A=0 SR=0000 (CS)
```

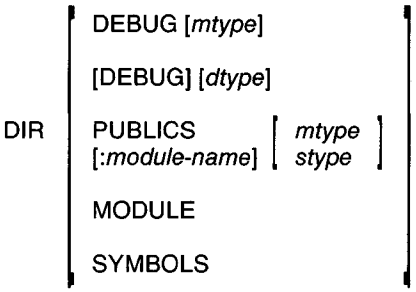
3. Set the LIMIT field of LDT(2) to 00FF:

```
*LDT(2).LIMIT = 00FF
```

DIR

Displays program symbols and debug object names

Syntax



Where:

DIR	displays the symbols for the current module as determined by NAMESCOPE.
DEBUG [*mtype*]	displays the names of all debug objects. If you specify *mtype*, only debug variables of that type are displayed.
DEBUG *dtype*	displays all the entries of the specified debug type.
mtype	is one of the memory types: BYTE, WORD, DWORD, ADDRESS, SELECTOR, SHORTINT, INTEGER, LONGINT, EXTINT, REAL, LONGREAL, TEMPREAL, BCD, POINTER, BOOLEAN, or CHAR. When any of these keywords is used as an option to the DIR command, the I²ICE system only lists the mtypes in the current module.
dtype	is one of the debug object types: PROC, LITERALLY, BRKREG, TRCREG, ARMREG, SYSREG, or EVTREG. Debug objects that are debug variables must be preceded by the DEBUG keyword to distinguish them from program variables.
PUBLICS [*mtype*] PUBLICS [*stype*]	displays symbols with the PUBLICS attribute for all modules. If *mtype* or *stype* is used, only symbols of that type are displayed. (Note that the *stype* LINE is not a valid PUBLICS type.)

[:*module-name*] [*mtype*] [:*module-name*] [*stype*]	displays the symbols for the named module. When :*module-name* is omitted, the current module is assumed. If *mtype* or *stype* is used, only symbols of that type within the module are displayed.
stype	is one of the special user program types: PROCEDURE, LINE, LABEL, FILE, ARRAY, RECORD, SET, or MODULE.
MODULE	displays the names of all modules currently loaded.
SYMBOLS	displays the names of all program symbols.

Discussion

When symbols from a module are displayed, indentation shows the scope of each symbol. The order of items displayed is undefined.

The I²ICE system recognizes I²ICE memory types and certain user program types. The I²ICE system may use different names for these types than the user program. Table 1-8 shows these differences.

Table 1-8 User Program Types with Corresponding I²ICE™ Name

ASM86	Corresponding I²ICE™ Name
BYTE	BYTE
DWORD	POINTER
QWORD	LONGREAL
STRUC	RECORD
STRUC ARRAY	ARRAY OF RECORD
TBYTE	TEMPREAL
WORD	WORD
PL/M-86	**Corresponding I²ICE™ Name**
BYTE	BYTE
DWORD	DWORD
INTEGER	INTEGER
POINTER	ADDRESS (small module)
POINTER	POINTER (large module)
REAL	REAL
SELECTOR	SELECTOR
STRUCTURE	RECORD
STRUCTURE ARRAY	ARRAY OF RECORD
WORD	WORD
Pascal-86	**Corresponding I²ICE™ Name**
ARRAY	ARRAY
BOOLEAN	BOOLEAN
CHAR	CHAR
FILE	FILE
INTEGER	INTEGER
LONGINT	LONGINT
LONGREAL	LONGREAL
REAL	REAL
RECORD	RECORD
SET	SET
TEMPREAL	TEMPREAL
WORD	WORD
FORTRAN-86	**Corresponding I²ICE™ Name**
CHARACTER*1	CHAR
INTEGER*1	SHORTINT
INTEGER*2	INTEGER
INTEGER*4	LONGINT
LOGICAL*1	BOOLEAN
LOGICAL*2	WORD
LOGICAL*4	DWORD
REAL*4	REAL
REAL*8	LONGREAL
REAL*TEMPREAL	TEMPREAL

Examples

1. The following example displays the symbols in the current module. Note that the type designations are normally aligned. Indentation indicates the nesting level of that object.

```
*DIR SYMBOLS
DIR of :PLM MODULE
MEMORY  . . . . . . . array[?] of byte
PLM_BYTE  . . . . . . byte
PLM_WORD  . . . . . . word
PLM_INTEGER . . . . . integer
PLM_REAL  . . . . . . real
PLM_DWORD . . . . . . dword
PLM_POINTER . . . . . address
PLM_BASED_BYTE  . . . byte BASED
PLM_BASED_WORD  . . . word BASED
PLM_BASED_INTEGER   . integer BASED
PLM_BASED_REAL  . . . real BASED
PLM_BASED_DWORD   . . dword BASED
ANOTHER_BYTE  . . . . byte
ANOTHER_WORD  . . . . word
ANOTHER_INTEGER . . . integer
ANOTHER_REAL  . . . . real
ANOTHER_DWORD . . . . dword
ANOTHER_POINTER . . . address
ANOTHER_BASED_BYTE  . byte BASED
ANOTHER_BASED_WORD  . word BASED
ANOTHER_BASED_INTEGER integer BASED
ANOTHER_BASED_REAL  . real BASED
ANOTHER_BASED_DWORD   dword BASED
ANY_SELECTOR  . . . . selector
PLM_BYTE_ARRAY  . . . array[10] of byte
PLM_WORD_ARRAY  . . . array[10] of word
PLM_INTEGER_ARRAY   . array[10] of integer
PLM_REAL_ARRAY  . . . array[10] or real
PLM_STRUCTURE . . . . record
  STR0_BYTE . . . . . . byte
  STR0_WORD . . . . . . word
  STR0_INTEGER . . . . integer
  STR0_REAL . . . . . . real
  STR0_BYTE_ARRAY   . . array[10] of byte
  STR0_WORD_ARRAY   . . array[10] of word
  STR0_INTEGER_ARRAY   array[10] of integer
  STR0_REAL_ARRAY   . . array[10] of real
  PLM_STRUCTURE_ARRAY   array[10] of record
```

DIR continued

```
STR1_BYTE .       byte
STR1_WORD .     . . word
STR1_INTEGER . . . . integer
STR1_REAL . . . . . . real
STR1_BYTE_ARRAY   . . array[10] of byte
STR1_WORD_ARRAY   . . array[10] of word
STR1_INTEGER_ARRAY . array[10] of integer
STR1_REAL_ARRAY   . . array[10] of real
```

2. Display the public symbols:

```
*DIR PUBLICS
DIR of PUBLICS
WCONN  . . . . . . . . . . word
RCONN  . . . . . . . . . . word
MEMORYWRITER . . . . . . . procedure
ERRCHK . . . . . . . . . . procedure
SYSTEMSTACK  . . . . . . . <null type>
```

3. Display the line numbers in the current module:

```
*DIR :check_out LINE
#1   #19  #29  #39  #40  #41  #42  #43  #44  #45
#46  #47  #48  #49  #50  #51  #52  #53  #54  #55
#59  #60  #61  #62  #63  #64  #65  #66  #67  #68
```

4. Display all debug object names:

```
*DIR DEBUG
bbb . . . byte
xxx . . . word
LIT . . . literally'literally'
WOR . . . literally'word'
BYT . . . literally'byte'
DEF . . . literally'define'
```

5. Display the directory of the module SORT, the current module:

```
*DIR :SORT
DIR of :SORT
   @10 . . . . . . . . . . . label
   @50 . . . . . . . . . . . label
   SORTARRAY . . . . . . . . ARRAYTYPE (array[100] of integer)
   CURRENTMAX . . . . . . . . INDEXTYPE (subrange of byte)
   CONTROLWORD . . . . . . . . word
  GETVALUES . . . . . . . . . procedure
     @22 . . . . . . . . . . . label
     @40 . . . . . . . . . . . label
     INDEX . . . . . . . . . . INDEXTYPE (subrange of byte) stack relative
     NEST_L . . . . . . . . . . procedure
       @21 . . . . . . . . . . label
       @30 . . . . . . . . . . label
   SORTVALUES . . . . . . . . procedure
     LEFT . . . . . . . . . . . INDEXTYPE (subrange of byte) stack relative
     RIGHT . . . . . . . . . . INDEXTYPE (subrange of byte) stack relative
     @20 . . . . . . . . . . . label
     TEMP . . . . . . . . . . . integer stack relative
     SENTINEL . . . . . . . . . integer stack relative
     J . . . . . . . . . . . . INDEXTYPE (subrange of byte) stack relative
     I . . . . . . . . . . . . INDEXTYPE (subrange of byte) stack relative
   PUTVALUES . . . . . . . . . procedure
     INDEX . . . . . . . . . . INDEXTYPE (subrange of byte) stack relative
```

Cross-Reference

Mtype

DO

Groups and executes commands

Syntax

```
DO
        [I²ICE commands]*
END
```

Where:

DO . . . END	executes one or more commands in a block.
I²ICE commands	is all I²ICE commands except LOAD, EDIT, INCLUDE, and HELP.

Discussion

The DO block is executed immediately after you enter END.

Debug variables are local only when defined in DO-END blocks. Use the GLOBAL option on the DEFINE command to define global debug objects within a DO-END block. LITERALLY definitions, debug procedures, and break and trace registers are always global.

Example

1. The following example shows how to access the values stored in an array by defining a local debug variable to serve as an index. Typically, this block would be defined in a debug procedure for reuse.

```
*DO
. *DEFINE BYTE local_var = :sort.currentmax
. *REPEAT
. . *:sort.sortarray[local_var]
. . *local_var = local_var - 1
. . *UNTIL local_var = = 0
. . *ENDREPEAT
. *END
+67
+34
+9
+8
+21
+2
+4
+7
```

DWORD

Displays or changes memory as 32-bit unsigned values

Syntax

DWORD *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

DWORD *partition*	displays the location specified in *partition* as a double word in the current base.
partition	is a single address or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to a 32-bit unsigned value for DWORD.
mtype	is any of the memory types except ASM.

Discussion

The DWORD (double word) command interprets the contents of memory as 32-bit unsigned values, overriding any type associated with the memory contents. Thus, DWORD 40:4 displays the first two words at the address of var1, regardless of the type of var1.

Examples

All the following examples assume a hexadecimal number base.

1. Display the current execution point as a double word:

```
*DWORD $
0020:0004H 168E2EFA
```

2. Display several adjacent values:

```
*DWORD $ LENGTH 5
0020:0004H 168E2EFA 16BC0000 1E8E2E00 0CEA0002 EF002100
```

3. Set a single value of type DWORD:

```
*DWORD 40:4 = 9876
```

DWORD continued

Display the value set:

```
*DWORD 40:4
0040:0004H 00009876
```

4. Set a series of adjacent values:

```
*DWORD 40:4 = 1234, 55555555, 89
```

Display the values set:

```
*DWORD 40:4 LENGTH 3
0040:0004H 00001234 55555555 00000089
```

5. Set a range of locations to the same value (block set):

```
*DWORD 40:4 LENGTH 10T = 0
```

6. Set a repeating sequence of values:

```
*DWORD 40:4 LENGTH 5 = 0A, 12345678, 4567
```

Display the values set:

```
*DWORD 40:4 LENGTH 6
0040:0014 0000000A 12345678 00004567 0000000A 12345678 90909090
```

Note that the sixth value is not affected by the command since a length of five was specified.

7. Copy a value from one memory location to another:

```
*DWORD 40:4 = DWORD $
```

8. Copy several values (block move):

```
*DWORD 40:4 = DWORD $ LENGTH 10
```

9. Copy values with type conversion:

```
*DWORD 40:4 = ADDRESS .var2
```

If the type on the right of the equal sign cannot be converted to the type on the left, an error message results. (Refer to the Expression entry in this encyclopedia for type conversion rules.)

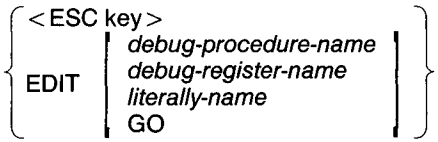
Cross-References

Expression
Mtype
Partition

EDIT

Invokes the I²ICE system editor

Syntax



Where:

<ESC key>	invokes the I²ICE screen editor if pressed while entering a command. Pressing the ESC key in response to the I²ICE prompt (*) places the last command group in the screen editor for editing.
EDIT	invokes the I²ICE screen editor and creates an empty edit buffer. You cannot invoke the EDIT command from inside a block or procedure.
debug-procedure-name	displays the definition of the named debug procedure for editing.
literally-name	displays the definition of the named literal for editing.
debug-register-name	displays the definition of the named debug register for editing.
GO	displays the GO command for editing. (The FROM clause of the GO command is not saved.)

Discussion

With the EDIT command you can create or modify previously defined debug objects. The Editors entry in this encyclopedia explains the menu-driven screen editor invoked by the EDIT command, including examples.

The I²ICE system editor has all the features of the AEDIT V1.0 editor. The *AEDIT Text Editor User's Guide* (order number 121756) describes AEDIT.

Cross-References

Editors
GO
Name

AEDIT Text Editor User's Guide (order number 121756)

The I²ICE system has a system editor and a line editor

The two editors available when you run the I²ICE software are the line editor and the menu-driven screen editor (AEDIT V1.0). The line editor (an input line processor) uses the control (CTRL) key in combination with other keys to perform editing functions. The menu-driven screen editor is invoked by the ESC key or the EDIT command.

When to Use the Editors

Use the line editor to alter commands either before pressing the carriage return or for commands in the history buffer. Use the menu-driven screen editor when creating or modifying debug objects or development system files.

NOTE

You cannot edit debug variables by typing EDIT *debug-variable-name*, but you can redefine debug variables (using the DEFINE command described in this encyclopedia).

The Line Editor

The I²ICE input line processor stores all command entries in a buffer until you press RETURN (or Enter). You can edit a command line either before pressing RETURN (or Enter) or when it is in the history buffer, thus by-passing the menu-driven screen editor.

The line editor uses the directional arrows and the CTRL key (in combination with other keys) to alter command lines.

While in line editor mode, you can press RETURN (or Enter) regardless of the position of the cursor without losing the line to the right of the cursor.

The keys that have special line editing functions are listed in Table 1-9.

Table 1-9 Line Editor Keys

Key Name	Function
RUBOUT (or ← on an IBM PC)	Deletes the character to the left of the cursor.
CTRL-A	Deletes the part of the line beginning at the cursor and continuing to the end of the line.
CTRL-C	Cancels the command in progress.
CTRL-E	Re-executes the last command.
CTRL-F	Deletes the character at the cursor and adjusts spacing.
CTRL-X	Deletes the part of the line to the left of the cursor and closes the space.
CTRL-Z	Deletes the current line.
ESC	Enters the screen editor.
Left Arrow	Moves the cursor left one character.
Right Arrow	Moves the cursor right one character.
Up Arrow	Restores the previous line from the history buffer for editing.
Down Arrow	Moves to the next line in the history buffer.
HOME	Magnifies the effect of the last arrow key. Causes jumps to the beginning or end of the current line when used with the right or left arrow.

The Screen Editor

The menu-driven screen editor has the features of the AEDIT V1.0 editor and provides functions, such as block moves, not available using the line editor. Screen editing is necessary when editing debug procedures, debug registers, LITERALLYs, or development system files (e.g., source and listing files). Note that you cannot invoke EDIT within an INCLUDE, SUBMIT, or block command.

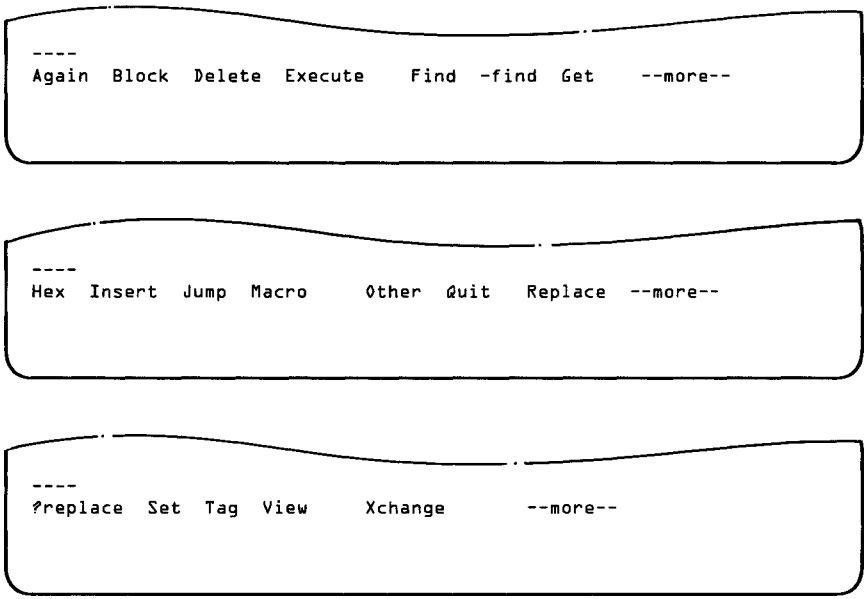
If you make an error while defining a debug object, you must press the ESC key before entering anything else. Unless immediately recalled for editing, the debug object definition is lost.

Using the ESC Key Versus EDIT Invocations

Both the ESC key and EDIT command invoke the same edit function. Use the EDIT command to create or modify previously-defined debug objects or development system files. Use the ESC key to display and modify the text of the last command sequence entered. This sequence includes all text entered since the last prompt was displayed.

Menu Contents

When invoked, the screen editor displays the edit menu at the bottom of the screen. Entering the first letter of a keyword from a menu invokes that function. The editing field at the top of the display is either blank or contains the requested command text. The following screens show the main menu prompt lines, and Table 1-10 lists each main menu item's function.



The Main Menu Screens

Table 1-10 Screen Editor Main Menu Commands and Functions

Command or Key	Function
RUBOUT	Deletes the character to the left of the cursor.
CTRL-A	Deletes that part of the line beginning at the cursor and continuing to the end of the line.
CTRL-BREAK	For IBM PC hosts, cancels the command in progress.
CTRL-C	For non-IBM PC hosts, cancels the command in progress.
CTRL-F	Deletes the character at the cursor and adjusts spacing.
CTRL-U	Restores characters deleted by the last CTRL-A, CTRL-X, or CTRL-Z to the current cursor position.
CTRL-X	Deletes the line to the left of the cursor and closes the space.
CTRL-Z	Deletes the current line.
Up Arrow	Moves the cursor up one row in the same column.
Down Arrow	Moves the cursor down one row in the same column.
Left Arrow	Moves the cursor left one character.
Right Arrow	Moves the cursor right one character.
HOME Key	Magnifies the effect of the last arrow key and causes jumps to the beginning or end of the current line when used with the right or left arrow.
Up Arrow and HOME*	Displays the previous page.
Down Arrow and HOME*	Displays the next page.
RETURN	Moves the cursor to the beginning of the next line.
ESC	Terminates the edit command in progress and returns to the main menu.
TAB	Displays the next screen of menu prompts.
A (Again)	Repeats the last command.
B (Block)	Delimits a section of text that can be deleted, copied, or moved.
D (Delete)	Delimits a section of text that can be deleted, copied, or moved.
E (Execute)	Executes the specified macro file.
F (Find)	Searches forward from the current cursor position for a specified string.
– (–find)	Searches backward from the current cursor position for the specified string.
G (Get)	Restores the contents of a block buffer or external file to the current cursor position.
H (Hex)	Converts ASCII characters to hexadecimal values and hexadecimal values to ASCII characters.
I (Insert)	Inserts text at the current cursor position.

* Pressed consecutively

Table 1-10 Screen Editor Main Menu Commands and Functions (continued)

Command or Key	Function
J (Jump)	Moves the cursor to a location specified in text by the TAG command, to the start or end of the file, or to a line or column.
M (Macro)	Creates, retrieves, and lists macro files of EDIT commands.
O (Other)	Switches between the primary and secondary buffers.
Q (Quit)	Ends the editing session.
R (Replace)	Searches for a specified string and replaces it with a new string or deletes it if found.
? (?replace)	Searches for a specified string and queries before deleting it or replacing it with a new string.
S (Set)	Sets switches that control automatic carriage return, back-up files, case significance, indents, displaying lines longer than 80 characters, tabs, displaying text when finding or replacing strings, tabs, and the view row.
T (Tag)	Specifies locations in a file to which you can jump (using the Jump command).
V (View)	Moves the cursor to the specified row.
X (Xchange)	Replaces characters on a one-for-one basis by typing over them.

Several of the screen-editor commands prompt for additional information or display sub-menus. The AEDIT manual (order number 121756) describes all the screen editor commands in detail and gives examples.

File Editing

One very useful screen-editor feature is the ability to edit development system files without exiting the I²ICE software. To edit another file, enter Q at the end the current editing session and then enter I to get the Init sub-menu. The Init sub-menu prompts for the name of the file to be edited. The file name must be a fully-qualified reference if the file resides on another drive (e.g., :F1:myfile). If you did not specify an output file before editing the external file (when the editor prompted enter [file [TO file]]), use Quit and then Write to save any changes. The Write sub-menu prompts for the name of the output file. Your changes will be lost if you do not specify an output file.

In addition to Write, the Quit command offers the eXit and Execute options. The eXit command updates the file you just edited and returns you to the I²ICE command level. The Execute command returns you to the I²ICE command level and executes the file you just edited.

Editors continued

You can also put an external file into the editor using the Get command. The Get command inserts the entire file at the current cursor position. After making your changes, delimit the text to be returned to the external file using the Block command (to retain the copy in the current file) or the Delete command (to delete the copy in the current file). Then use the Put command to return the delimited text to an external file. Note that the block buffer containing the delimited text has a fixed maximum size of 2K bytes. Use the Quit/Write commands to save larger files.

Displaying Text

The screen editor displays up to 79 characters per line. In lines exceeding 79 characters, the last (80th) character is displayed as an exclamation point (!) to indicate that text overflows the screen display width. (Use the Set command to move the left margin so that you can view characters beyond column 79.)

The editor displays tabs and unprintable characters differently from the I²ICE system. Unprintable characters are displayed as question marks (?) in the editor. The I²ICE system does not support tabs. They are displayed as single spaces.

NOTE

Editing appears on the terminal screen only. Editing sessions cannot be recorded in list files.

Cross-References

DEFINE
EDIT

The I²ICE system tutorial has modules that introduce the I²ICE screen editor and line editor.

ENABLE

Conditions the unit to accept system level breaks and traces

Syntax

```
{ENABLE  }  {SYSBREAKIN  }
{DISABLE }  {SYSTRACEIN  }
```

Where:

ENABLE	causes the system trace or system break condition on the current unit to be recognized.
DISABLE	prevents the I²ICE system from recognizing the current unit's system trace or system break condition.
SYSBREAKIN	indicates that the system break input is to be enabled or disabled. A system break is caused by SYSTRIG with the system armed.
SYSTRACEIN	indicates that the system trace input is to be enabled or disabled.

Defaults

SYSBREAKIN ENABLED
SYSTRACEIN ENABLED

Discussion

The ENABLE/DISABLE commands refer to input signals to the probe. When the current probe is enabled, it can break or trace based on input from other probes or inputs from the Intel logic timing analyzer (iLTA). You cannot configure the iLTA to break or trace on probe conditions. Refer to the *iLTA Reference Manual* (order number 163257) for the specific commands.

The system must be armed using the SYSTEM command or by the GO command using an EVTREG or SYSREG with the SYSARM option. The system break or trace conditions are activated the same way (with SYSTRIG or SYSTRACE).

You can enter any combination of enables and disables. When you enable any unit's SYSTRACEIN, that probe gathers trace data while any probe is asserting trace. When you enable any unit's SYSBREAKIN, that probe breaks when any unit asserts SYSTRIG.

ENABLE continued

To ensure that the iLTA is ready to trace emulation, specify the LAGO command before you specify the GO command (which starts probe emulation).

Cross-Reference

SYSREG

ERROR

A pseudo-variable that controls the display of error information

Syntax

```
ERROR [ =TRUE
      | =FALSE
      | =boolean-expression ]
```

Where:

ERROR	displays the current setting (TRUE or FALSE).
TRUE	tells the I²ICE system to search the disk-resident error file for the text of error messages to be displayed.
FALSE	tells the I²ICE system to display "Error Message Inhibited" and the error number. No file search occurs.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default

TRUE

Discussion

Setting ERROR to FALSE speeds up I²ICE system operation by eliminating the disk search. Because the HELP file also contains the text of error messages, you can enter the HELP command when you want expanded error information.

Examples

1. Display the current setting:

```
*ERROR
TRUE
```

ERROR continued

2. Display an error message:

```
*ERROR = TRUE
*:MOD1.BEGIN
:MOD1
ERROR #24
Cannot perform symbol table request. No user program loaded.
```

3. Suppress error messages by setting ERROR to FALSE:

```
*ERROR = FALSE
*:MOD1
ERROR #24
<Error message inhibited>
```

4. Use ERROR as a variable:

```
*IF NOT ERROR THEN
.*WRITE 'Error messages are disabled.'
.*END
Error messages are disabled.
*
```

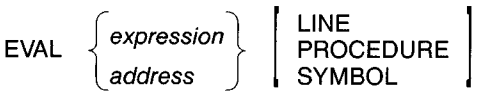
Cross-Reference

HELP

EVAL

Calculates and displays
the result of an expression

Syntax



Where:

EVAL *expression* EVAL *address*	is any valid combination of values and operations.
LINE	evaluates the expression as a line number reference.
PROCEDURE	evaluates the expression as a procedure reference.
SYMBOL	evaluates the expression as a symbolic reference (label or variable). Specify only pointer values in *address* when using SYMBOL.

Discussion

If you do not specify an option (LINE, PROCEDURE, or SYMBOL), the value of the expression is displayed. Most results are displayed in all bases (binary, decimal, and hexadecimal) and in ASCII. If the ASCII interpretation is a non-printing character, a period (.) is displayed. Results of types POINTER, unsigned values bigger than DWORD, signed values bigger than LONGINT, and strings longer than two characters are displayed as bytes in hexadecimal.

If you specify LINE, the display has the form *:module#line-number*. If the expression does not evaluate to an exact match with a line number, the system displays the line number's address that is closest to, but lower than, the value of the expression and adds + *offset*, the difference in bytes. The offset is displayed in the current base.

If you specify PROCEDURE, the display has the form *:module.procedure-name* for exact matches and adds + *offset* for inexact matches, as described for LINE.

If you specify SYMBOL, the display is a fully qualified reference to the matching user symbol, with an offset for inexact matches.

NOTE

If no symbol table information is available, the display for the LINE, PROCEDURE, or SYMBOL options gives the offset from the beginning of the current module.

If the object's address of the requested type (LINE, PROCEDURE, or SYMBOL) is less than the expression, the message <UNKNOWN> is displayed.

The SYMBOL display accesses program variables (i.e., data symbols) and labels, but not procedure names.

Examples

1. Display the result of a numeric calculation:

```
*EVAL 357T * 33H
1000111000111111Y 18207T 471FH '.G..'
```

2. Display the line number corresponding to an absolute address:

```
*EVAL 0100H:0F70H LINE
:MOD1#51
```

3. Display the location in a procedure corresponding to a line number in a module:

```
*EVAL :MOD1#05 PROCEDURE
:MOD1.SET_SCAN+24
```

4. Display a data location as a program symbol:

```
*EVAL DS:SI SYMBOL
:TCA.BIG_ARRAY+014
```

Cross-References

Address
Expression

Monitors processor events

The I²ICE system contains two state machines that work in parallel to monitor processor events: the execution event machine (XEM) monitors instruction execution and the system event machine (SEM) monitors processor bus activity (fetches, reads, and writes), address and data lines, and logic clip signal lines.

You can access these state machines in two ways. One way is to use the GO command or debug registers to set conditions for break and trace. The I²ICE system translates these conditions for appropriate event-machine testing. The second way is to manually load the event machines using an EVTREG. By programming an EVTREG, you can set up complex break and trace conditions.

Each state 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. Break or trace conditions that must match execution instructions and system action require communication between the two event machines. While either state machine is in state S3, the Boolean variable for that machine (i.e., XLINK is the Boolean variable for the XEM and SLINK is the Boolean variable for the SEM) is set to TRUE, and the XEM and SEM can communicate. Thus, decisions can be made in one machine based on the condition of the other.

Each event machine has an event counter input. Counters permit conditionally delayed triggers to be programmed. For example, the counter can be used to detect the fifth occurrence of an event, to count bus cycles, or to count the number of instructions after a trigger.

Word recognizers are the programmable portion of the internal execution state machine that compares user match specifications with conditions on the bus it monitors. When the match occurs, the state machine halts emulation. Refer to the Event machines entry in this encyclopedia for details.

The XEM state machine, shown in Figure 1-7, gathers 24 lines of execution address information from the I²ICE bus through the word recognizers. Additional information about the counter and the state of the SEM is merged with the XEM information. This merging causes the event machine to either remain in the same state or change states, then increment the counter, and halt emulation and trace collection.

The SEM, shown in Figure 1-8, monitors bus address and data, logic clips, and probe processor status through its word recognizers. Additionally, it monitors the state of the execution event machine (XLINK) and the trace buffer full-condition.

A match with any of these inputs can activate the same actions as the execution event machine. Event matching can also activate system arm, system disarm, system trigger, and system trace.

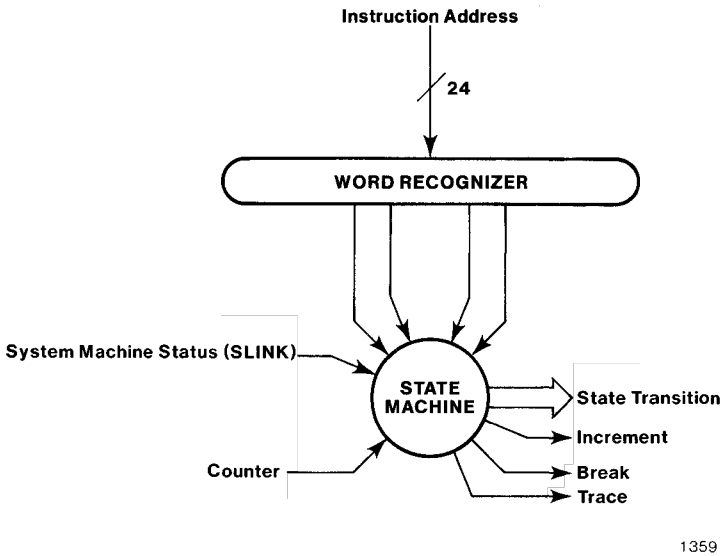


Figure 1-7 Execution Event Machine in a Sample State

Trace Buffer Full

When you start emulation (using thte GO or ISTEP commands), trace data is collected into a 1024-frame buffer. The buffer signals the event machine when it is full and about to be overwritten. You can use EVTREG or SYSREG to detect the buffer full condition. You can use buffer full detection to break emulation or switch to a state that has no trace specification (e.g., trace off).

Programming Restrictions

Normally, you can emulate with up to four break specifications in any one named break register, although this number can be lower in some cases. Where the break is a range (partition), for instance, more than the available number of word recognizers may be required to validate the match condition.

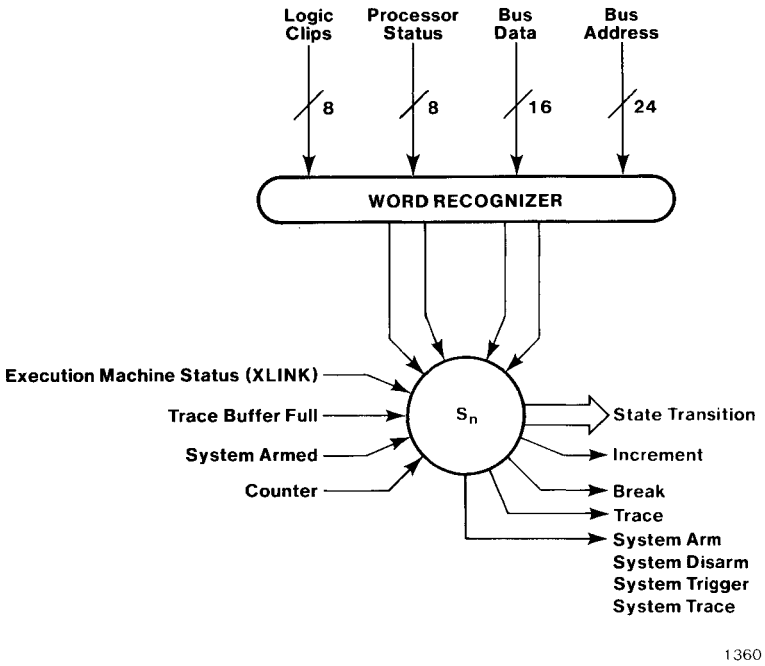


Figure 1-8 System State Machine in a Sample State

Cross-References

Event machines
EVTREG
GO

EVTREG

Defines a register that controls
the event machines

Syntax

For clarity, this command format differs from the usual format of the entries in this encyclopedia. The following skeleton syntax illustrates how the entire command appears. Detailed substitution lists follow. You should read the Event machines entry in this encyclopedia before using the DEFINE EVTREG command.

Skeleton Syntax:

```
DEFINE EVTREG name = DO
{ XEM execution-event-program-block [ SEM system-event-program-block ] }
{ SEM system-event-program-block [ XEM execution-event-program-block ] } [CALL dproc]
END
```

The skeleton example illustrates the syntax for the two event machines (the execution event machine and the system event machine) and the way they can be nested. Each event machine is controlled by a program block. The program block syntax varies, depending on which event machine you are programming.

The command entered at the terminal might look like the following:

```
DEFINE EVTREG skeleton = DO
        SEM S0 IF READ AT data THEN GOTO S1
               S1 IF WRITE AT data THEN BREAK
        END
```

The following detailed syntax diagrams describe the two machines separately. Machines are either defined individually or combined according to the format shown in the skeleton syntax.

1. Syntax to define a register to control the execution event machine:

```
DEFINE EVTREG name = DO

    XEM { state-# x-if-block  }*  [CALL dproc]
        { CTR = count         }
        { START = state-#     }

END
```

Where:

state-# is one of the following:

S0
S1
S2
S3

x-if-block is one of the following:

```
{ IF x-condition THEN x-action | ORIF x-condition THEN x-action          |*
{                              | ELSEIF x-condition THEN x-action       |
{                              | ELSE x-action                          |
{                              | BUT ALWAYS x-action                    |
{ ALWAYS x-action                                                       |
```

x-condition is

```
{ break-specification } |  WITH { break-specification } |
{ [NO] ENDCNT         }        { [NO] ENDCNT         }
{ [NO] SLINK          }        { [NO] SLINK          }
```

x-action is

```
{ GOTO state# }  |  AND { GOTO state# }  |*
{ BREAK       }        { BREAK       }
{ TRACE       }        { TRACE       }
{ INCREMENT   }        { INCREMENT   }
```

2. Syntax to define a register to control the system event machine:

```
DEFINE EVTREG name = DO

SEM  { state-# s-if-block }*  [CALL dproc]
     { CTR = count        }
     { START = state-#    }

END
```

Where:

state-# is one of the following:

S0
S1
S2
S3

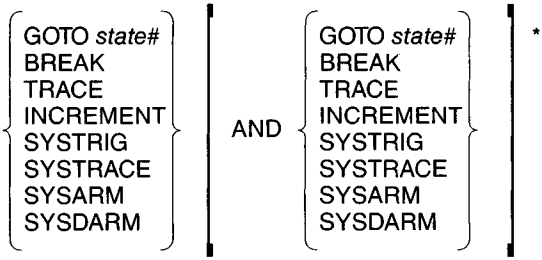
s-if-block is one of the following:

```
{ IF s-condition THEN s-action  | ORIF s-condition THEN s-action      }*
{                               | ELSEIF s-condition THEN s-action   }
{                               | ELSE s-action                      }
{                               | BUT ALWAYS s-action                }
{ ALWAYS s-action               |                                     }
```

s-condition is

```
{ system-specification }  |  [WITH] { system-specification }  |*
{ [NO] ENDCNT          }  |         { [NO] ENDCNT          }  |
{ [NO] XLINK           }  |         { [NO] XLINK           }  |
```

s-action is



You must enclose the contents of each EVTREG debug register in a DO-END block. With other debug registers, using the DO-END block is optional. Either or both of the event machines (XEM or SEM) may be programmed in any one EVTREG. You can activate only one EVTREG at a time with the GO command.

Syntax for the *execution-event-program-block* or *system-event-program-block* defines which event machine is being programmed, counter and start state initialization, and state numbers with their corresponding IF blocks.

The *conditions* the IF block tests include execution-events (addresses) and system-events (addresses, data, clips, and trace buffer full), depending on which event machine is specified. The *actions* the IF block causes when the *condition* for the XEM is matched include state changes, breaks, traces, and counter increments. The SEM adds system triggers, system arms, system disarms, and system traces.

The following paragraphs describe each keyword and variable and their legal values and defaults.

XEM — defines the execution event machine. The execution event machine recognizes break conditions for break specifications, the state of the XEM counter, and the state of the system event machine (SLINK).

SEM — defines the system event machine. It recognizes break conditions for bus data, bus addresses, logic clips, trace buffer full, processor status, the state of its counter, and the state of the execution event machine (XLINK).

state-# — is the state number and is either S0, S1, S2, or S3. S3 is the state that provides the link to the other event machine (see *x-event* and *s-event*).

EVTREG continued

CTR = *count*

sets up the event machine event counter. The count must evaluate to an unsigned integer, maximum size 64K bytes. You can omit the event machine counters and set them externally using the SCTR or XCTR commands.

START = *state-#*

indicates the state where execution is to begin. If you do not specify *start-#*, S0 is the default.

ALWAYS *actions*

causes all *actions* specified to occur (when you do not specify conditional clauses).

if-block

is either the *x-if-block* or the *s-if-block*. While the syntax for each machine looks alike at this level, each machine has different *conditions* it can recognize and *actions* it can perform. The difference in syntax accents the differences between the two event machines. The *if-block* msut be preceded by a state number.

The *if-block* is a conditional control block. When the IF condition is satisfied, the THEN action is performed. If the initial IF condition is FALSE, then the following lines are evaluated in order and executed when true. The conditions and actions that you can specify in an *if-block* vary with the event machine.

ORIF *condition*
THEN *action*

is an inclusive clause. If one or more ORIF clauses are true, including the preceding IF or ELSEIF clause, all true ORIFs produce actions. The first GOTO specified takes precedence in case of contention.

ELSEIF *condition*
THEN *action*

is an exclusive clause. IF more an one ELSEIF clause is true, incuding the IF clause, only the first true conditional clause, including any immediately following TRUE ORIF clauses, produces actions.

ELSE *action*

is evaluated when none of the other care TRUE.

BUT ALWAYS *action*

causes all actions specified in that state to occur unconditionally.

x-condition

can be stated singly or ANDed together using the optional WITH keyword. They are break specifications that include a single address, a list of addresses or partitions, the state of the event counter in the execution event machine, or the state of the system event machine (SLINK).

The SLINK execution condition **is true** when the system event machine (SEM) is in state 3. With this option the SEM can arm the execution event machine (XEM).

The [NO]ENDCNT execution condition tests whether the associated event machine counter is equal to the counter value set (CTR, XCTR, or SCTR).

s-condition

can be stated singly or ANDed together using the optional WITH keyword. An *s-condition* includes any *system-specification*, such as bus data, bus address, clips and buffer full, the state of the event counter in the system event machine, or the state of the execution event machine (XLINK).

The *system-specification* syntax, because of its length and because other debug registers share the same format, is detailed in the System specification entry in this encyclopedia.

The XLINK system condition is true when the execution event machine (XEM) is in state 3. This option lets the XEM arm the system event machine (SEM).

The [NO]ENDCNT system condition tests whether the associated event machine counter is equal to the counter value set (CTR, XCTR, or SCTR).

x-action

can be listed singly or ANDed together using the AND keyword. An *x-action* is the result of an event in the execution event machine being recognized as true.

GOTO state-#	transfers control to a new state.
BREAK	causes the probe to break emulation.
TRACE	causes the emulator to trace while the associated conditional clause is true.
INCREMENT	adds one to the counter, in the current base.

s-action — can be listed singly or ANDed together using the AND keyword. An *s-action* is the result of an event in the system event machine being recognized as true.

GOTO state-#	transfers control to a new state.
BREAK	causes the probe to halt emulation.
TRACE	causes the emulator to trace while the associated conditional clause is true.
INCREMENT	adds one to the counter, in the current base.
SYSTRIG	causes a system trigger to be sent to all enabled units.
SYSTRACE	causes conditional trace collection as a result of any enabled unit's trigger.
SYSARM	causes a system arm to be sent to all enabled units.
SYSDARM	causes a system disarm to be sent to all enabled units.

The ANDed lists require parentheses.

CALL *dproc* — calls the debug procedure named when a GO USING *evtreg-name* causes an emulation break.

Discussion

Event machine control is an automatic process in the high-level break and trace control commands (e.g., GO USING *brkreg-name*). By using event registers, you can control the event machines directly. Regardless of how they are specified, all breaks and traces occur through the event machine hardware.

When to Use EVTREGs

Consider using event registers in the following situations:

- When a GO command exceeds the number of break specifications the system can handle. The I²ICE system reports an error when this happens.
- When the complexity of the statement exceeds the capabilities of other debug registers.
- When you need both the counting features of an ARMREG and multiple arm and disarm features of a SYSREG in one statement.

Specifying EVTREGs

The body of the syntax of the *event-program-block* (pictured in the skeleton diagram) is essentially an IF-THEN-ELSE control structure. It adds state numbers, similar to line numbers or labels, to transfer control from state to state.

Because the number of word recognizers is limited, you can specify only a finite number of break criteria. Notice that omitting the *condition* preceding the optional keyword WITH in the IF block statement lets you use additional word recognizers.

For example:

```
*DEFINE EVTREG wr = DO
**XEM   S0   IF address THEN INCREMENT
**           ORIF WITH ENDCNT THEN BREAK
**END
```

This example is missing the *condition* between the ORIF and WITH keywords. It is syntactically legal. The I²ICE system inserts the *address* specification following the IF. This form is also legal in the GO command. (See the GO entry for details.)

Manipulating EVTREGs

Manipulate an EVTREG by referring to its name. You can manipulate EVTREGs in the following ways:

- Create an EVTREG with the DEFINE command
- Delete an EVTREG from memory with the REMOVE command
- List EVTREG names with the DIR command
- Save an EVTREG on file with the PUT or APPEND commands
- Retrieve an EVTREG from a file with the INCLUDE command

- Display an EVTREG with the EVTREG command
- Execute an EVTREG with the GO USING command
- Modify an EVTREG with the editor

NOTE

Defining new break and trace specifications using an old EVTREG name destroys the old definition in memory. An error occurs if you try to assign an EVTREG name to any other debug object in memory.

Restoring a saved EVTREG that has the same name as an EVTREG in memory overwrites the latter.

An error results when you try to restore a saved EVTREG that has the same name as any other debug object in memory.

Using the Optional Call

When emulation halts because an EVTREG included a CALL, the CALL transfers control to the named debug procedure. This debug procedure must return a Boolean value (TRUE or FALSE) to the EVTREG. If TRUE is returned, emulation stops. If FALSE is returned, emulation continues.

NOTE

Emulation halts if a Boolean value is not returned or there is an error in the called debug procedure. An error message indicates that the halt was not caused by a normal execution break.

Examples

1. The following example illustrates how the same specification can be made using one EVTREG rather than two other debug registers. Both versions catch an event when execution takes place in .proc__a and a data value (0123H) is written to .adr__a.

```
*DEFINE SYSREG catchit = DO
**WRITE AT .adr__a IS 123H END
*DEFINE BRKREG temp = proc__a
*GO USING BOTH (temp) AND (catchit)
```

   The same specification using EVTREG is as follows:

```
*DEFINE EVTREG catchit =
**DO
**XEM S0 IF .proc__a THEN GOTO S3
**S3 ALWAYS GOTO S3
**SEM S0 IF WRITE AT .adr__a IS 0123H WITH XLINK
**THEN BREAK
**END
*GO USING catchit
```

2. The following example illustrates an event machine program that causes a break at line 68. Furthermore, a break only occurs if line 68 is executed after lines 32, 44, and 56 are executed (in order) and line 125 is not executed.

```
*DEFINE EVTREG only__on__tuesdays = DO
**XEM SO IF #32 THEN GOTO S1
**S1 IF #125 THEN GOTO S0
**ELSEIF #44 THEN GOTO S2
**S2 IF #125 THEN GOTO S0
**ELSEIF #56 THEN GOTO S3
**S3 IF #125 THEN GOTO S0
**ELSEIF #68 THEN BREAK
**END
*GO USING only__on__tuesdays
```

Cross-References

Break specification
Event machines
GO
System specification

EXIT

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

Syntax

```
EXIT
```

Discussion

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

You cannot use the EXIT command in two cases:

- If any probe has any memory mapped to the MULTIBUS (MB) memory. (To exit in this case, reset MAP by entering RESET MAP before entering EXIT). [Note that MULTIBUS memory mapping is not available on IBM PC hosts.]
- If any of I/O memory is mapped to I²ICE while any probe is emulating. (To exit in this case, reset MAPIO by entering RESET MAPIO before entering EXIT.)

Expression

One or more numbers, variables, pseudo-variables, or functions separated by operators

Syntax

*[unary-op] operand [binary-op [unary-op] operand]**

Where:

unary-op	(unary operator) acts on a single operand (Table 1-14 defines unary operators).
operand	can be a constant, a variable, a pseudo-variable, a function, or a sub-expression. Some operands are user-defined; others are system-defined.
binary-op	(binary operator) acts on two operands. The result is a single operand (Table 1-15 defines binary operators).

Discussion

An expression is a combination of operands and operators. Evaluating an expression applies the operators to the operands until a single result is obtained. This section explains how to display the result of an expression, tells how expressions are evaluated, and describes the operands and operators that are valid in I²ICE system expressions.

Evaluating Expressions

An expression entered as a command is evaluated directly. The result is displayed in the current base. For example, assuming the default base is DECIMAL:

```
*357 * 51
18207
```

You can also use the EVAL command and the WRITE command to display the result of an expression. However, the examples in this section use direct evaluation.

You can use the contents of a programming location read as an mtype in an expression (mtypes are described in the Mtype entry in this encyclopedia).

Expression continued

To evaluate an expression, the system scans the expression iteratively from left to right, one iteration for each operator in the expression. The series of scans ends when either of two conditions occurs:

- Nothing remains except a single numeric result
- A syntax error, type combination error, or other error occurs

On each iteration, the scan identifies the operator that must be applied next. This operator can be unary (requires one operand) or binary (requires two operands). The next operator is always the left-most operator with the highest precedence that is enclosed in the inner-most pair of parentheses. (Precedence rules are discussed later in this section.)

If the next operator is unary, its operand must be adjacent to it and of a proper type. If so, the operator is applied to produce a numeric result. If not, an error results. The operation may change the type of the operand.

If the next operator is binary, its two operands must be of proper types. The operation then produces a numeric result. If not, an error results. The operation may change the types of the operands. (Refer to the Mtype entry in this encyclopedia for the rules of type combination and conversion.)

An error occurs if the next operator does not have the required number of operands. Spaces are allowed between operators and operands.

A pair of parentheses is unnecessary when it contains a single result. For example, (7) is the same as 7.

After an operation is performed, the numeric result becomes an operand for the next scan. Parentheses are cleared before the next scan begins.

Operands

The following sections summarize the classes of operands that the system accepts. The four classes of operands are constants, variables, functions, and sub-expressions. Within each class, some operands are user-defined and others are built-in (that is, the form is defined by the I²ICE system). An expression can be a single operand without any operators.

Constants

Constants do not change value during execution. Table 1-11 summarizes the user-defined and built-in constants. Subsequent sections give additional information on constants.

Table 1-11 Constants

Constant	Description
USER-DEFINED CONSTANTS	
unsigned integer constants	Interpreted in current base, stored as a double word (DWORD).
signed integer constants	Interpreted in current base, stored as a long integer (LONGINT).
real number constants	Always decimal, stored as a temporary real number (TEMPREAL).
string constants	ASCII characters (maximum 254), enclosed in delimiters ('). You can use one-character strings as operands with arithmetic operators.
BUILT-IN CONSTANTS	
TRUE	Boolean value TRUE.
FALSE	Boolean value FALSE.
FLDPI	pi, type TEMPREAL; value 3.14159265358979324E + 00000.
FLDL2T	$\log_2(10)$, type TEMPREAL; value 3.32192809488736235E + 00000.
FLDL2E	$\log_2(e)$, type TEMPREAL; value 1.44269504088896341E + 00000.
FLDLG2	$\log_{10}(2)$, type TEMPREAL; value 3.01029995663981195E – 00001.
FLDLN2	$\log_e(2)$, type TEMPREAL; value 6.93147180559945309E – 00001.

Unsigned-Integer Constants

An unsigned integer contains one or more valid digits and (optionally) a character indicating the number base. If you omit the number base character, the digits are interpreted in the current number base. The valid digits and characters for binary, decimal, and hexadecimal number bases are as follows:

Base	Valid Digits	Number Base Character
BINARY	0, 1	Y
DECIMAL	0 through 9	T
HEX	0 through 9, A through F	H

NOTE

To avoid confusion with variables and symbols, a hexadecimal number must not have a letter as the first digit. For this reason, a hexadecimal number requires a leading 0; for example, use 0AB6H instead of AB6H.

Integers of the form *n*K are valid constants, where *n* is an unsigned decimal integer, and K is 1024.

Examples of unsigned integers (decimal base) are as follows:

```
*1
1

*10110111Y
183

*15T
15

*0F7AH
3962

*64K
65536
```

Unsigned integers belong to the unsigned class of basic program types.

Signed-Integer Constants

A signed integer includes unary plus or minus. For example:

```
*+10
+10

*-157T
-157
```

Signed integers of 8, 16, and 32 bits belong to the signed class of program types. Signed integers with 64 bits use an 8087 coprocessor or 8087 emulator and belong to the 8087 class of program types.

Real-Number Constants

Real numbers have the following general format:

[sign][numeral][.numeral]*[E[sign][numeral*]]*

Real numbers are always decimal. The sign, plus or minus, is optionally included. The numerals are the decimal numerals 0 through 9. The decimal point can be anywhere in the sequence of numerals. The following two examples show how to enter a real number at the terminal and shows the I²ICE system's evaluation of that number.

```
*0.1
1.0000000000000000E-1

*12345.6789
1.2345678900000000E+4
```

The exponent (E) form is also called scientific notation for real numbers. No space is permitted before the E. For example:

```
*43.337E4
4.3337000000000000E+5

*-1.0445E-5
1.0445000000000000E-5
```

NOTE

When you use the E-form, a decimal point is required to distinguish them from hexadecimal integers of the form nnnnEnnnn. Numerals are required both on the left and on the right of the decimal point.

All real numbers are stored as TEMPREALs (10 bytes) and must be in the range of TEMPREALs. Real numbers require the 8087 coprocessor and belong to the 8087 class of program types.

Strings

A string contains up to 254 characters enclosed in apostrophes (′). An apostrophe within a string is entered as a double apostrophe (′′). The value of a string is its ASCII representation, with a byte for each corresponding character. You can use one-character strings as operands for arithmetic operators. For example:

```
*'abcdef'
 abcdef

*'c%'
 c%

*'a'+5                          /* A one-character string used as an operand */
 102
```

You can also use string functions such as CONCAT and SUBSTR in expressions. String functions are included in Table 1-13.

Built-in Constants

Built-in constants are of two types, BOOLEAN and TEMPREAL.

The BOOLEAN constants are true (representing the value 1) and false (representing 0). Because only the least significant bit of a value is used in a BOOLEAN type context, these constants provide the expected Boolean logic. The BOOLEAN constants are useful for setting up variables in the I²ICE system. For example:

```
*MEMRDY = TRUE

*BUSACT = FALSE
```

The TEMPREAL constants (FLDPI, FLD2T, FLD2E, FLDLG2, FLDLN2) correspond to 8087 constant instructions. For example:

```
*DEFINE REAL radius = 2.445E4
*DEFINE REAL circumference = FLDPI * radius * 2.0
*circumference
 1.53624E+5
```

Variables

Variables store values that can change during execution or by user command. The name of the variable represents the current value. Table 1-12 summarizes the user-defined variables recognized by the I²ICE system.

Table 1-12 User-Defined Variables

Variable	Definition
procedure reference	Returns the address of the first executable (machine) instruction in the procedure.
line number reference	Returns the address of the first executable instruction in the line.
label reference	Returns the address of the first executable instruction in the labeled statement.
program variable	Returns the contents of the data variable.
debug variable	Returns the contents of the debug variable.

User-defined variables include symbolic references to program addresses and variables and debug variables defined during the debug session.

Symbolic References

Symbolic references to program addresses include procedure names, line numbers, and labels and represent the address of the first executable instruction within the procedure, line, or labeled statement, respectively. For example:

```
*:mod1.parser                                  /* Procedure reference */
0100:0F30H

*:mod2#523                                     /* Line number reference */
0000:0100H

*:mod3.parser.start                            /* Label reference */
0F72:001AH
```

The name of a variable in the user program represents the current value (contents) of the variable. For example:

```
*:mod1.parser.character_count                  /* Simple variable */
15

*table[50]                                     /* Array variable */
123

*data_record.item                              /* Field in a structure or record */
12
```

Debug Variables

Debug variables are defined by the user within the debug session to hold temporary values. To refer to a debug variable in an expression, enter the name of the variable.

The following example shows a command block in which most of the numbers are to be in binary. The block saves the current base by defining a debug variable TEMPRADIX, switches to binary radix for the commands, then restores the previous base by naming TEMPRADIX in the assignment command. (Note that the variable TEMPRADIX is local to the block and is removed automatically after the block finishes executing.)

```
*DO
. *DEFINE BYTE tempradix = BASE
. *BASE = 2T
. *                                   /* Commands using binary numbers */
. *BASE = tempradix
. *END
```

Functions

You call a function by naming the function and specifying any required parameters. The function returns a value to the place in the expression or command from which it was called. Table 1-13 summarizes the available functions.

User-defined functions are debug procedures that include the RETURN command. An error occurs if the debug procedure does not have a RETURN command when it is used as a function. The following is an example of a debug procedure that uses RETURN.

```
*DEFINE PROC in = DO
 . *IF CI = = 'y' THEN RETURN TRUE
 . *ELSE RETURN FALSE
 . *END
*END
*
*in                                   /* User enters N */
FALSE
```

The built-in functions are the mathematical, general-purpose, and string functions and are summarized in Table 1-13. For example:

```
*ACTIVE(get_char)
FALSE
*FSQRT(10)
3.1622776601683793
*SUBSTR('abcdef', 3, 2)
cd
```

Table 1-13 Functions

Function	Description
USER-DEFINED FUNCTIONS	
debug procedure call	A debug procedure must have a RETURN statement in its definition to be used as a function. The call then returns the expression specified in the RETURN command.
BUILT-IN FUNCTIONS	
Mathematical Functions	
FPTAN (*x*)	Partial tangent (*x* is converted to TEMPREAL).
FPATAN (*x*,*y*)	Partial arctangent (*x* and *y* are converted to TEMPREAL).
FSQRT (*x*)	Square root (*x* is converted to TEMPREAL).
F2XM1 (*x*)	$2^x - 1$ (*x* is converted to TEMPREAL).
FYL2X (*x*, *y*)	$y * \log_2(x)$ (*x* and *y* are converted to TEMPREAL).
FYL2XP1 (*x*, *y*)	$y * \log_2(x+1)$ (*x* and *y* are converted to TEMPREAL).
General-purpose Functions	
ACTIVE (*symbolic-reference*)	Returns TRUE if the symbolic reference is active at the current execution point (i.e., is a static object or a dynamic object with space allocated to it); otherwise, returns FALSE.
CI	Enables you to enter a single character from the terminal and returns that character as the operand value.
OFFSETOF (*pointer*)	Returns the offset portion of the pointer.
SELECTOROF (*pointer*)	Returns the selector (segment) portion of the pointer.
PTR (*partition*, *mtype*, *unit*)	Returns a pointer to the partition of the type and unit specified.
String Functions	
string-reference	The reference can be characters enclosed in apostrophes, a string expression using CONCAT or SUBSTR, or a reference to a CHAR type debug variable.
STRLEN (*string-reference*)	Returns the number of characters in the string.
CONCAT (*string-reference* [, *string-reference*]*)	Creates a new string by concatenating the strings referenced.

Table 1-13 Functions (continued)

Function	Description
SUBSTR (*string-reference*, *start*, *length*)	Returns the substring of (maximum) length *length* starting at the character indexed by *start* (string indexes begin at 1).
STRTONUM (*string-reference*)	Returns the numeric value of the string, based on the ASCII code. The type of the result depends on the context.
NUMTOSTR (*expression*)	Converts the expression into its ASCII representation.
INSTR (*stringref1*, *stringref2* [, *start*])	Searches for *stringref2* within *stringref1* and returns the index of the first character of *stringref2*. The optional *start* defines where to begin the search in *stringref1*.

Sub-expressions

A sub-expression is an expression enclosed in parentheses. Parentheses override the precedence of the operators. An expression inside parentheses is evaluated first, thus becoming an operand for the rest of the expression outside the parentheses. When parentheses are nested, the sub-expression in the inner-most pair of parentheses is evaluated first. For example:

```
*10/(5-3)
5
```

Operators

Expressions can use a variety of operators. Unary operators act on a single operand; binary operators combine two operands.

Unary Operators

Table 1-14 summarizes the unary operators; the following paragraphs provide details.

Table 1-14 Definitions of Unary Operators

Operator	Operation
"	A double quotation mark must precede symbolic references (forcing look-up of the reference in the user symbol table) when the symbol name duplicates a keyword or debug variable name.
.	The dot operator returns the address (type POINTER) of a symbolic reference to a user program variable. Without the dot operator, a reference to a program variable returns the memory contents of the variable.
+	Unary plus denotes a positive number.
–	Unary minus denotes a negative number and converts an unsigned value to a 2's complement signed value.
NOT	NOT is the 1's complement.

Double-Quote Operator

You must use the double-quote operator (") when a user program symbol duplicates an I²ICE keyword. (See the Keywords entry in this encyclopedia for a list of I²ICE keywords.) The double-quote operator forces the system to use the symbol defined in your program for the reference. The following example command causes an error because exit is an I²ICE system keyword.

```
*:mod1.exit
```

But use of the double-quote operator makes possible the desired reference. For example:

```
*:mod1."exit
```

Dot Operator

The dot operator placed before a symbolic reference to a program variable, returns the address of the variable as a POINTER value. For example, if your program has a variable named COUNTER of type BYTE, then the following command returns the BYTE content of the variable.

```
*counter
12
```

But the variable preceded by the dot operator returns the memory address of the variable. For example:

```
*.counter
0FE8:0014H
```

Unary Plus and Minus

The unary plus (+) causes the operand to be treated as a signed integer rather than unsigned number. For example:

1 + 255 = 256

+1 + (+255) = +256

The unary minus (−) reverses the sign of its operand. If the operand is an unsigned type, unary minus converts the operand to a signed integer, using the 2's complement. Examples:

```
*-a
-546378923

*-1.0
-1.0000000000000000

*-5*10
-50
```

NOT Operator

The NOT operator returns the 1's complement of its operand. For example:

```
*NOT ACTIVE (var2)
TRUE
```

The operand must be unsigned or Boolean; NOT is invalid with any other mtype.

Binary Operators

The sign determines the value with which the PICE system calculates in binary arithmetic. A binary operator working on two signed constants does signed arithmetic. A binary operator working on two unsigned constants does unsigned arithmetic. Table 1-15 summarizes the binary operators. The Mtype entry in this encyclopedia lists the rules for type conversions. The following examples illustrate binary operators:

```
*+10-(-15)
25
*10-15.0
-5.0000
```

Table 1-15 Definitions of Binary (Two-Operand) Operators

Operator	Function
Pointer	
:	Creates a pointer (selector:offset) from two operands
Arithmetic	
*	Multiplication
/	Division
MOD	Modulo reduction (remainder after division)
+	Addition
–	Subtraction
Relational	
= =	Equal to
>	Greater than
<	Less than
> =	Greater than or equal to
< =	Less than or equal to
< >	Not equal to
Logical	
AND	Bit-wise AND
OR	Bit-wise inclusive OR
XOR	Bit-wise exclusive OR

Table 1-16 shows the relative precedence of the unary and binary operators.

Table 1-16 The I²ICE™ System Operators in Order of Precedence

Precedence*	Operators
1	., "
2	:
3	unary +, –
4	*, /, MOD
5	binary +, –
6	NOT
7	AND
8	OR, XOR

* 1 = highest precedence (evaluated first), 8 = lowest precedence.

Pointer Operator

The pointer operator (:) creates a pointer out of two 16-bit values. A pointer has the following format:

selector:offset

where *selector* and *offset* can be any type except BOOLEAN. For example:

```
*0000H:1234H
0000:1234H
*CS:IP
FFFF:0000H
```

Arithmetic Operators

The binary arithmetic operations in order of precedence are multiplication, division, modulo reduction, addition, and subtraction.

The multiplication operator (*) returns the product of its operands. For example:

```
*index = 2
*index * 16T
32
*radius = 2.0
*2.0 * FLDPI * radius
1.25663706143591730E+1
```

The division operator (/) returns the quotient of its two operands. If both operands are integers (signed or unsigned) the result is the integer part of the quotient (i.e., integer division for integer operands). If either operand is real, the result is also real.

For example:

```
* +5/2
+2
*5.0/2.0
2.5
*FLDPI/2
1.57079632679489662
```

The MOD operator returns the remainder after division of its two operands. With integer operands, MOD returns an integer; with real operands, MOD returns the fractional quotient. For example:

```
*10 MOD 3
1
*FLDPI MOD 3
1.41592653589793239E-1
```

The addition operator (+) returns the sum of its two operators; the subtraction operator (−) returns the difference. For example:

```
*$-5
0014:FFFFH
*real_var = 12.34
*real_var + 2.34E2
2.4634000015258791E+2
```

Note that addition and subtraction have lower precedence than multiplication and division. For example:

```
*2 + 3 * 4
14
```

Relational Operators

The relational operators (included in Table 1-15) compare two operands. If the comparison holds, the operation returns TRUE (1). If not, it returns FALSE (0). For example:

```
*$ > :mod1.last_action
FALSE
```

Logical Operators

The logical operators are AND, OR, and XOR (exclusive OR). Operands other than Boolean and unsigned types are invalid with logical operators.

Logical operations performed on relational expressions evaluate to a Boolean value. For example:

```
*($ = = :mod1.entry) AND (count = = 5)
TRUE
```

Expression continued

Logical operations performed on unsigned expressions return bit-by-bit values. For example:

```
*BASE = 2T
*DEFINE x = 1000
*DEFINE y = 1111
*x AND y
1000
```

Examples

The following examples assume a decimal base.

1. Arithmetic expression:

```
*357 + 28
385
```

2. Using real constants and variables:

```
*RADIUS = 2.445E4
*2.0 * FLDPI * RADIUS
1.536238807605408 89E+5
```

3. Logical operator with a function call:

```
*NOT ACTIVE (myproc.var1)
TRUE
```

Cross-References

EVAL
Keywords
Masked constant
Mtype
Strings
WRITE

EXTINT

Displays or changes memory as 64-bit signed values

Syntax

EXTINT *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

EXTINT *partition*	displays the location specified in *partition* as an extended integer in decimal.
partition	is a single address or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to a 64-bit signed value for EXTINT.
mtype	is any of the memory types except ASM.

Discussion

The EXTINT command interprets the contents of memory as 64-bit signed values, overriding any type associated with the memory contents. Thus, entering EXTINT .var1 displays the extended integer that begins at the address of var1, regardless of the type of var1. If the most significant nibble of the unsigned data comprising the EXTINT is 8 through F, it is interpreted as a negative number and displayed in the 2's complement form of the unsigned data.

Note that the I²ICE system always displays values for signed-integer memory types as decimal numbers, regardless of the selected number base.

Examples

In the following examples, the number base is hexadecimal and $ refers to the current execution point.

1. Display a single value:

```
*EXTINT $
0020:0006H          +1673000016443179
```

2. Display consecutive values:

```
*EXTINT $ LENGTH 3
0020:0006H +16730000164431?9 +03350002178129?0
0020:0016H -13742276109934?0
```

3. Set a single value of type EXTINT:

```
*EXTINT 40:04 = +123456789ABCDEF
```

4. Set adjacent values:

```
*EXTINT 40:04 = +7, +123456789ABCDEF, +12AB
```

Display the values set (you can set memory locations to signed integer values using a hexadecimal base, but the I²ICE system displays the values in decimal):

```
*EXTINT 40:04 LENGTH 3
0040:0004H      +7        +81985529216486895
0040:0014H      +4779
```

5. Set a range of locations to a single value:

```
*EXTINT 40:04 LENGTH 10 = 0
```

6. Set a repeating sequence of values:

```
*EXTINT 40:04 LENGTH 10 = +7, +123456789ABCDEF, +12AB
```

7. Copy a value from one memory location to another:

```
*EXTINT 40:04 = EXTINT $
```

The destination is the memory location left of the equal sign; the source is on the right.

8. Copy several values (block move):

```
*EXTINT 40:04 = EXTINT $ LENGTH 10
```

9. Copy values with type conversion:

```
*EXTINT 40:04 = INTEGER .var2
```

An error messages is displayed if the type on the right side of the equal sign cannot be converted to the type on the left. (Refer to the Expression entry in this encyclopedia for the rules concerning type conversions.)

Cross-References

Expression
Mtype
Partition

F2XM1

$2^X - 1$ function

Syntax

F2XM1 (*x*)

Where:

F2XM1 (*x*)	represents the function $2^x - 1$.
x	is a number or expression that evaluates to a number ($0 < x < 0.05$).

Discussion

The F2XM1 function is identical to the 8087 instruction.

The parameter (*x*) is converted to type TEMPREAL, and the result is TEMPREAL.

You can use the F2XM1 function anywhere an expression is valid.

NOTE

If *x* is outside the range $0 \leq x \leq 0.5$, F2XM1 produces an undefined result without signaling an exception.

Example

1. Calculate the $2^x - 1$ function for x = .25:

```
*F2XM1 (0.25)
1.8920711500272106?E-1
```

Cross-Reference

Expression

Displays or modifies 8086/8088 flags

Syntax

{ FLAGS | FL | FH | *8086/8088-flag* } [= *expression*]

Where:

FLAGS	displays the 8086/8088 flags register.
expression	is an expression (of the correct data type) used to set flag values.
FL	displays the lower (least significant) byte of the 8086/8088 flags register.
FH	displays the upper (most significant) byte of the 8086/8088 flags register.
8086/8088-flag	displays the current value of a flag and is one of the keywords shown in Figure 1-9.

Discussion

Display flag values by entering their keywords or by entering the keyword FLAGS. Flag values are displayed as Boolean values. The REGS command displays the flag mnemonic of all flags set to 1. If no flags are set, the word "none" is displayed.

You can modify individual flags in two ways. One way is to enter the word FLAGS (or FL or FH) ORed or ANDed with the proper bit pattern. The other way is to assign a value to the individual flag. The flag is set according to the value of the least significant bit (LSB) in *expression*.

The FLAGS Register

Bit	Keyword	Description	I²ICE™ System Memory Type
15		Don't care	
12			
11	OFL	Overflow flag	BOOLEAN
10	DFL	Direction flag	BOOLEAN
9	IFL	Interrupt flag	BOOLEAN
8	TFL	Trap flag	BOOLEAN
7	SFL	Sign flag	BOOLEAN
6	ZFL	Zero flag	BOOLEAN
5	XX	Don't care	
4	AFL	Auxiliary flag	BOOLEAN
3	XX	Don't care	
2	PFL	Parity flag	BOOLEAN
1	XX	Don't care	
0	CFL	Carry flag	BOOLEAN

1597

Figure 1-9 8086/8088 Flags Register Bit Pattern

Example

1. Display the value of the zero flag and set the trap flag:

```
*ZFL
TRUE
*FLAGS = FLAGS OR 100000000Y                    /* Set the trap flag */
*TFL
TRUE
```

Cross-Reference

Expression

Displays or modifies 80186/80188 flags

Syntax

```
{ FLAGS
  FL
  FH
  80186/80188-flag } [ = expression]
```

Where:

FLAGS	displays the current value of the 80186/80188 flags register.
expression	is an expression (of the correct data type) used to set flag values.
FL	displays the lower (least significant) byte of the 80186/80188 flags register.
FH	displays the upper (most significant) byte of the 80186/80188 flags register.
80186/80188-flag	displays the current value of a flag and is one of the keywords shown in Figure 1-10.

Discussion

Display flag values by entering their individual keywords or by entering the keyword FLAGS. Flag values are displayed as Boolean values. The REGS command displays the flag mnemonic of all flags set to 1; if no flags are set, the word "none" is displayed.

You can modify individual flags in two ways. One way is to enter the word FLAGS ORed or ANDed with the proper bit pattern (only the least significant 16 bits of *expression* are used). The other way is to assign a Boolean value to the individual flag.

Example

1. The following example shows two ways to set the trap flag.

```
*FLAGS = FLAGS OR 100000000Y                    /*Set the trap flag */
*TFL = TRUE
```

The FLAGS Register

Bit	Keyword	Description	I²ICE™ System Memory Type
15		Don't care	
12			
11	OFL	Overflow flag	BOOLEAN
10	DFL	Direction flag	BOOLEAN
9	IFL	Interrupt flag	BOOLEAN
8	TFL	Trap flag	BOOLEAN
7	SFL	Sign flag	BOOLEAN
6	ZFL	Zero flag	BOOLEAN
5	XX	Don't care	
4	AFL	Auxiliary flag	BOOLEAN
3	XX	Don't care	
2	PFL	Parity flag	BOOLEAN
1	XX	Don't care	
0	CFL	Carry flag	BOOLEAN

1598

Figure 1-10 80186/80188 Flags Register Bit Pattern

Cross-Reference

Expression

Syntax

{ FLAGS | FL | FH | *80286-flag* } [= *expression*]

{ MSW | *80286-flag* } [= *expression*]

Where:

FLAGS	displays the current value of the 80286 flags register (see Figure 1-11).
MSW	displays the current value of the 80286 machine status word (MSW) register (see Figure 1-12).
FL	displays the lower (least significant) byte of the 80286 flags register.
FH	displays the upper (most significant) byte of the 80286 flags register.
80286-flag	displays the current value of a flag and is one of the keywords shown in Figures 1-11 and 1-12. The flags may belong to the 80286 flags register or to the 80286 machine status word (MSW). Figure 1-11 shows the bit pattern of the FLAGS register. Figure 1-12 shows the bit pattern of the MSW.
expression	is an expression (of the correct data type) used to set flag values.

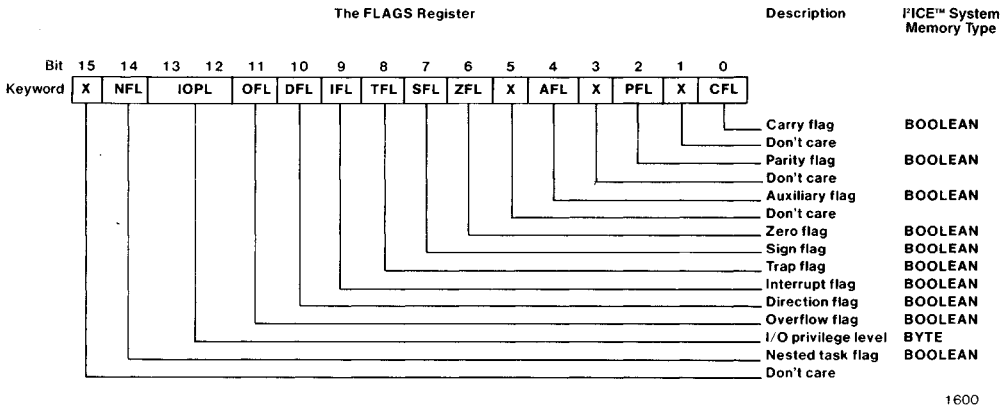


Figure 1-11 80286 Flags Register Bit Pattern

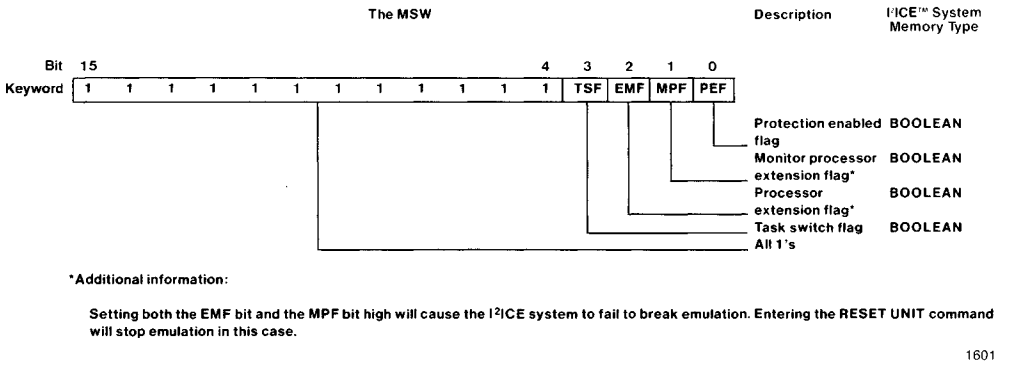


Figure 1-12 The MSW Bit Pattern

Discussion

Display flag values by entering their individual keywords or by entering the keyword FLAGS. Flag values are displayed as Boolean values. The REGS command displays the flag mnemonic of all flags set to 1; if no flags are set, the word "none" is displayed.

You can modify individual flags in two ways. One way is to enter the word FLAGS ORed or ANDed with the proper bit pattern (only the least significant 16 bits of *expression* are used). The other way is to assign a Boolean value to the individual flag.

A task switch always sets the task-switch flag (TSF). A task switch performed with a CALL or an INT instruction also sets the nested task flag (NFL). A task switch performed with a JMP instruction leaves the NFL unchanged.

Examples

1. Set the trap flag (TFL) to TRUE.

```
*FH = FH OR 1Y
*TFL
TRUE
```

2. Another way to set the trap flag to TRUE:

```
*TFL = TRUE
*TFL
TRUE
```

Cross-References

Expression
Multitasking
80286 registers
REGS

FLDL2E

Constant $\log_2(e)$

Syntax

FLDL2E

Where:

FLDL2E	is the constant $\log_2(e)$, type TEMPREAL, value 1.4426950408896341. The value (e) is the base of the natural logarithm.

Discussion

The constant FLDL2E is identical to the 8087 constant instruction. You can use it anywhere an expression is valid.

Example

```
*FLDL2E
1.4426950408896341
*FLDL2E * 3.7
5.337971651289161461
```

Cross-Reference

Expression

Syntax

FLDL2T

Where:

FLDL2T	is the constant $\log_2(10)$, type TEMPREAL, value 3.32192809488736235.

Discussion

The constant FLDL2T is identical to the 8087 constant instruction. You can use it anywhere an expression is valid.

Example

```
*FLDL2T
3.32192809488736235
```

Cross-Reference

Expression

FLDLG2

Constant $\log_{10}(2)$

Syntax

FLDLG2

Where:

FLDLG2	is the constant $\log_{10}(2)$, type TEMPREAL, value 5601029995663981195E−1.

Discussion

The constant FLDLG2 is identical to the 8087 constant instruction. You can use it anywhere an expression is valid.

Example

```
*FLDLG2
3.01029995663981195E-1
```

Cross-Reference

Expression

Syntax

FLDLN2

Where:

FLDLN2	is the constant $\log_e(2)$, type TEMPREAL, value $6.9314718055994530 9E-1$.

Discussion

The constant FLDLN2 is identical to the 8087 constant instruction. You can use it anywhere an expression is valid.

Example

```
*FLDLN2
6.93147180559945309E-1
```

Cross-Reference

Expression

FLDPI

Mathematical constant pi

Syntax

FLDPI

Where:

FLDPI	is the constant pi, type TEMPREAL, value 3.14159265358979324.

Discussion

The constant FLDPI is identical to the 8087 constant instruction. You can use it anywhere an expression is valid.

Examples

1. The following debug procedure calculates the circumference of a circle. The radius of the circle is passed to the procedure as a parameter (indicated by %0), and the value of the circumference is passed back with a RETURN command.

```
*DEFINE PROC circum = DO
. *RETURN 2.0 * FLDPI * %0
. *END
```

2. The following example shows a call to the previous debug procedure using the radius 24,450T.

```
*circum (2.445E4)
1.536238807605408B9E+5
```

Cross-Reference

Expression

Partial arctangent function

Syntax

FPATAN (*x*,*y*)

Where:

FPATAN (*x*,*y*)	is the partial arctan (y/x) function (angle in radians) (0 $< y < x < \infty$).

Discussion

The FPATAN function is identical to the 8087 partial arctangent instruction. The result is type TEMPREAL.

The value *x* is converted to TEMPREAL before the function is applied.

You can use the FPATAN function anywhere an Expression is valid.

NOTE

If *x* or *y* is outside the range $0 < y < x < \infty$, FPATAN produces an undefined result without signaling an exception.

Example

```
*FPATAN (2,5)
4.63647609000806116E-1
```

Cross-Reference

Expression

FPTAN

Partial tangent function

Syntax

FPTAN (*x*)

Where:

FPTAN (*x*)	is the partial tangent function of an angle. The argument x is the angle in radians and $(0 \leq x \leq \pi/4)$.

Discussion

The FPATAN function is identical to the 8087 partial tangent instruction.

The argument x is converted to type TEMPREAL before the function is applied. The result is type TEMPREAL.

You can use FPTAN anywhere an expression is valid.

NOTE

If x is outside the range $0 \leq x \leq \pi/4$, FPTAN produces an undefined result without signaling an exception.

Example

1. Calculate and display the partial tangent of .5T:

```
*FPTAN (0.5)
5.463024898437905132E-1
```

Cross-Reference

Expression

FSQRT

Square root function

Syntax

FSQRT (*x*)

Where:

FSQRT	is the square root function.
x	is a number or expression that evaluates to a number ($0 \leq x \leq +\infty$).

Discussion

The square root function is identical to the 8087 square root instruction.

The parameter *x* is converted to type TEMPREAL, and the result is TEMPREAL.

You can use the function FSQRT anywhere an expression is valid.

Example

1. Calculate and display the square root of 10T:

```
*FSQRT (10T)
3.162277660168379933
```

Cross-Reference

Expression

FYL2X

Y * $\log_2(x)$ function

Syntax

FYL2X (*x*, *y*)

Where:

FYL2X	is the function $y * \log_2(x)$.
x	is a number or expression that evaluates to a number ($0 < x < +\infty$).
y	is a number or expression that evaluates to a number ($-\infty < y < +\infty$).

Discussion

The FYL2X function is identical to the 8087 log instruction.

The parameters *x* and *y* are converted to TEMPREAL before the function is applied, and the result is type TEMPREAL.

You can use FYL2X anywhere an expression is valid.

Examples

1. Calculate and display the FYL2X function of 4T and 1T:

```
*FYL2X (4,1)
2.0000000000000000
```

2. Calculate and display the FYL2X function of 38.7T, 23.34T:

```
*FYL2X (38.7, 23.34)
1.2310126717373949 99E+2
```

Cross-Reference

Expression

Syntax

FYL2XP1 (*x*, *y*)

Where:

FYL2XP1	is the function $y * \log_2(x + 1)$.
x	is a number or expression that evaluates to a number ($0 < \|x\| < (1-(\sqrt{2})/2)$.
y	is a number or expression that evaluates to a number ($-\infty < y < +\infty$).

Discussion

The FYL2XP1 function is identical to the 8087 log instruction.

The parameters *x* and *y* are converted to TEMPREAL before the function is applied and the result is type TEMPREAL.

You can use FYL2XP1 anywhere an expression is valid.

Example

1. Calculate and display the FYL2XP1 function for .1T and 1T.

```
*FYL2XP1 (0.1, 1)
1.37503523749934908E-1
```

Cross-Reference

Expression

GET87
8086/8088 probe specific

Defines register handling conditions
for the 8087 coprocessor

Syntax

GET87 [(*address*)]

Where:

GET87	tells the I²ICE system that an 8087 coprocessor is present.
address	describes the starting address of a 110-byte buffer area in user-mapped memory; *address* must be 10H or greater. The I²ICE system uses this area to save and restore the register contents of the external 8087 coprocessor when entering and exiting emulation. Use *address* with the external coprocessor only.

Discussion

Enter the GET87 command before starting emulation to tell the I²ICE system that an 8087 coprocessor is present. When emulation breaks, the I²ICE system preserves the values of the 8087 registers by saving them in memory. This makes all 8087 register data available to you for display and modification. You cannot display or modify the 8087 registers if you have not entered the GET87 command. When emulation resumes, the registers are restored to the 8087 coprocessor. The GET87 command does not affect the operation of either internal or external 8087's during emulation.

Note that you need enter the GET87 command only once before emulation begins. However, if the specified 110-byte buffer changes status and no longer starts at 10H or greater, enter the GET87 command again to set up the new buffer area.

The I²ICE system determines the memory area in which to save registers by first determining the location of the user 8087 coprocessor. External 8087's save registers in user memory, while internal 8087's save registers in reserved system memory. The I²ICE system looks for the 8087 coprocessor and responds in one of three ways:

- If an external 8087 is present, the I²ICE system uses it.
- If no internal 8087 is present, the I²ICE system assumes an external 8087.
- If no external 8087 is present, the I²ICE probe hangs.

When using an internal 8087 coprocessor, user memory is not altered and the optional *address* is not required; *address* input is ignored.

When using an external 8087, you must identify a 110-byte buffer in user mapped memory using the *address* option. This area is used 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.

Example

1. Set up the I²ICE system to recognize an external 8087 coprocessor:

```
*MAP CS:100 LENGTH 100 USER
*GET87 (100H)
*GO USING math_brk
/* BREAK MESSAGE */
*ST0
+4.02932000900000000E-12
```

Cross-Reference

Address

GET87
80186/80188 probe specific

Defines register handling conditions
for the 8087 coprocessor

Syntax

GET87 (*address*)

Where:

GET87	tells the I²ICE system that an 8087 coprocessor is present.
address	describes the starting address of a 110-byte buffer area in user-mapped read/write memory; *address* must be greater than 10H. The I²ICE system uses this area to save and restore the register contents of the external 8087 coprocessor upon entering and exiting emulation. After restoring 8087 coprocessor register contents, the I²ICE system restores user memory to its previous contents.

Discussion

Enter the GET87 command before emulation begins to tell the I²ICE system that an 8087 coprocessor is present. When emulation breaks, the I²ICE system preserves the values of the 8087 registers by saving them in memory. This makes all 8087 register data available to you for display and modification. You cannot display or modify the 8087 registers if you have not entered the GET87 command. When emulation resumes, the registers are restored to the 8087 coprocessor. The GET87 command does not affect the operation of the external 8087 coprocessor during emulation.

Note that you need enter the GET87 command only once before emulation begins. However, if the specified 110-byte buffer changes status and the starting address is no longer greater than 10H, enter the GET87 command again to set up a new buffer area.

The I²ICE system responds in one of two ways to a GET87 command.

- If an external 8087 is present and the GET87 command is entered, the I²ICE system uses the external 8087.
- If no external 8087 is present and the GET87 command is entered, the I²ICE system hangs.

Example

1. Set up the I²ICE system to recognize an external 8087 coprocessor:

```
*MAP CS:100 LENGTH 100 USER
*GET87 (100H)
*GO USING math_brk
/* BREAK MESSAGE */
*ST0
+4.02932000900000000E-12
```

Cross-Reference

Address

GO

Starts emulation and controls
break and trace functions

Syntax

```
GO [FROM address] [ go-til-spec    ] [go-trace-spec]
                  [ go-using-spec  ]
```

Where *go-til-spec* is one of the following:

```
{ FOREVER                                                    }
{ TIL  | break-spec                                         | }
{      | SYSARM system-spec                                 | }
{      | SYSDARM system-spec                                | }
{      | SYSTRIG system-spec                                | }
{      | arm-spec                                           | }
{      | [DO] evt-spec END   /*see note*/                   | }
{                                                            }
{ TIL BOTH (break-spec) AND (system-spec)                    }
```

Where go-using-spec is one of the following:

```
{ FOREVER                                                    }
{ USING | evtreg-name   /*see note*/                        | }
{       | BRKREG                                            | }
{       | SYSREG                                            | }
{       | armreg-name                                       | }
{       | brkregs                                           | }
{       | sysregs                                           | }
{                                                            }
{ USING BOTH (brkregs) AND (sysregs)                         }
```

Where *go-trace-spec* is one of the following:

```
TRACE { trcreg-name [,trcreg-name]* }
      { trace-spec                  }
```

trace-spec is

```
[ break-spec          ]
[ [SYSTRACE] system-spec ]
```

NOTE

You cannot use the TRACE option with *evt-spec*. You can use the TRACE option with *evtreg-name*.

Where:

GO — starts emulation from the current execution point (CS:IP) without altering break or trace specifications. The I²ICE system's initial condition is to GO FOREVER and to always collect trace.

FROM *address* — changes the current execution point to the *address* specified.

CAUTION

Changing the execution point can invalidate the stack.

If the FROM location is not the first byte of a machine language instruction, the I²ICE system may execute an incorrect data item as an opcode.

FOREVER — clears all active break specifications and starts emulation. GO FOREVER is the initial condition.

TIL — specifies break or trace conditions (or both) without registers. The I²ICE probe remembers these conditions until you either execute a GO FOREVER, specify a new break or trace condition, or issue a RESET BREAK.

break-spec — is a numeric or symbolic address (line number, module name, label, or a list of addresses). *break-spec* syntax appears under the Break specification entry in this encyclopedia.

system-spec — is a bus address, bus data, logic clip information, the buffer full condition, or probe processor status. *system-spec* syntax appears under the System specification entry in this encyclopedia.

arm-spec is replaced by the following syntax:

```
{ ARM cond [DISARM cond] TRIG t-cond
{ [ARM cond] TRIG t-cond | AFTER { INSTRUCTION count
                                 { OCCURRENCE count } | }
```

The ARMREG entry in this encyclopedia contains details on *arm-spec*.

evt-spec is replaced by the following syntax:

```
{ XEM { state-# x-if-block
      { CTR = count
      { START = state-# }
{ SEM { state-# s-if-block
      { CTR = count
      { START = state-# } }
```

The EVTREG entry in this encyclopedia contains details on *evt-spec*.

TIL BOTH (*break-spec*) AND (*system-spec*) combines a logically ORed list of execution addresses with a logically ORed list of system specifications. The combination is logically ANDed. *break-spec* must precede *system-spec*.

USING specifies break or trace conditions or both using previously-defined break and trace registers. The I²ICE probe remembers these conditions until you either execute a GO FOREVER, specify a new break or trace condition, or issue a RESET BREAK.

evtreg-name causes the I²ICE probe to break or trace (or both) based on conditions specified in the named event register.

BRKREG causes the I²ICE probe to respond to all BRKREGs currently defined in memory.

SYSREG causes the I²ICE probe to respond to all SYSREGs currently defined in memory.

armreg-name — causes the I²ICE system to break based on conditions specified in the named arm register.

brkregs — causes the I²ICE probe to break based on conditions specified in one or more named break registers. The form is as follows:

brkreg-name [,*brkreg-name*]*

sysregs — causes the I²ICE probe to break based on conditions specified in one or more named system registers. The form is as follows:

sysreg-name [,*sysreg-name*]*

GO USING BOTH (*brkregs*) AND (*sysregs*) — combines a logically ORed list of *brkregs* with a logically ORed list of *sysregs*. The combination is logically ANDed. The *brkregs* must precede the *sysregs*. For simplicity and accuracy, use only one BRKREG and SYSREG.

TRACE — informs the I²ICE system that a trace specification follows.

trcreg-name — causes the I²ICE system to collect traces based on conditions specified in the named trace register. You can specify the contents of trace registers directly in the GO command. (The TRCREG command specifies syntax.)

SYSTRACE — specifies that when the *system-spec* is met, any I²ICE units properly enabled are triggered and traced according to the defined criteria. Do not specify SYSTRACE on any unit which also has SYSARM, SYSDARM, or SYSTRIG specified.

Discussion

This section explains when to choose one form of the GO command over another. The choice is based on the type of condition specified, the type of action the I²ICE system is to take, and whether the condition specification is to be reused.

The types of conditions specified to the I²ICE system are break specifications and system specifications. The types of actions the I²ICE system is to take are breaking emulation, halting trace collection, triggering, arming, and disarming. The Execution with Breakpoints section describes these features.

Use a debug register to save a condition specification for a later debug session. Debug registers are saved and recalled by name. The Execution with Debug Registers section describes this feature.

Note that you can edit the most recent GO command by entering EDIT GO.

Using the GO Syntax

Figure 1-13 illustrates the branches of the GO command syntax. If you specify GO without options, the last *go-til-spec* or *go-using-spec* is used.

While a probe is emulating, you cannot issue a command that requires a probe, such as GO or ISTEP. During emulation, the I²ICE system replaces the asterisk prompt (*) with a question mark prompt (?) to remind you that the current probe is emulating.

When using 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

Execution without Breakpoints

To start emulation without setting breakpoints, use the FOREVER option, which executes your program through its normal end. The I²ICE system automatically collects trace information unless otherwise directed. Stop emulation before the end of the program by entering CTRL-C. For IBM PC hosts, use CTRL-Break instead of CTRL-C.

Execution with Breakpoints

To conditionally break emulation, use the GO TIL or the GO USING option. Emulation also stops when aborted by CTRL-C (or by CTRL-Break, for IBM PC hosts) or when an error occurs.

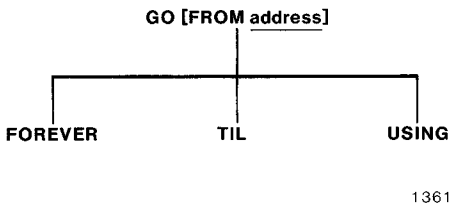


Figure 1-13 Branches of the GO Command Syntax

With the TIL construct you can specify break conditions directly on the GO command line. Use the TIL construct for simple break conditions that require few keystrokes and which will be used only once.

The USING construct requires defined debug registers. Use the USING construct when the break condition exceeds one line or when the break condition will be used more than once.

NOTE

When using the 8086/8088 probe, you must not specify an execution break on an instruction that accesses memory locations 08H to 0BH.

Execution with Debug Registers

Consider using debug registers when breakpoints and trace conditions are complex and will be reused. By putting conditions into debug registers you can identify specifications by name so that changing conditions is simplified, in addition to saving re-entry time.

The keywords for the five debug register types are BRKREG, SYSREG, ARMREG, EVTREG, and TRCREG. Each register type permits only certain kinds of conditions. Refer to each debug register type's keyword in this encyclopedia for details.

Examples

1. The following example shows a simple GO command. The current I²ICE probe starts execution from the current contents of CS:IP (e.g., the last breakpoint).

```
*GO
```

2. The following example specifies a starting address in the GO command.

```
*GO FROM 12:0 FOREVER
```

3. The following example shows two ways to specify a break at location 12:26 and begin trace collection from locations 12:8 to 12:18.

```
*GO TIL 12:26 TRACE 12:8 TO 12:18
```

or

```
*DEFINE TRCREG trace_it = 12:8 TO 12:18
*GO TIL 12:26 TRACE trace_it
```

4. The following example shows two ways to specify a break at locations 2 or 4.

```
*DEFINE EVTREG count = DO
**XEM S0 IF 2 THEN BREAK
**ORIF 4 THEN BREAK END
*GO USING count
```

or

```
*GO TIL DO XEM S0 IF 2 THEN BREAK ORIF 4 THEN BREAK END
```

5. The following example shows two ways to specify a combined BRKREG and SYSREG specification. Both versions cause a break where execution takes place in .proc__b, and a data value (0123H) is written to .adr__b.

```
*DEFINE BRKREG temp = .proc__b
*DEFINE SYSREG sysbreak = WRITE AT .adr__b IS 123H
*GO USING BOTH (temp) and (sysbreak)
```

or

```
*GO TIL BOTH (.proc__b) AND (WRITE AT .adr__b IS 123H)
```

Cross-References

ARMREG
Break specification
BRKREG
Debug registers
DEFINE
EVTREG
Name
SYSREG
System specification
TRCREG

Determines the block size used for memory mapping

Syntax

GRANULARITY [= 1K | = 16K]

Default Value

1K

Discussion

You can map 1024 blocks of memory in either 1K-byte blocks or 16K-byte blocks. When GRANULARITY = 1K, only the lower megabyte of memory is mappable. When GRANULARITY = 16K, the entire address space is available. Before you can change the GRANULARITY, program memory must be mapped to all USER or all GUARDED.

The I²ICE system always maps to OHS in 16K-byte blocks, even if GRANULARITY is 1K.

GRANULARITY Is 1K

The memory map behaves differently for memory mapped to HS, MB, OHS, or GUARDED and for memory mapped to USER.

Memory Mapped to HS, MB, OHS, or GUARDED

When you map to HS, MB, or GUARDED and the granularity is 1K, the 80286 probe ignores the upper four address bits (<A23-A20>). Consequently, an address wrap-around occurs, and each physical memory location has 16 physical addresses. For example, if you map the 1K-byte block from physical address 0K to physical address 1K – 1 to HS, you are actually mapping the following 16 blocks to the same RAM:

0 to 1K – 1
1M to 1M + 1K – 1
2M to 2M + 1K – 1
3M to 3M + 1K – 1
4M to 4M + 1K – 1

5M to 5M + 1K – 1
6M to 6M + 1K – 1
7M to 7M + 1K – 1
8M to 8M + 1K – 1
9M to 9M + 1K – 1
10M to 10M + 1K – 1
11M to 11M + 1K – 1
12M to 12M + 1K – 1
13M to 13M + 1K – 1
14M to 14M + 1K – 1
15M to 15M + 1K – 1

Memory Mapped to USER

USER memory decodes an address according to USER's own decode logic.

GRANULARITY Is 16K

When the granularity is 16K, there is no wrap-around. The memory map is as specified.

NOTE

If you use the RESTART option with the I2ICE command, the granularity will be reset to 1K if it previously was set to 16K. The probe hardware retains the 16K mapping boundaries set up previously, but the map display will be based on 1K granularity.

Example

1. Set the granularity to 16K:

```
*GRANULARITY = 16K
*GRANULARITY
16K GRANULARITY
```

HALT

Breaks emulation from the terminal

Syntax

```
HALT [ unit-number[,unit-number]* ]
     [ ALL                        ]
```

Where:

unit-number	is the number of the unit you want to stop (0, 1, 2, or 3) or an expression that evaluates to 0, 1, 2, or 3.
ALL	stops all emulating units.

Discussion

The HALT command stops program emulation. Entering HALT aborts execution from the terminal without altering break or trace specifications. Restarting from HALT begins execution without disrupting the emulating program. Use the GO command to resume execution.

NOTE

CTRL-C (or CTRL-Break, for IBM PC hosts) has no effect on emulating programs.

Example

```
?HALT
*Probe 0 stopped at location 0027:00AEH because of halt
```

HELP

Provides on-line operating assistance

Syntax

HELP[*unit-name*] | *debug-topic* | E*n* | E | *n* |

Where:

HELP	displays the list of HELP information available.
unit-name	refers to the various I²ICE chassis, each containing a probe or a logic analyzer or both. Probes are designated P86, P186, or P286. The logic analyzer is designated PLTA. The default *unit-name* is the current unit.
debug-topic	is one of the help topics shown in the following Example section. Entering HELP *debug-topic* displays information for that topic.
E*n*	displays the expanded error message number *n*. The existence of extended messages is indicated by a [*] symbol following the error. The *n* must be a decimal number.
E	requests the extended error message for the last error. Do not specify a *unit-name* option with this command. Specifying E causes the I²ICE system to default to the unit where the error was generated. An error occurs if you specify any *unit-name*.
n	displays error message number *n* without the expanded text. The *n* must be a decimal number.

Discussion

When the error message display is suppressed with the ERROR command, use the HELP command to display the text of selected error messages. You cannot use the HELP command within any block structure, a REPEAT, DO-END, COUNT, IF, or debug procedure (PROC).

See the Paging entry in this encyclopedia for information on how to control text movement on the screen during the display of HELP information.

Example

1. Display the list of available help information.

```
*HELP

HELP is available for:

        ACTIVE      APPEND      ARMREG      ASM         BACKSLASH   BASE
        BCD         BOOLEAN     BRKREG      BTHRDY      BUSACT      BYTE
        CALLSTACK   CAUSE       CI          COENAB      COMMENTS    CONSTRUCTS
        COUNT       CPMODE      CURHOME     CURX        CURY        DEBUG
        DEFINE      DESCRIPTOR  DIR         DISPLAY     DO          DWORD
        EDIT        ERROR       EVAL        EVTREG      EXIT        EXPRESSION
        GET87       GO          HALT        HELP        HOLDIO      IF
        INCLUDE     INTEGER     INVOCATION  IORDY       ISTEP       KEYS
        LABEL       LINES       LIST        LITERALLY   LOAD        LONGINT
        LONGREAL    MACRO       MAP         MAPIO       MEMRDY      MENU
        MODIFY      MODULE      MTYPE       NAMESCOPE   OFFSETOF    OPERATOR
        PAGING      PARTITION   PATHNAME    PCHECK      PHANG       PINS
        PORT        PORTDATA    PRINT       PROC        PSEUDO_VAR  PUT
        REALS       REFERENCE   REGISTERS   REGS186     REGS286     REGS86
        RELEASEIO   REMOVE      REPEAT      RESET       RETURN      RSTEN
        SASM        SAVE        SELECTOR    SELECTOROF  SHORTINT    STATUS
        STRING      SYSREG      TIMEBASE    TRCBUS      TRCREG      TYPES
        UNIT        UNITHOLD    VARIABLE    VERSION     WAIT        WORD
        WPORT       WRITE       LA          LAACQMODE   LAAUXMEM    LABEGIN
        LACH        LACLKQ      LACNTR      LACOMPARE   LACURSOR    LAEND
        LAFIND      LAGO        LAHALT      LAICELINE   LAMAINMEM   LAOCCURQ
        LAPRETRIG   LAREF       LARESET     LASAVE      LASTATUS    LATHRESH
        LATIME      LATIMEBASE  LATMODE     LATRACE     LATRANSFER  LATRIGWR
        LAVAR       LAWR
```

Cross-Reference

Paging

HOLDIO

Suspends I/O requests to ICE-mapped ports

Syntax

HOLDIO

Discussion

The HOLDIO command suspends I/O requests to ports mapped to ICE, thus allowing you to enter I²ICE commands. All probe-related commands are invalid when the HOLDIO command is active. Resume emulation by entering the RELEASEIO command.

NOTE

Use the HOLDIO command in only one circumstance: after the I²ICE system requests input. If you enter HOLDIO at any other time, the system returns a syntax error.

Example

1. Suspend I/O requests:

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

Cross-Reference

RELEASEIO

I2ICE

Invokes the I²ICE system software

Syntax

```
I2ICE [ CRT [(pathname)] | NOCRT ] [ MACRO [(pathname)] | NOMACRO ] [ SUBMIT | NOSUBMIT ] [RESTART]

[VSTBUFFER (number)] [HELP (pathname)] [ERROR (pathname)] [ P086 (pathname) | P186 (pathname) | P286 (pathname) ]
```

Where:

I2ICE	loads the I²ICE software from drive 0 and the default programs I2ICE.CRT and I2ICE.MAC on the system disk. I2ICE also selects the files I2ICE.OVE and I2ICE.OVH if they are on drive O.
CRT [(*pathname*)]	specifies a CRT file containing the character definitions for terminals other than the Series III or IV development systems or the IBM PC hosts. CRT, with an optional *pathname*, causes the I²ICE system to load the specified file. A fatal error occurs if no CRT file is found. Specifying CRT without a *pathname* loads the file I2ICE.CRT. If you do not have a Series III, Series IV, or IBM PC host, the default is CRT without a *pathname*. (For these hosts there is no CRT file and the CRT option is not needed when you use the I2ICE command.) The abbreviation for CRT is CR.
pathname	describes the location of files on peripheral devices to the I²ICE system. How this information is specified depends on the host system you are using. Refer to the Pathname entry in this encyclopedia for more information on *pathname*.
NOCRT	prevents the I²ICE system from loading the I2ICE.CRT file. Specify NOCRT when you use a Series III or IV or IBM PC terminal and you happen to have a CRT file in your directory. The abbreviation for NOCRT is NOCR.

MACRO [(*pathname*)] — specifies a file containing I²ICE commands to be displayed and executed during initialization. The abbreviation for MACRO is MR.

The MACRO option, with an optional *pathname*, causes the I²ICE system to load the specified file. A fatal error occurs if no MACRO file is found. Specifying MACRO without a *pathname* loads the file I2ICE.MAC.

The default is MACRO without a *pathname*.

NOMACRO — prevents the I²ICE system from loading the I2ICE.MAC file. The abbreviation for NOMACRO is NOMR.

SUBMIT — specifies that the I2ICE program will be used in batch load mode. Using SUBMIT disables the EDIT command and passes all line-editing and command echoing functions to the operating system. You can abbreviate SUBMIT to SM. If a file is loaded using the operating system SUBMIT program and the SUBMIT option in this invocation is not specified, each command is echoed to the terminal twice.

NOSUBMIT — prohibits using the SUBMIT files to load jobs to the operating system. The default is NOSUBMIT. You can abbreviate NOSUBMIT to NOSM.

RESTART — reloads the host development system portion of the I²ICE software. It does not affect the probe, and so it preserves the state of the probe's hardware.

NOTE

For the 80286 probe, if you use the RESTART option with the I2ICE command, the granularity will be reset to 1K if it previously was set to 16K. The probe hardware retains the 16K mapping boundaries set up previously, but the map display will be based on 1K granularity.

VSTBUFFER (*number*) — specifies the amount of physical memory to be used by the virtual symbol table. The virtual symbol table can range from 5K bytes to 61K bytes. The default is 5K bytes. The larger the resident portion of the virtual symbol table, the less time the I²ICE system spends manipulating the virtual symbol table. Increasing the buffer size uses more memory but improves performance. The abbreviation for VSTBUFFER is VSTB.

number — specifies the number of Kbytes in physical memory reserved by the virtual symbol table. The minimum is 5, and the maximum is 61.

HELP (*pathname*) — selects the I²ICE help text file. If you do not use this option, the I²ICE system looks for a file called I2ICE.OVH on the same device that I2ICE.86 resides.

ERROR (*pathname*) — selects the I²ICE error text file. If you do not use this option, the I²ICE system looks for a file called I2ICE.OVE on the same device that I2ICE.86 resides.

P086 (*pathname*) — selects the file containing I²ICE 8086/8088 probe software. If you do not use this option, the I²ICE system looks for a file called I2ICE.O86 on the same device that I2ICE.86 resides.

P186 (*pathname*) — selects the file containing I²ICE 80186/80188 probe software. If you do not use this option, the I²ICE system looks for a file called I2ICE.186 on the same device that I2ICE.86 resides.

P286 (*pathname*) — selects the file containing I²ICE 80286 probe software. If you do not use this option, the I²ICE system looks for a file called I2ICE.286 on the same device that I2ICE.86 resides.

Discussion

The I2ICE command invokes the I²ICE software. (For detailed instructions on loading and executing I²ICE software, see the *I²ICE™ System User's Guide*.) Because the I2ICE command is for use with the host operating system and is not an I²ICE command, you cannot specify it once the I²ICE software is running. The I²ICE software runs in the 8086 environment. When using an Intellec Series III development system with the ISIS operating system, you must invoke the RUN program before invoking the I²ICE software.

If the I²ICE software is loaded into a system configured for the 8086/8088 probe, the I²ICE software checks the user pins. If an unusual state exists (such as a pin held low), the system displays a message advising you to check pin status by entering the PINS command. You must reset the hardware to reflect the following pin values:

8086/8088 probe in MIN mode:

RESET = 1 NMI = 0 HOLD = 0 HLDACK = 0 INTR = 1

8086/8088 probe in MAX mode:

RESET = 1 NMI = 0 RQ/GT0 = 1 RQ/GT1 = 1 INTR = 0

The file containing the I²ICE host software is called I2ICE.86 (or, for IBM PC hosts, I2ICE.EXE); you invoke the software by entering its file name. The other I²ICE files are as follows:

I2ICE.OVE	The I²ICE error text file.
I2ICE.OVH	The I²ICE help text file.
I2ICE.086	The I²ICE 8086/8088 probe software.
I2ICE.186	The I²ICE 186/188 probe software.
I2ICE.286	The I²ICE 286 probe software.

When you invoke I2ICE.86 (or I2ICE.EXE), the I²ICE system assumes that these files exist on the same device as I2ICE.86 (or I2ICE.EXE). You need the probe software for only the probe or probes attached to the host development system.

If you rename I2ICE.86 (or I2ICE.EXE), then you must invoke the software by entering the new name. If you do rename I2ICE.86 (or I2ICE.EXE) and want the I²ICE system to use the default pathnames, rename all the other I²ICE files. For example, if you rename I2ICE.86 (or I2ICE.EXE) to MYFILE.86 (or MYFILE.EXE), the I²ICE system looks for a HELP file called MYFILE.OVH.

With the options on the invocation line you can override the default pathnames and specify the name and location of each I²ICE file. For example, if the invocation line contains P186 (:F1:PROBE.186), the I²ICE system assumes that the 80186/80188 probe software is in a file called PROBE.186 on :F1:.

All the I²ICE files must be valid. For example, if the invocation line contains HELP (:F2:MYHELP), the file MYHELP must really be the I²ICE help file.

Instead of entering the options on the command line, you can construct a configuration file called I2ICE.CFG. If the disk from which you invoke the host software contains a configuration file, the I²ICE system uses that file. (If there are conflicts between the configuration file and the invocation line, the system uses the information on the invocation line.) The configuration file is a text file; it uses the same syntax as the invocation line. The following example is a typical configuration file for using I²ICE software that is on floppy disks (i.e., not on a hard disk):

```
HELP (:F2:I2ICE.OVH) ERROR (:F2:I2ICE.OVE) P186 (:F1:I2ICE.186)
VSTB (10)
```

Examples

1. Run the I²ICE software on the Series III.

```
-RUN I2ICE
```

2. Run the I²ICE software on the Series IV and specify that the I²ICE system load the submit file.

```
-I2ICE.86 SUBMIT
```

3. Run the I²ICE software from the current directory on an IBM PC host. (The prompts shown in the example assume that you have set your PC prompt using the command PROMPT = PG. It is also assumed that you initially loaded your I²ICE software into the directory ICEDIR.)

```
C\ICEDIR>I2ICE
```

Cross-Reference

EXIT
Pathname

I²ICE™ System User's Guide, Installation Instructions

IF

Groups and conditionally executes commands

Syntax

IF *boolean-condition* THEN

[*I²ICE commands*]*

[ELSE [*I²ICE commands*]*]

END[IF]

Where:

IF *boolean-condition* THEN *I²ICE commands*	executes the THEN *I²CE commands* when *boolean-condition* is true. The *boolean-condition* specifies a conditional test whose result evaluates to a TRUE (LSB = 1) or a FALSE (LSB = 0). All I²ICE commands are legal except LOAD, EDIT, HELP, and INCLUDE.
ELSE *I²ICE commands*	executes the ELSE clause when the IF *boolean-condition* is false.

Discussion

An IF block is executed immediately after you enter its END statement.

NOTE

Debug objects are local only in memory type definitions and DO-END blocks. Literals, debug procedures, and all break and trace registers are always global.

Example

1. Create a debug procedure containing an IF block. The debug procedure returns TRUE if the number passed as a parameter (indicated by %0) is evenly divisible by three.

```
*DEFINE PROC divthree = DO
. *IF (%0 mod 3) = = 0 THEN
. . *RETURN true
. . *ELSE
. . *RETURN false
. . *ENDIF
. *END
*divthree (6)
true
*divthree (5)
false
```

Cross-Reference

Boolean condition

INCLUDE

Retrieves command definitions
from a system file

Syntax

INCLUDE *pathname* [NOLIST]

Where:

INCLUDE *pathname*	loads and displays the entire command file into the development system in a form usable by the I²ICE system.
pathname	is the fully-qualified reference to the file you want to include. For information on *pathname*, see the Pathname entry in the *I²ICE™ System Reference Manual*.
NOLIST	suppresses the listing of the included file to the terminal.

Discussion

The INCLUDE command retrieves a command file and executes it.

You can create command files in two ways: Create a file using the screen editor (refer to the Editors entry) or save definitions created during a debug session to a file with the PUT or APPEND commands.

INCLUDE has the following restrictions:

- You can nest INCLUDE commands (limited by available memory), but they must be the last item on a command line.
- An INCLUDE command cannot appear in block structures (i.e., REPEAT, COUNT, IF, DO-END, or a debug procedure).

Examples

1. The following example shows how to retrieve a sequence of I²ICE commands stored in a file named setup.tes. This is useful when creating debug objects in one session that are required in another session.

This example shows one way to reset a circuit connected to the user prototype hardware. By including the previously developed debug procedures to perform this task, debug objects are restored. (If you have an IBM PC host, disregard the symbol ":f3:". If the file that you wish to INCLUDE is in your current disk directory, you would use the command: INCLUDE setup.tes. If the file is on another drive, replace :f3: with d:, where d is the letter of the file's disk drive.)

```
*INCLUDE :f3:setup.tes          /* Retrieve and display a file created with the editor */
*BASE = 10T
*define literally io_base = '0A000h'
*define literally command = '+2'
*define literally io_data = '+4'
*define proc reset_io = do
. *define byte dummy_read
. *count 3
. . *byte io_base command = 0
. . *end
. *byte io_base command = 4eh
. *byte io_base io_data = 0ffh
. *dummy_read = io_base io_data
. *write 'Peripheral reset'
. *end
*reset_io                       /*Invoke the peripheral reset procedure */
Peripheral reset
*
```

2. When a long file is created, you may not want to see it listed to the screen when it is included. The following example shows how the NOLIST option suppresses display. (If you have an IBM PC host, disregard the symbol ":f3:".)

```
*INCLUDE :f3:setup.tes NOLIST
*reset_io                       /*Invoke the peripheral reset procedure */
Peripheral reset
*
```

3. The following example shows how to confirm what debug objects are included when the NOLIST option is specified. The DIR command displays the debug objects in the symbol table after including the procedure from example 1. (If you have an IBM PC host, disregard the symbol ":f3:".)

```
*INCLUDE :f3:setup.tes NOLIST
*DIR DEBUG
IO_BASE  . . . literally  '0a000h'
COMMAND  . . . literally  '+2'
IO_DATA  . . . literally  '+4'
RESET_IO . . . proc
*reset_io
Peripheral reset
*
```

Cross-References

APPEND
Editors
Pathname
PUT

Function that returns the index
of a substring within a given string

Syntax

INSTR (*string-reference1, string-reference2* [, *start*])

Where:

string-reference	can be characters enclosed in apostrophes, a string expression using CONCAT, NUMTOSTR, or SUBSTR functions, or a reference to a CHAR type debug variable.
start	defines where to begin the search in *string-reference1*. The *start* is an index number or an expression that evaluates to an index number from 1 through 254 in the current base.

Discussion

The INSTR function searches for *string-reference2* within *string-reference1* and returns the index (in decimal) of the first character of *string-reference2*.

Examples

1. Return the decimal index of the first occurrence of the substring 'def':

```
*INSTR ('abcdef', 'def')
4
```

2. Define a string variable "longmsg". Find the index of the first instance of the substring 'Add'. To locate the substring 'Add' at the beginning of the second sentence, skip the first instance of this substring by including an index (which is 10).

```
*DEFINE CHAR longmsg = &
**'Addresses < 0 are invalid. Addresses > 1024K are also invalid.'
*INSTR (longmsg, 'Add')
1
*INSTR (longmsg, 'Add', 10)
29
```

INSTR continued

Notice the continuation character (&). It enables you to continue a command to the next line.

Cross-Reference

Strings

INTEGER

Displays or changes memory as 16-bit signed values

Syntax

INTEGER *partition* [=*expression* [, *expression*]* | =*mtype partition*]

Where:

INTEGER *partition*	displays the value of the memory location specified in *partition* as a decimal integer.
partition	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to a 16-bit signed value for INTEGER.
mtype	is any of the memory types except ASM.

Discussion

The INTEGER command interprets the contents of memory as 16-bit signed values, overriding any type associated with the memory contents. Thus, INTEGER .var1 displays the integer that begins at the address of var1, regardless of the type of var1. If the most significant nibble of the unsigned data comprising the integer is 8 through F, the value is interpreted as a negative number and displayed as the 2's complement form of the unsigned data.

Note that the I²ICE system always displays values for signed-integer memory types as decimal numbers, regardless of the selected number base.

Examples

The number base is hexadecimal in the following examples.

1. Display a single value.

```
*INTEGER $
0020:0006H    +3473
```

2. Display several adjacent values:

```
*INTEGER $ LENGTH 0D
0020:0006H +3473    0    0    0 -12187 -30293 +17767 +
297 +4779
0020:001CH    0    0    0 -14113 +18508 +20841 +24135
```

3. Set a single value of type INTEGER:

```
*INTEGER 40:04 = 2567
```

4. Set several adjacent values:

```
*INTEGER 40:04 = 1234, 0ABCD, 3
```

Display the values set (you can set memory locations to signed integer values using a hexadecimal base,but the I²ICE system displays the values in decimal):

```
*INTEGER 40:04 LENGTH 3
0040:0004H    +1234    -21555    +3
```

5. Set a range of locations to the same value (block set):

```
*INTEGER 40:04 LENGTH 10 = 0
```

6. Set a repeating sequence of values:

```
*INTEGER 40:04 LENGTH 10 = 1234, 5678, 9ABC, 0DEF0
```

7. Copy a value from one memory location to another:

```
*INTEGER 40:04 = INTEGER $
```

8. Copy several values (block move):

```
*INTEGER 40:04 = INTEGER $ LENGTH 10
```

9. Copy values with type conversion:

```
*INTEGER 40:04 = BYTE .var2
```

An error message is displayed if the type on the right side of the equal sign cannot be converted to the type on the left. (Refer to the Expression entry in this encyclopedia for the rules concerning type conversions.)

Cross-Reference

Expression
Mtype
Partition

IORDY

A pseudo-variable that allows a system time-out when an I/O access takes more than one second.

Syntax

```
IORDY [ = TRUE
      | = FALSE
      | = boolean-expression ]
```

Where:

IORDY	(I/O ready) with no options displays the current setting (TRUE or FALSE).
TRUE	enables processor I/O time-outs.
FALSE	disables I/O time-outs.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default

TRUE

Discussion

When IORDY = TRUE, a time-out occurs if an I/O access during emulation takes longer than one second. A time-out breaks emulation.

Examples

1. Display the current setting of IORDY:

```
*IORDY
TRUE
```

2. Disable the IORDY time-out:

```
*IORDY = FALSE
```

3. Use IORDY as a variable:

```
*DEFINE PROC ready = DO
. *IF IORDY = = FALSE THEN
. . *IORDY = TRUE
. . *END
. *END
```

Syntax

ISTEP [*increment*] [FROM *address*]

Where:

ISTEP	executes by machine language instruction.
increment	is an unsigned integer expression in the current base specifying the number of steps to take. The default increment is 1. The maximum increment value is 65,535T.
FROM *address*	specifies a starting address where ISTEPs are to begin. The default start address is the current execution point ($). (The Address entry in this encyclopedia contains more information on addresses.)

Discussion

The ISTEP command single-steps through machine language instructions. An ISTEP command executes one instruction and halts. Break messages are not displayed. Use the CAUSE command to display break messages. Use the ASM command to display the current machine language instruction.

NOTE

When you use the 8086/8088 probe, the instruction being single-stepped must not access locations 4 through 0BH. Stepping through a POPF or IRET instruction may clear the trap flag (TF) if the instruction is programmed that way. To enable single-stepping without clearing the TF, define the event register and procedure, as shown in the following example. Because ISTEP uses the hardware break facility, it may slide through an instruction.

```
*DEFINE EVTREG hstep = DO
. *XEM S0 ALWAYS BREAK END
*DEFINE PROC S = GO USING hstep
```

Any NMIs are ignored when stepping using the 8086/8088 probe.

When any of the probes is used, stepping through an instruction that alters a segment register executes two instructions.

Example

1. Step and display the probe processor's registers at each line in the program:

```
*DEFINE PROC rat = DO
. *REPEAT
. . *ISTEP
. . *ASM $
. . *REGS
. . *ENDREPEAT
. *END
```

Cross-References

Address
Expression
LSTEP
PSTEP

Discussion

This entry explains I²ICE system keywords and symbols that are reserved by the I²ICE system software.

Keywords

Following is a list of keywords for the I²ICE system software. If any of these keywords is used as a debug symbol, the I2ICE software will respond with a syntax error message. However, if a keyword has been used as a program variable or label, there is a way to use it in a debug session: Precede the keyword with a double quote (").

For example, in the tutorial PL/M program, there is a variable named *char*. On the list of keywords, you will find *char*. To use this variable in a debug session, enter *"char*.

> NOTE: If you have iLTA software, *LA* keywords will also result in an error message when used as debug symbols. See the *iLTA Logic Timing Analyzer Reference Manual* (order number 163257) for the iLTA keywords that begin with the "LA" prefix.

ACTIVE	BOTH	CLIPSIN	DEFINE
ADDRESS	BP	CLIPSOUT	DFL
ADR	BREAK	COENAB	DH
AFL	BRKREG	CONCAT	DI
AFTER	BTHRDY	COREQ	DIR
AH	BUFBREAK	COUNT	DISABLE
AL	BUSACT	CPMODE	DISARM
ALL	BUT	CS	DL
ALWAYS	BX	CSAR[2]	DMA0[1]
AND	BYTE	CSBAS[2]	DMA1[1]
APPEND	CALL	CSCTRL[1]	DO
ARM	CALLSTACK	CSLIM[2]	DS
ARMREG	CAUSE	CSSEL[2]	DSAR[2]
ARRAY	CFL	CTR	DSBAS[2]
ASM	CH	CURHOME	DSLIM[2]
AT	CHAR	CURX	DSSEL[2]
AX	CI	CURY	DWORD
BASE	CL	CX	DX
BCD	CLEAR	CYCLES	EDIT
BH	CLEAREOL	DATA	ELSE
BL	CLEAREOS	DBG	ELSEIF
BOOLEAN	CLIPS	DEBUG	ENABLE

[1] 80186/80188 only; [2] 80286 only

Keywords continued

END
ENDCNT
ENDCOUNT
ENDREPEAT
ENUMERATION
ERROR
ES
ESAR[2]
ESBAS[2]
ESLIM[2]
ESSEL[2]
EVAL
EVTREG
EXIT
EXTINT
FALSE
FCS[2]
FCW
FDA
FDOFF[2]
FDSEL[2]
FETCH
FIA
FILE
FIO
FIP[2]
FLAGS
FLD2E
FLDL2T
FLDLG2
FLDLN2
FLDP1
FOREVER
FPATAN
FPTAN
FSQRT
FSW
FTW
FULLBUF
FYL2X
FYL2XP1
GDT[2]
GDTBAS[2]
GDTLIM[2]
GET87
GLOBAL
GO
GOTO
GRANULARITY
GUARDED
HALT
HELP
HOLDIO
HPORT
(reserved for Intel)
HS
IBYTE
(reserved for Intel)
ICE
IDT[2]
IDTBAS[2]
IDTLIM[2]
IF
IFL
INCLUDE
INCREMENT
INPUT
INSTR
INSTRUCTION
INTEGER
INTRIN
(reserved for Intel)
INTRPT[1]
IOPL[2]
IORDY
IP
IS
ISTEP
LABEL
LAST
LDT[2]
LDTAR[2]
LDTBAS[2]
LDTLIM[2]
LDTSEL[2]
LENGTH
LEVELS
LINE
LINK
LIST
LITC[1]
LITERALLY
LOAD
LONGINT
LONGREAL
LSTEP
MAP
MAPIO
MB
MEMRDY
MOD
MODULE
MS
MSW[2]
NAMESCOPE
NEWEST
NEXT
NFL[2]
NO
NOCODE
NOCR
NOLINES
NOLIST
NOSYMBOLS
NOT
NP
NS
NUMTOSTR
OCCURRENCE
OFFSETOF
OFL
OHS
OLDEST
OR
ORIF
OUTPUT
OUTSIDE
PCHECK
PFL
PHANG
PINS
POINTER
PORT
PORT__ADDRESS__BASE
(reserved for Intel)
PORTDATA
PRINT
PROC
PROCEDURE
PSTEP
PUBLICS
PUT
QSTAT
READ
REAL
RECORD
REGS
RELEASEIO
RELREG[1]
REMOVE
REPEAT
RESET
RETURN
RSTEN
S0
S1
S2
S3
SASM
SAVE
SCREEN
SCTR
SEL286
SELECTOR
SELECTOROF
SEM
SET
SFL
SHORTINT
SI
SLINK
SP
SS
SSAR[2]
SSBAS[2]
SSLIM[2]
SSSEL[2]
ST0
ST1
ST2
ST3
ST4
ST5
ST6
ST7
STACK
START
STATUS

[1] 80186/80188 only; [2] 80286 only

STRLEN	TAG	TRBAS[2]	USING
STRTONUM	TEMPREAL	TRCBUS	VERSION
SUBSTR	TFL	TRCREG	WAIT
SYMBOL	THEN	TRIG	WAITSTATE
SYMBOLIC	TIL	TRLIM[2]	WHILE
SYMBOLS	TIMEBASE	TRSEL[2]	WITH
SYSARM	TIR0[1]	TRUE	WORD
SYSBEAKIN	TIMER1[1]	TSS	WPORT
SYSDARM	TIMER2[1]	UNIT	WRITE
SYSREG	TO	UNITHOLD	XCTR
SYSTEM	TR[2]	UNTIL	XEM
SYSTRACE	TRACE	US	XLINK
SYSTRACEIN	TRAR[2]	USER	XOR
SYSTRIG			ZFL

[1] 80186/80188 only
[2] 80286 only

Delimiters

The following delimiters cannot be included as part of debug symbols.

&	%	(	)	*	+	,	.	–	&	#
:	;	<	>	< =	> =	[	]	/	\	

Characters that can be used are the question mark (?), an underscore(_), and an at symbol (@). The dollar sign ($) is permitted, but ignored.

A double quote (") is used if a program variable is also an I²ICE keyword. Precede the variable with a " when using it in a debug session.

LIST

Opens or closes a list file

Syntax

{ LIST *pathname* | NOLIST }

Where:

LIST	displays the *pathname* of the current list file.
LIST *pathname*	opens a list file named *pathname*.
pathname	is the fully-qualified reference to the file you want for a list file (e.g., :f1:listing). The file is created if it does not exist; if it already exists, the question "Overwrite existing file? (y or [n])" is displayed.
NOLIST	closes the open list file.

Discussion

A list file is an I²ICE utility file. Typically, a list file is used as a debug session log. All interactions between the I²ICE system and the terminal (except edits) are recorded in an open LIST file.

You can open only one list file at a time. Close list files by issuing the NOLIST command or by terminating the debug session.

Example

1. Open a list file named AUG1.84 (if you have an IBM PC host, disregard the symbol ":F2:"):

```
*LIST :F2:AUG1.84
```

Cross-Reference

Pathname

LITERALLY

Defines, modifies, displays, or removes a name that the I²ICE system interprets as a previously-defined character string

Syntax

DEFINE LITERALLY *literally-name* = *'character-string'*

Where:

DEFINE LITERALLY *literally-name* = *'character-string'*	replaces *literally-name* with the string in *'character-string'* whenever *literally-name* is invoked.

Discussion

LITERALLY definitions are special debug objects with character strings as values. They are similar to PL/M LITERALLY definitions.

When you enter a command that has a LITERALLY name in it, as soon as you press the space bar after the name, the name is automatically expanded on the screen. If you wish to disable this automatic expansion feature, use the following command:

```
MENU = 0
```

(This command also deletes the syntax menu at the bottom of the screen.) To re-enable the automatic expansion feature (and to reactivate the syntax menu display), enter MENU = 1. You can also use CTRL-V to toggle off and on the automatic expansion feature (and the menu).

When the I²ICE scanner sees a LITERALLY name in a debug object such as a PROC, the scanner replaces that name with the character string value that defines it. However, if the command containing the LITERALLY name is echoed to the terminal, the terminal displays the LITERALLY name rather than the defined character string.

With LITERALLY definitions, you can abbreviate keywords or complete commands, which can have up to 254 characters. With LITERALLY definitions you can also create mnemonics, such as substituting "temp_control_reg" for the command PORT(40H).

LITERALLY definitions are always global. They must not duplicate a keyword.

Examples

1. Create LITERALLY definitions:

```
*DEFINE LITERALLY def = 'define'
*def LITERALLY lit = 'literally'
*def lit len = 'length'
*def lit ioreg = 'port (4dh)'
```

2. Display LITERALLY definitions:

```
*LITERALLY len
DEFINE LITERALLY len='length'
```

3. Display the directory of LITERALLY definitions:

```
*DIR LITERALLY
DEF..........literally 'define'
LIT..........literally 'literally'
LEN..........literally 'length'
IOREG........literally 'port (4DH)'
```

4. Edit LITERALLY definitions:

```
*EDIT len
```

5. Delete the LITERALLY definition for LEN:

```
*REMOVE len
```

6. Delete all LITERALLY definitions:

```
*REMOVE LITERALLY
```

Cross-References

Keywords
Name
Strings

LOAD

8086/8088 probe specific
80186/80188 probe specific

Copies an object from a file into mapped memory

Syntax

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

Where:

LOAD *pathname*	copies the object file (including code, line numbers, and symbols, if present) from the designated file into mapped memory.
pathname	is the fully-qualified reference to the file you want to load.
NOCODE	ignores content records (the code and data) when loading; just the symbol table information is loaded.
NOLINES	ignores debug line number information when loading.
NOSYMBOLS	ignores debug symbol table information when loading.
APPEND	allows multiple symbolic loads without purging the symbol table.

Discussion

Before loading, you must link and locate your program. The I²ICE system accepts only absolute object files, not run-time locatable (RTL) object files or load-time locatable (LTL) object files.

If the load file contains a start address, that address is loaded into the CS:IP registers. If the address is not present (as will always be the case when a file has been stored using the SAVE command), a warning message is displayed. Other segment registers may be initialized, depending on the programming language used to create the load file.

You can load all of a file or part of a file by selecting LOAD command options. You can combine the options in any order. If you do not specify any options, all symbol table and line number information present in the file is loaded into mapped program memory. The DEBUG control on the assembler or compiler causes program symbol and debug information to be included in the object file.

LOAD (8086/8088 and 80186/80188) continued

When loading a translated file into memory, the I²ICE system first determines which probe is active.

NOTE

You must map memory (using the MAP command) before loading your program.

You cannot use the LOAD command in block structures (i.e., REPEAT, COUNT, IF, DO-END, or debug procedures).

Examples

1. Load an object file. (If you have an IBM PC host, disregard the symbol ":f1:"; assume the file is in your current hard disk directory. To load the file, you would use the command: LOAD prog01.186.)

```
*LOAD :f1:prog01.186
*
```

   The I²ICE system returns the * prompt when loading is complete.

2. Load an object file with code only, suppressing symbols and line numbers. (If you have an IBM PC host, disregard the symbol ":f3:".)

```
*LOAD :f3:scaner.86 NOSYMBOLS NOLINES
*
```

3. Load program symbols and line numbers only (no code). (If you have an IBM PC host, disregard the symbol ":f1:".)

```
*LOAD :f1:comlib.asm NOCODE
*
```

Cross-References

MAP
MAPIO
Pathname
SAVE

LOAD

80286 probe specific

Copies an object file from disk into mapped memory

Syntax

LOAD *pathname* [NOCODE] [NOLINES] [NOSYMBOLS] [SEL286] [APPEND]

Where:

LOAD	loads an object file into mapped memory. The program file must be absolute code, not load-time-locatable code.
pathname	is the fully-qualified reference to the file you want to load.
NOCODE	prevents loading of the code and data.
NOLINES	prevents loading line numbers.
NOSYMBOLS	prevents loading the symbols.
SEL286	loads a file that is in 8086 object module format (OMF) when you want the program's addresses interpreted as 80286 selector:offset pairs. The selector of an 80286 address is an index into either the LDT or the GDT. If the file is an 8086 OMF and the SEL286 option is not present, the I²ICE system obtains the base address of the segment by shifting the selector left by four bits.
APPEND	adds the symbol table of the current LOAD to the symbol table of the previous LOAD.

Default Value

By default the I²ICE system loads code, data, line numbers, and the symbol table. By default the symbol table of the current load overwrites the symbol table of the previous load.

The I²ICE host software has two loaders: the 8086 loader and the 80286 loader. When you load a file that is in 8086 OMF, the I²ICE host software uses the 8086 loader. When you load a file that is in 80286 OMF, the I²ICE host software uses the 80286 loader. You need not specify what OMF the file is in.

Discussion

The LOAD command loads a file from disk into mapped memory. Memory must be mapped to the physical locations expected by the file. The I²ICE system expects absolute code, not load-time locatable code.

Refer to the Using the Initialization Segment section (in the Probes chapter of the *I²ICE™ System User's Guide*) for additional information on loading, including loading the initialization segment with the 8086 loader.

Constructing a Program File

Construct a program file by compiling (or assembling) the source file, binding the object file, and building the bound file. To compile the source file, use one of the following compilers:

PL/M-86	Version 2.1 or greater
PL/M-286	Version 2.5 or greater
PASCAL-86	Version 2.0 or greater
PASCAL-286	Version 3.1 or greater
ASM-86	Version 1.1 or greater
ASM-286	Version 1.1 or greater
FORTRAN-86	Version 1.1 or greater
FORTRAN-286	Version 3.0 or greater
C-286	Version 3.0 or greater

Include the DEBUG and TYPE options with the compiler or assembler. To bind the resulting object file, use BIND286 (version 3.0 or greater) with the NOLOAD option. To build the resulting bound file, use BUILD286 (version 3.0 or greater) with the BOOTLOAD option.

Mapping Program Memory

You must have sufficient mapped memory to contain the object file. The physical locations expected by the object file must be mapped to existing memory.

The SEL286 Pseudo-Variable and the SEL286 LOAD Option

The SEL286 pseudo-variable determines whether the I²ICE system performs 8086 address translation (SEL286 is FALSE) or 80286 address translation (SEL286 is TRUE). When you load an 8086 OMF, the SEL286 pseudo-variable becomes FALSE. When you load an 80286 OMF, the SEL286 pseudo-variable becomes TRUE.

The SEL286 option of the LOAD command is distinct from the SEL286 pseudo-variable. Use the SEL286 option to load a file in 8086 OMF and have its addresses interpreted as 80286 addresses during the load. The selector:offset pairs in the 8086 OMF must point to valid descriptors in a local or global descriptor table (which must be already set up).

When you load an 8086 OMF and specify the SEL286 option, the I²ICE system sets the SEL286 pseudo-variable to TRUE. Setting the SEL286 variable to TRUE after loading an 8086 OMF without including the SEL286 option is insufficient because 80286 address translation must occur during the load.

The Protection Enabled Flag

The LOAD command affects the protection enabled flag (PEF) in the MSW. If the program file is an 80286 OMF, PEF becomes 1, and the loader is in protected mode. If the program file is an 8086 OMF, PEF becomes 0, and the loader is in real mode.

Initial Values for Registers

When you load a program file that is an 80286 OMF, the 80286 registers attain the values specified by the 80286 builder.

Example

1. Load only the code from an object file on disk drive 1 into mapped memory. (If you have an IBM PC host, disregard the symbol ":F1:"; assume the file is in your current disk directory. To load the file, you would use the command: LOAD cmaker.286 NOLINES NOSYMBOLS.)

```
*LOAD :F1:cmaker.286 NOLINES NOSYMBOLS
```

Cross-References

Pathname
PCHECK
SEL286
SAVE

LONGINT

Displays or changes memory
as 32-bit signed values

Syntax

LONGINT *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

LONGINT *partition*	displays the value of the location specified by *partition* as a long integer in decimal.
partition	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to a 32-bit signed value for LONGINT.
mtype	is any of the memory types except ASM.

Discussion

The LONGINT (long integer) command interprets the contents of memory as 32-bit signed values, overriding any type associated with the memory contents. Thus LONGINT .var1 displays the 32-bit integer that begins at the address of var1, regardless of the type of var1.

If the most significant nibble of any LONGINT is 8 through F, it is interpreted as a negative number and the value displayed is the 2's complement form of the unsigned data.

You cannot use LONGINT in a block structure (i.e., REPEAT, COUNT, IF, DO-END, or debug procedures).

Note that the I²ICE system always displays values for signed-integer memory types as decimal numbers, regardless of the selected number base.

Examples

Hexadecimal is assumed in the following examples.

1. Display a single value:

```
*LONGINT $
0020:0006 +16855312
```

2. Display several adjacent values:

```
*LONGINT $ LENGTH 5
0020:0006H +16855312 +16667999 +12321330 +02340002-10333700
```

3. Set a single value of type LONGINT:

```
*LONGINT 40:04 = 012345678
```

4. Set several adjacent values:

```
*LONGINT 40:04 = 12345678, 0ABCD, 3
```

Display the values set (you can set memory locations to signed integer values using a hexadecimal base, but the I²ICE system displays the values in decimal):

```
*LONGINT 40:04 LENGTH 3
0040:0004H +305419896 +43981 +3
```

5. Set a range of locations to the same value (block set):

```
*LONGINT 40:04 LENGTH 10 = 0
```

6. Set a repeating sequence of values:

```
*LONGINT 40:04 LENGTH 10 = 1234, 56789ABC, 0DEF0
```

7. Copy a value from one memory location to another:

```
*LONGINT 40:04 = LONGINT $
```

8. Copy several values (block move):

```
*LONGINT 40:04 = INTEGER $ LENGTH 10
```

Cross-References

Expression
Mtype
Partition

LONGREAL

Displays or changes memory
as 64-bit floating-point values

Syntax

LONGREAL *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

LONGREAL *partition*	displays the memory location specified by *partition* as a long real number in scientific format.
partition	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to a 64-bit floating-point value for LONGREAL.
mtype	is any of the memory types except ASM.

Discussion

The LONGREAL command interprets the contents of memory as 64-bit floating-point decimal values, overriding any type associated with the memory contents. Thus, LONGREAL .var1 displays the 64-bit floating-point value that begins at the address of var1, regardless of the type of var1.

Examples

The following examples show the ICE system responses in decimal because all real numbers are displayed in decimal, regardless of the base of the input information.

1. Display a single value:

```
*LONGREAL $
0020:0006H +3.3657976670200075E -199
```

2. Display several adjacent values:

```
*LONGREAL $ LENGTH 3
0020:0006H +3.3657976670200756 -199 +1.8592913465363336 -246
0020:0016H -7.2718499473213626 +214
```

3. Set a single value of type LONGREAL:

```
*LONGREAL 40H:04H = 0.00012
```

4. Set several adjacent values:

```
*LONGREAL 40H:04H = 1212121212121212, 0.00012,  -12000
```

Display the values set:

```
*LONGREAL 40H:04H LENGTH 3
0040:0004H +1.2121212121212126 +15 1.2000000000000006 -4
0040:0014H -1.2000000000000006 +4
```

5. Set a range of locations to the same value:

```
*LONGREAL 40H:04H LENGTH 10 = 0
```

6. Set a repeating sequence of values:

```
*LONGREAL 40H:04H LENGTH 10 = 5.678,  -2300, 23456,  -7.567
```

7. Copy a value from one memory location to another:

```
*LONGREAL 40H:04H = LONGREAL $
```

8. Copy several values (block move):

```
*LONGREAL 40H:04H = LONGREAL $ LENGTH 10
```

Cross-References

Expression
Mtype
Partition

LSTEP

Single-steps through user programs by high-level language instructions

Syntax

LSTEP [*increment*] [FROM *address*]

Where:

LSTEP	executes by numbered high-level language statements.
increment	is an unsigned integer expression in the current base specifying the number of steps to take. The default increment is 1. The maximum increment value is 65,535T.
FROM *address*	specifies a starting address where LSTEP is to begin. The default start address is the current execution point ($). (The Address entry in this encyclopedia contains more information on addresses.)

Discussion

The LSTEP command single-steps through user programs by numbered high-level language statements. The LSTEP command executes the next consecutive statement and halts. Break messages are not displayed. Use the CAUSE command to display break messages.

After LSTEP executes a line, it displays a message of the following form:

[:*module-name#line-number*]

NOTE

When you use the 8086/8088 probe, the instruction being single-stepped must not access locations 4 through 0BH. Stepping through a POPF or IRET instruction may clear the trap flag (TF) if the instruction is programmed that way. To enable single-stepping without clearing the TF, define the event register and procedure, as shown in the following example. Because LSTEP uses the hardware break facility, it may slide through an instruction.

```
*DEFINE EVTREG hstep = DO
. *XEM S0 ALWAYS BREAK END
*DEFINE PROC S = GO USING hstep
```

Any NMIs are ignored during single-stepping using the 8086/8088 probe.

When you use any of the probes, stepping through an instruction that alters a segment register executes two instructions.

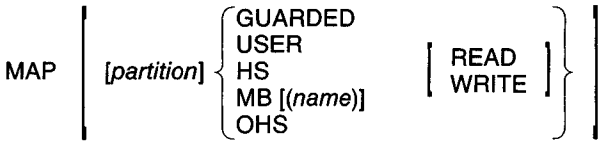
Cross-References

Address
Expression
ISTEP
PSTEP

MAP

Displays or sets physical locations for program memory

Syntax



Where:

MAP	with no options, displays the current memory map.
partition	is a single address, an expression that evaluates to a single address, or a range of addresses. The range is specified as either *address* TO *address* or *address* LENGTH *number-of-bytes*. The partition is in multiples of 1K bytes (e.g., 2K = 2048 bytes).
GUARDED	reports attempts to access memory in the location specified by *partition*. Initially, all program memory is GUARDED.
USER	directs memory references to your prototype hardware. If you are using an 8087 or 80287 external coprocessor, you must map all of program memory to USER. If you are using an 8089 external coprocessor, you must map all memory that the 8089 accesses to USER.
HS	directs memory references to high-speed memory on the MAP board in 1K-byte blocks.
MB [(*name*)]	directs memory references to the MULTIBUS expansion memory in the host development system. [The IBM PC hosts cannot map to MB.] The *name* option allows more than one *partition* of addresses to be mapped to the same physical memory (shared memory). The *name* can be one to six characters long.

OHS	directs memory references to the optional high-speed memory board (in the instrumentation chassis) in 16K-byte blocks.
READ	specifies that *partition* is read only. Emulation breaks if a write occurs.
WRITE	suppresses the normal read-after-write verification on program loads and memory writes from the terminal.

Discussion

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. Because all memory is initially guarded, you must map memory before loading programs. The MAP command displays or changes the map. The *partition* option specifies the size of the map, while the other options specify the locations and whether they are READ or WRITE.

Specifying the Number of Blocks to be Mapped

The memory map size is described to the I²ICE system in blocks using the *partition* option. The high-speed memory board is mapped in 1K-byte blocks, and the optional high-speed memory board is mapped in 16K-byte blocks. Exceeding the size of available memory causes an error, and nothing is mapped.

Addresses specified in *partition* must begin on a block boundary. When the starting address does not begin on a block boundary or the last location in the range does not fill a whole block, the I²ICE system automatically expands the map to the next boundary and reports the expansion.

The *partition* has two forms, a TO form and a LENGTH form. The TO form maps memory from a starting address TO an ending address. The LENGTH form maps memory starting from an address for the specified number of bytes. If you omit *partition*, the entire address space is mapped to the location specified.

Mapping Blocks to Guarded Memory

Initially, all blocks in the map are guarded. If the program accesses guarded memory during emulation, a break occurs after completion of the current instruction. Note that the access does occur. An error occurs if guarded memory is accessed from the terminal. You can reset all of the blocks to guarded with the MAP or RESET MAP command.

Mapping to the User System

When mapping occurs, no check is made to ensure that the amount of memory installed in the prototype matches the map. However, the system normally performs a read-after-write verification during program load. When your system memory is ROM or PROM, use the READ option to avoid an error message on program load.

NOTE

To perform real math operations, you must map memory to your system when using a coprocessor on the system bus.

Your system receives both the read or write signals from memory and the data for writes generated by the probe processor, regardless of the map. The map determines the source of data for reads. When mapped to USER, data for reads is accepted through the chip interface connector. Otherwise, user data is ignored.

Mapping to High-speed Probe Memory

Each probe contains 32K bytes of mappable high-speed memory. You can map probe memory to any location in 1K increments. Probe memory must not overlap any other memory space.

Mapping to MULTIBUS® Memory in the Development System

[NOTE: Mapping to MULTIBUS memory is not available for IBM PC hosts.]

The MULTIBUS (MB) expansion memory must reside on the same physical bus as the I²ICE interface board. Wait-states are automatically inserted when MULTIBUS memory occurs since all MULTIBUS activity must arbitrate for control of the bus.

The *name* option assigns a temporary name to the portion of user memory mapped to MULTIBUS memory. To enable two or more blocks to share the same area of MULTIBUS memory, map all blocks to MB with the same name for all. The partition sizes of shared blocks must match.

Unlike other types of memory, MULTIBUS expansion memory has one mapping restriction. Usually, any block of mapped memory can be remapped by entering the MAP command again with a new *partition*. When memory has been mapped to MB, there are only two ways to change it: reset the entire map or remap the entire MB area. Remapping just a portion of MB memory produces an error. An example is provided in the following Example section.

Mapping to Optional High-speed Memory

The instrumentation chassis has extra slots for up to two optional 128K-byte high-speed (zero wait state) memory boards. If the optional memory is installed in the chassis, you can map program memory to it in 16K-byte blocks on 16K-byte boundaries.

When changing the map, a new *partition* can partially overlap a *partition* previously mapped to optional high-speed memory. However, the boundary (start or end) of the new *partition* that falls within the partition previously mapped to optional high-speed memory must be on a 16K-byte boundary.

Read and Write Controls on the MAP Command

The READ control on the MAP command designates the mapped partition as read-only (write-protected during emulation). Without this control, program memory can be both read and written. Emulation breaks if a read-only address is written (the write is executed anyway). The break occurs after completing the instruction that produced the write.

The WRITE control suppresses any reads, such as the read-after-write verification. The READ option is useful for memory blocks that are ROM or memory-mapped I/O. The WRITE option is useful if memory is write-only memory.

Normally, the system verifies program memory at two times:

- During program loads
- When you change memory from the terminal

You can use READ and WRITE controls with any of the physical memories available to the map.

Lock Prefix on Instructions

Instructions with a bus lock prefix are supported during emulation when the program memory is mapped to the MULTIBUS memory. When program memory is mapped to USER, the LOCK pin on the user plug is active during locked instructions.

Restrictions

Note that if your prototype system is connected to the I²ICE system, I/O data always goes out to your prototype system, whether the I/O ports are mapped to USER or I²ICE. Mapping to the I²ICE system only prevents the I²ICE system from receiving user system input. Thus, you should disconnect your system if your prototype system must not respond to I²ICE system output.

Examples

1. Display the current memory map:

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

2. Map to USER prototype memory:

```
*MAP USER                                   /* Maps all blocks */

*MAP 0 LENGTH 4K USER                       /* Maps four 1K-byte blocks */
```

3. Three ways to restore memory to the guarded state:

```
*MAP 20K LENGTH 2K GUARDED                  /* Guards 2 blocks */

*MAP GUARDED                                /* Sets all blocks to GUARDED */

*RESET MAP                                  /* Another way to set all blocks to GUARDED */
```

4. Map all blocks to probe memory:

```
*MAP 0 LENGTH 32K HS                        /* Map lowest 32K addresses */
```

5. Map to MULTIBUS memory in the development system [not available with IBM PC hosts]:

```
*MAP 4K LENGTH 1K MB
```

6. Map two program address spaces to the same area of MULTIBUS memory, using the name COMMON [not available with IBM PC hosts]:

```
*MAP 16K LENGTH 8K MB (COMMON)
*MAP 32K LENGTH 8K MB (COMMON)
```

7. Map to an optional high-speed 128K-byte memory board:

```
*MAP 32K LENGTH 128K OHS
```

8. Designate a partition of memory as read-only:

```
*MAP 0 LENGTH 32K USER READ
```

9. Designate a partition of memory as write-only:

```
*MAP 64K LENGTH 8K USER WRITE
```

10. Memory, once mapped to MULTIBUS memory, cannot be partially remapped. The following example shows this error condition [not relevant for IBM PC hosts]:

```
*MAP 0 LENGTH 4K MB
*MAP 0 LENGTH 2K USER
SEVERE ERROR #265:
Illegal map change.
```

To remap MULTIBUS memory, all the MB blocks must be remapped as shown:

```
*MAP 0 LENGTH 4K MB
*MAP 0 LENGTH 6K USER                    /* Valid map change */
```

11. The following example shows how the I²ICE system adjusts partitions to match block boundaries. All memory is initially guarded. The partition entered is less than a complete 1K-byte block. The I²ICE system adjusts the boundary upward to completely enclose the partition requested.

```
*BASE = DECIMAL
*MAP                          /*Display the current map settings */
MAP 0K LENGTH 0124K GUARDED
*MAP 1 TO 10 HS               /*Partition not on a 1K-byte
                                boundary*/
WARNING: Map address boundaries changed to match hardware
*MAP                          /*Display the 1K-byte boundary change*/
MAP 0K LENGTH 1K HS
MAP 1K LENGTH 1023K GUARDED
```

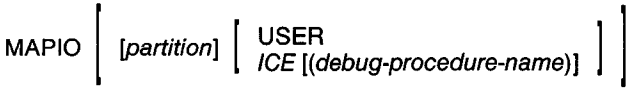
Cross-References

Name
Partition

MAPIO

Displays or sets physical locations for I/O ports

Syntax



Where:

MAPIO	displays the current map of I/O port address blocks. One block is 64 bytes.
partition	is an entry specifying a range of addresses with one of the following forms: *port-address* to *port-address* *port-address* LENGTH *number-of-bytes* Omitting *partition* maps all 64K bytes of I/O space.
USER	transfers data values between the user's prototype system and the I²ICE probe.
ICE	transfers I/O data values between the terminal, not the prototype system, and the I²ICE probe.
ICE [(*debug-procedure-name*)]	calls the named procedure when I/O accesses occur. The I/O data values are transferred between the I²ICE probe and the debug procedure designed to simulate the prototype I/O operation.

Discussion

The I²ICE system uses the I/O port map to control input to and output from peripherals. With the MAPIO command you can display or change the I/O port map. All mapped I/O data is displayed on the host system terminal.

Mapping by Blocks

The I/O port addresses are mapped in blocks using the MAPIO command, as program memory is mapped in blocks using the MAP command. There are 64K bytes of I/O space, divided into 1K-byte blocks. Each block is 64 bytes long.

If an address partition is specified that is not on one of the 64-byte block boundaries, the system expands the partition to the next block boundary and displays the following message:

```
WARNING: MAPIO address boundaries changed to match hardware
```

I/O Simulation Using the Terminal

The ICE option causes I/O requests to appear on the terminal. When input data is required, the following message is displayed:

```
UNIT n PORT mH REQUESTS type INPUT (ENTER VALUE) :
```

Where:

n	is the unit number.
*m*H	is the port number in hexadecimal.
type	is BYTE or WORD.

Enter either the desired data values or the command HOLDIO. Data values cannot be expressions.

You can halt emulation during an I/O operation to enter I²ICE commands. Any commands except PORT and WPORT are valid. When you enter HOLDIO, emulation and I/O requests for that unit are suspended so that you can enter I²ICE commands. To resume emulation and the flow of I/O requests, enter the command RELEASEIO. If the I/O requests for that unit have not been suspended, entering RELEASEIO causes the input request to be repeated.

If you enter information incorrectly, the following message is returned:

```
Input options are a number or HOLDIO
 UNIT n PORT mH REQUESTS type INPUT (ENTER VALUE) :
```

When output data values are returned, they are displayed in the following format:

```
UNIT n PORT mH OUTPUT type value
```

Where:

value	is the numerical value (hexadecimal) of the output data.
PORT	is the port number displayed in hexadecimal.
UNIT	is the unit number displayed in decimal.

I/O Simulation Using an I/O Debug Procedure

An I/O procedure is a special case of debug procedures. It is created, displayed, modified, and removed in exactly the same way that all debug procedures are (see the entry for PROC in this encyclopedia). However, to be useful, an I/O procedure should simulate your system's handling of I/O data.

If you specify an I/O debug procedure name in the ICE option of the MAPIO command, I/O requests within the specified *partition* generate a call to the debug procedure of that name. When the debug procedure is invoked in this context, its parameters are set to identify the form of the I/O request as follows:

Parameter	Interpretation
%0	port number
%1	Boolean value: TRUE if read requested; FALSE if write requested
%2	Boolean value: TRUE for byte port access, FALSE for word port access

Within the I/O debug procedure, use the built-in pseudo-variable PORTDATA to read and write the I/O port. The PORTDATA pseudo-variable is only valid when used inside a procedure that is executed as a result of an I/O access. Any other use results in an error.

When the procedure simulates input (the unit wants to read data from the procedure), the procedure must have a statement with the following syntax:

```
PORTDATA = port-value
```

Where:

port-value	is a positive whole number of type byte or word, depending on the size of your port.

If the procedure supplies more than one value, the system returns the following message:

```
Too many values supplied. Kept the last one.
```

When the procedure simulates output (the current unit wants to write data to the procedure), the procedure can either receive the data in a variable or write it to the terminal. The syntax for writing to a variable is as follows:

variable = PORTDATA

The syntax for writing data values to the terminal is as follows:

WRITE PORTDATA

An error message is returned and the I/O request is handled on the terminal if a procedure exists and any of the following errors occur while emulating:

- An error occurs while the procedure is being executed
- The unit wants to read data and the procedure supplies no data
- The procedure tries to write data when the unit requests a read
- The procedure tries to read data when the unit requests a write

An error that occurs while executing a PORT or WPORT command in the procedure is not recoverable. In this case, an error message is displayed followed by the prompt (*).

Restrictions

Note that if your target system is connected to the I²ICE system, I/O data always goes out to your system, whether the I/O ports are mapped to USER or the I²ICE system. Mapping to the I²ICE system only prevents the I²ICE system from receiving your system input. Thus, you should disconnect the user (target) system if you do not want it responding to the I²ICE system output.

Examples

1. Display I/O port mapping:

```
*MAPIO
MAPIO 0000:0000H LENGTH 00000400 ICE
MAPIO 0040:0000H LENGTH 0000FC00 USER
```

2. The following examples (2a through 2c) simulate data input using the terminal. The examples are taken from the following disassembled program, which is a loop requiring input to port 22H.

```
*BASE=10T
*ASM 0 LENGTH 5
000000H     90          NOP
000001H     E522        IN AX,22H ; +35T
000003H     90          NOP
000004H     EBFA        JMP A=0000H ; $-4
000006H     90          NOP
```

a.
```
*GO FROM 0 TIL 4
*UNIT 0 PORT 22H REQUESTS WORD INPUT (ENTER VALUE)*7a66h
*Probe 0 stopped at 0000:0000 because of execute break
```

NOTE

After the GO command in example 2a, the only user input was '7a66h'.

Confirm that the port values entered are now in register AX:

```
*AX
7A66
```

b. Using HOLDIO and RELEASEIO:

```
*GO FROM 0:0 FOREVER
*UNIT 0 PORT 22H REQUESTS WORD INPUT (ENTER VALUE)*6788
*UNIT 0 PORT 22H REQUESTS WORD INPUT (ENTER VALUE)*holdio
*EVAL 6788h                                   /* HOLDIO permits command entry */
1100111110001000Y 26504T 6788H '.g..'
*releaseio
*UNIT 0 PORT 22H REQUESTS WORD INPUT (ENTER VALUE)*1234h
```

c. System response to incorrect input:

```
*GO FROM 0:0 TIL 3
UNIT 0 PORT 22H REQUESTS WORD INPUT (ENTER VALUE)*5+6
Input options are a number or HOLDIO
```

No expressions are allowed.

```
UNIT 0 PORT 22H REQUESTS WORD INPUT (ENTER VALUE)*aaah
Input options are a number or HOLDIO
```

Hexadecimal values must have a leading zero.

3. The following example demonstrates data output from the terminal. The data output is required by the following program, which asks for data output from port 23H. Assume that AL is initialized.

```
*BASE=10T
*ASM 0 LENGTH 5
000000H      90          NOP
000001H      E623        OUT 23H,AL
000003H      90          NOP
000004H      EBFA        JMP A=0000H ; $-4
000006H      90          NOP

*GO FROM 0 TIL 3
UNIT 0 PORT 23H OUTPUT BYTE *55                    /*User input is 55*/
Probe 0 stopped at 0000:0004 because of execute break
*AL                                                /*Display AL register*/
55
```

4. The following examples show how to use the ICE *debug-procedure-name* option, which involves creating a procedure that simulates your system I/O operation.

   a. This procedure (mapioproc) simulates data output from a port and writes the data to the screen. It is called whenever the user writes data to a port (or wport) within the mapped area.

```
*BASE=16T
*DEFINE PROC mapioproc=WRITE PORTDATA        /* Define the debug
                                                procedure */
*MAPIO 0 LENGTH 1K ICE(mapioproc)            /* Map I/O to ICE with a debug
                                                procedure call */
*PORT(23)=1234   /* Execute the debug procedure by accessing the byte port */
34
```

The PORT command accepts only byte inputs.

```
*WPORT(47) = 1cba      /* Execute the debug procedure by accessing the word port */
0BA
1C
```

Data in from WPORT(47) is displayed on two lines because the port is on an odd boundary.

b. This procedure (inputproc) is called whenever a program asks for input from one of the ports mapped to ICE:

```
*DEFINE BYTE b = 0
*DEFINE PROC inputproc = DO
. *IF b = =4 THEN
. . *halt
. . *END
. *PORTDATA = b
. *b = b + 1
. *END
```

c. This program is a loop that requires input from port 12H. It is used to test procedure inputproc:

```
*ASM 0 LENGTH 5
000000H     90          NOP
000001H     E512        IN AX,12H ; +18T
000003H     90          NOP
000004H     EBFA        JMP A=0000H ; $-4
000006H     90          NOP
```

Run the program and halt it when the emulation prompt appears:

```
*GO FROM 0 FOREVER
?HALT
Probe 0 stopped at 0000:0003 because of halt
*AX
4
```

Cross-References

Name
Partition
PROC

Masked constant

A number with don't-care bits
for match conditions in
break and trace controls

Masked constants are used as patterns for matching addresses and data values in break and trace controls. The X bits in the masked constant are don't-care bits; these bits match both 0 and 1 in the address or data value to be matched.

Only binary and hexadecimal masked constants are allowed. All masked constants are stored internally as 32-bit values.

If you omit Y or H for the base of the number, the number is interpreted in the current default base. (An error occurs if the digits are not valid in the current base.)

Examples

1. A binary masked constant that accepts either 0 or 1 for the lower eight bits:

   ```
   01110001XXXXXXXXY
   ```

2. A hexadecimal masked constant that accepts any number (0-F) for the lower byte:

   ```
   0F7XXH
   ```

MEMRDY

A pseudo-variable that allows a system time-out based on memory access time

Syntax

```
MEMRDY [ =TRUE
       | =FALSE
       | =boolean-expression ]
```

Where:

MEMRDY	displays the current setting (TRUE or FALSE).
TRUE	lets a time-out occur when memory access time during emulation exceeds one second. The default value is TRUE.
FALSE	disables memory time-outs.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Discussion

The I²ICE system senses the READY line of the probe processor. A time-out occurs if this line is high for more than one second and MEMRDY is TRUE. A time-out halts emulation.

Example

1. Use MEMRDY as a variable:

```
*DEFINE PROC memrdy = DO
. *IF MEMRDY = = FALSE THEN
. . *MEMRDY = TRUE
. . *END
. *END
```

MENU

Enables and disables the I²ICE system menu display

Syntax

```
MENU [ = TRUE
     | = FALSE
     | = boolean-expression ]
```

Where:

MENU	displays the setting, either TRUE or FALSE.
TRUE	enables the menu display. TRUE is the default.
FALSE	disables the menu display.
boolean-expression	is an expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Discussion

With the MENU command you can enable or disable the I²ICE menu display at the bottom of the terminal screen. The I²ICE menu is a syntax directory on the bottom of the screen. The syntax directory aids in construction of syntactically correct commands. Note, however, that you can construct a syntactically correct command that is semantically incorrect.

When MENU = TRUE, the menu displays all the tokens you can enter at the current cursor position. As you advance the cursor to the next token (with a space or other delimiter), the menu is updated to show the legal tokens at the new position. Often all the available tokens do not fit on one line. In that case, press the TAB key to display the other choices.

If you enter a token not on the current token list, the I²ICE system returns a syntax error.

You must be in command mode to enter the MENU command, but you can change the menu mode at any time by pressing CTRL-V.

Example

1. The following example shows the first menu display after entering the I²ICE command:

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

MENU continued

Pressing the TAB key advances to the following menu display:

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

Mtype

Basic program memory types
used in commands and displays

Syntax (List of Keywords)

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

Default

BYTE

Discussion

The I²ICE command language assumes that every object in memory has a type. Many commands involve explicit references to memory types, and displays also depend on the type of the value. Type BYTE is the default when no type is specified.

This section describes the available memory types. The discussion includes the keywords used to specify types, the formats for displaying various types, and the rules for converting one type to another.

NOTE

Although every object in memory has a type, debug objects cannot be accessed using the memory type keywords.

Types and Type Classes

Table 1-17 lists the program types with their basic definitions. The table also classifies the types by common attributes. The following sections provide more detailed information on mtypes.

Table 1-17 Basic Mtypes

Type	Keyword	Definition
UNSIGNED	BYTE	8-bit unsigned quantity.
	WORD	16-bit unsigned quantity*.
	ADDRESS	16-bit unsigned quantity*.
	SELECTOR	16-bit unsigned quantity*.
	DWORD	32-bit unsigned quantity.
SIGNED	SHORTINT	8-bit signed quantity.
	INTEGER	16-bit signed quantity.
	LONGINT	32-bit signed quantity.
87	REAL	32-bit floating-point number.
	EXTINT	64-bit signed quantity.
	LONGREAL	64-bit floating-point number.
	TEMPREAL	80-bit floating-point number.
	BCD	80-bit packed decimal number.
POINTER	POINTER	32-bit quantity, consisting of a segment selector component and an offset component. Each component is a 16-bit WORD.
BOOLEAN**	BOOLEAN	TRUE (LSB = 1) or FALSE (LSB = 0).
CHARACTER	CHAR	8-bit ASCII value.
	ASM	Assembly language mnemonic; ASM is a read-only type.

* The I²ICE system does not distinguish between these types. The difference is significant only in your program.

** The Boolean type is determined by the least significant bit (LSB) of the byte.

Unsigned Types

The unsigned types are BYTE, WORD, ADDRESS, SELECTOR, and DWORD. ADDRESS and SELECTOR are synonyms for WORD and are included for compatibility with high-level languages that support these types.

Signed Types

The signed types are SHORTINT, INTEGER, and LONGINT. Internally, these types have a leading sign bit; they use the 2's complement for negative values. When signed types are used in expressions, they are converted internally to LONGINT.

87 Types

The 87 types are BCD, EXTINT, REAL, LONGREAL, and TEMPREAL. These types use either the 87 coprocessor or internal I²ICE 87 emulator software. The internal representations of these types are described in the *iAPX-86,88 User's Manual*. The 87 types used in expressions are converted to TEMPREAL.

Pointer Type

Objects of type POINTER are used in address calculations. The method of calculation is processor-specific.

Boolean Type

Boolean objects have one of two values, TRUE or FALSE. When the Boolean mtype is applied to memory, the contents of memory are treated as bytes. If the low-order bit of a byte is a 1, the Boolean value is TRUE. If the low-order bit is a 0, the Boolean value is FALSE.

Character Types

The character types are CHAR and ASM. Type CHAR consists of bytes containing the ASCII representation of characters. Type ASM produces a string of characters, the disassembly of the instructions coded in memory. The format of the disassembly depends on the processor. ASM is a read-only type; you cannot define a variable of type ASM, nor can you assign a new value to an object of type ASM. You can use type ASM as a source value. For example:

```
*DEFINE CHAR HERE = ASM $
```

Displaying Mtypes

Table 1-18 summarizes the formats used to display objects of the various types. Refer to the EXAMPLE section for sample displays of each type.

Table 1-18 Display Formats for Mtypes

Typet	Display
BYTE	Displayed in the current base.
WORD	Displayed in the current base.
DWORD	Displayed in the current base.
ADDRESS	Displayed as a WORD.
SELECTOR	Displayed as a WORD.
SHORTINT	Displayed in decimal, has leading + or −.
INTEGER	Displayed in decimal, has leading + or −.
LONGINT	Displayed in decimal, has leading + or −.
EXTINT	Displayed in decimal, has leading + or −.
REAL	Displayed in scientific format.
LONGREAL	Displayed in scientific format.
TEMPREAL	Displayed in scientific format.
BCD	Displayed in decimal, has leading + or −.
POINTER	Treats the object as a pair of words. Displayed in the form *nnnn:nnnnH* in hexadecimal.
BOOLEAN	Treats the object as a byte. If the low order bit is 0, FALSE is displayed. If the low order bit is 1, TRUE is displayed.
CHAR	Treats the object as an ASCII character string. The display is a quoted string. A non-printing character is indicated by a period (.).
ASM	Displayed as mnemonics (disassembled code) for active processor type.

Type Conversions

The I²ICE system handles conversions between types. Type conversions are necessary when types are combined within an expression and when a value of one type is assigned to a variable of another type. The following paragraphs contain the rules for type conversions.

Type Conversions in Expressions

You can combine objects of two different types using a binary (two-operand) operator such as +, *, or AND. The type of the result depends on the types of the operands and the operator used. Table 1-19 summarizes the valid combinations and results.

The binary operators are grouped into four classes: arithmetic, logical, relational, and pointer. Within each class, Table 1-19 shows what types you can combine with that class of operator and the type of the result. The table also shows the valid unary (one-operand) operations on typed objects and the types of the results.

NOTE

Table 1-19 shows the only valid type combinations within expressions. An error results from any combination not shown in the table.

Table 1-19 Type Conversion by Combination as Operands

Operator	Operands	Result
Arithmetic (+, −, *, /, MOD)	Unsigned and Unsigned	DWORD
	Signed and Unsigned	LONGINT
	Signed and Signed	LONGINT
	87 and Unsigned	TEMPREAL
	87 and Signed	TEMPREAL
	87 and 87	TEMPREAL
	Pointer and Unsigned (+, − only)	POINTER*
	Pointer and Signed (+, − only)	POINTER*
	Pointer and 87 (+, − only)	POINTER*
	Pointer and Pointer (− only)	DWORD
	Character and Unsigned	DWORD**
Logical (AND, OR, XOR)	Unsigned and Unsigned	DWORD
	Signed and Unsigned	LONGINT
	Signed and Signed	LONGINT
	Boolean and Unsigned	BOOLEAN
	Boolean and Signed	BOOLEAN
	Boolean and Boolean	BOOLEAN
	Character and Unsigned	DWORD**

*The operations only affect the offset portion of the pointer.

**You can use only one-character strings with binary operators.

Table 1-19 Type Conversion by Combination as Operands (continued)

Operator	Operands	Result
Relational (= =, > =, < =, >, <, < >)	Unsigned and Unsigned	BOOLEAN
	Signed and Unsigned	BOOLEAN
	Signed and Signed	BOOLEAN
	87 and Unsigned	BOOLEAN
	87 and Signed	BOOLEAN
	87 and 87	BOOLEAN
	Pointer and Unsigned	BOOLEAN
	Pointer and Signed	BOOLEAN
	Pointer and 87	BOOLEAN
	Pointer and Pointer	BOOLEAN
	Boolean and Unsigned	BOOLEAN
	Boolean and Signed	BOOLEAN
	Boolean and 87	BOOLEAN
	Boolean and Boolean	BOOLEAN
	Character and Unsigned	BOOLEAN**
	Character and Character	BOOLEAN
Pointer (:)	Unsigned and Unsigned	POINTER
	Signed and Unsigned	POINTER
	Signed and Signed	POINTER
	87 and Unsigned	POINTER
	87 and Signed	POINTER
	87 and 87	POINTER
	Character and Unsigned	POINTER**
Unary		
+, −	Unsigned	LONGINT
+, −	Signed	LONGINT
+, −	87	TEMPREAL
NOT	Unsigned	DWORD
NOT	Boolean	BOOLEAN
.	Symbolic reference	POINTER

*The operations only affect the offset portion of the pointer.

**You can use only one-character strings with binary operators.

Type Conversion by Assignment

Type conversion also occurs when a value of one type is assigned to an object of a different type. For example:

```
*DEFINE REAL VAR1 = BYTE .byte_variable
*BYTE .TABLE = INTEGER 0100:0000
```

Table 1-20 shows the valid conversions by assignment, including the internal conversion path when applicable. Table 1-20 also indicates invalid combinations.

NOTE

Conversions also occur when a particular type is required by the syntax (e.g., in COUNT *expression, expression* is converted to a word).

Table 1-20 Assignment Type Conversions*

From Type T1	To Type T2		
	Unsigned	**Signed**	**87**
Unsigned (BYTE, WORD, DWORD, ADDRESS, SELECTOR)	Convert to DWORD, truncate to T2	Zero fill to DWORD, truncate to T2	Zero fill to DWORD, convert to TEMPREAL, convert to T2
Signed (SHORTINT, INTEGER, LONGINT	Sign extend to LONGINT, truncate to T2	Sign extend LONGINT, truncate to T2	Convert to TEMPREAL, convert to T2
87 (EXTINT, BCD, REAL, LONGREAL, TEMPREAL; 87 required)	Convert to TEMPREAL, then to EXTINT, truncate to T2	Convert to TEMPREAL, then to EXTINT, truncate to T2	Extend to TEMPREAL, convert to T2
Pointer (POINTER)	Convert pointer to absolute (DWORD), truncate to T2	Convert pointer to absolute (DWORD) Truncate to T2	INVALID
Boolean (BOOLEAN)	INVALID	INVALID	INVALID
Character (CHAR, ASM)	Make first character a byte, zero-fill to T2	INVALID	INVALID

*The value of type T1 is assigned to a variable of type T2 (i.e., *mtype* T2 = *mtype* T1).

Table 1-20 Assignment Type Conversions* (continued)

From Type T1	To Type T2		
	Pointer	**Boolean**	**Character**
Unsigned (BYTE, WORD, DWORD, ADDRESS, SELECTOR)	Zero fill to DWORD, convert to pointer (may be illegal operation with some units, e.g., 286)	IF LSB(T1) = 1 then T2 = TRUE else T2 = FALSE	Convert T2 to one-character string
Signed (SHORTINT, INTEGER, LONGINT	INVALID	IF LSB(T1) = 1 then T2 = TRUE else T2 = FALSE	INVALID
87 (EXTINT, BCD, REAL, LONGREAL, TEMPREAL; 87 coprocessor required)	INVALID	IF LSB(T1) = 1 then T2 = TRUE else T2 = FALSE	INVALID
Pointer (POINTER)	No conversion necessary	INVALID	INVALID
Boolean BOOLEAN	INVALID	No conversion necessary	INVALID
Character (CHAR, ASM)	INVALID	INVALID	No conversion necessary

*The value of type T1 is assigned to a variable of type T2 (i.e., *mtype* T2 = *mtype* T1).

Examples

The following examples list the displays produced by single objects of each type. The displays assume that the contents of the 10 bytes starting at the current execution point ($) are FA, 2E, 8E, 16, 00, 00, BC, 72, 00, and 2E (hexadecimal). The default base for the displays is assumed to be decimal.

1. ***BYTE $**
   0020:0004H 26

2. ***WORD $**
   ```
   0020:012026
   ```
3. ***ADDRESS $**
   ```
   0020:0004H 12026
   ```
4. ***SELECTOR $**
   ```
   0020:0004H 12026
   ```
5. ***DWORD $**
   ```
   0020:0004H 378416890
   ```
6. ***SHORTINT $**
   ```
   0020:0004H -6
   ```
7. ***INTEGER $**
   ```
   0020:0004H +
   ```
8. ***LONGINT $**
   ```
   0020:0004H +378416890
   ```
9. ***EXTINT $**
   ```
   0020:0004H +2674830804922
   ```
10. ***REAL $**
    ```
    0020:0004H +2.297E-25
    ```
11. ***LONGREAL $**
    ```
    0020:0004H +4.779632974980l0E+244
    ```
12. ***TEMPREAL $**
    ```
    0020:0004H +0.128024639217211O3E-1386
    ```
13. ***BCD $**
    ```
    0020:0004H +732200001694356O
    ```
14. ***POINTER $**
    ```
    0020:0004H  168E:2EFAH
    ```
15. ***BOOLEAN $**
    ```
    0020:0004H  FALSE
    ```
16. ***CHAR $**
    ```
    0020:0004H '.'
    ```
17. ***ASM $**
    ```
    0020:0004H FA    CLI
    ```

Mtype continued

Cross-References

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

The 80286 supports multitasking with a built-in, task-switch operation. When a task switch occurs, the 80286 saves the entire execution state in a task state segment, loads a new execution state, and begins executing the new task. Each task has its own virtual address space.

The Task State Segment and Address Protection

The user program switches to another task by transferring execution control to a task-state segment. The new task is called the incoming task; the old task is called the outgoing task. The user program can transfer control in two ways: directly or through a task gate.

When the user program transfers control to a task state segment directly, the task state segment must be at the same or lower (numerically higher) privilege level than the outgoing task; that is, the DPL of the task state segment descriptor (TSSD) must be equal to or greater than the outgoing task's CPL.

When the user program transfers control to a task state segment through a task gate, the task gate must be at the same or lower (numerically higher) privilege than the outgoing task; that is, the DPL of the task gate must be equal to or greater than the outgoing task's CPL. In addition, the task-state segment pointed to by the task gate must be at an equal or higher (numerically lower) privilege level than the task gate; that is, the DPL of the task state segment descriptor must be equal to or less than the DPL of the task gate.

Note that the destination offset field of the task gate is not used in a task switch.

These protection rules are not concerned with the privilege level of the incoming task (i.e., the incoming task's CPL). To determine the incoming task's CPL, look at the CS selector stored in the task state segment. This CS selector points to a CS descriptor the DPL of which is the CPL of the incoming task.

Task Switching

A user program switches tasks by executing a jump (JMP), call (CALL), or software-interrupt (INT) instruction. With the jump and call instructions, the new address points to either a task state segment descriptor or a task gate. With the software interrupt instruction, the interrupt type identifies a task gate in the interrupt descriptor table (IDT).

A task switch may also occur as the result of an external interrupt (an external device asserts INTR or NMI). The interrupt controller supplies an 8-bit vector that, when multiplied by 4, is an offset into the IDT. If the IDT entry is a task gate, a task switch occurs.

The task state segment descriptor (TSSD) contains the base address of the task state segment. The TSSD also identifies the task state segment as available or busy. The task busy flag belongs to the access byte of the TSSD. The TSSDs must reside in the global descriptor table (GDT).

The task gate contains a selector that points to the TSSD. A task gate may reside in the GDT, the LDT, or the IDT.

The task register (TR) contains a selector that points to a TSSD in the GDT. When you load the TR with a selector, the 80286 automatically copies this TSSD into the TR's explicit cache. The TR contains the selector of the currently executing task. The TR's explicit cache contains the TSSD of the currently executing task.

The nested task flag (NFL) resides in the FLAGS word. When 0, NFL indicates that the primary task is executing (i.e., no task switch has occurred). When 1, the NFL indicates that a task switch has occurred.

The task-switch flag (TSF) resides in the machine status word (MSW). When 1, the TSF indicates that a task switch has occurred and that the current processor extension context may belong to a previous task.

The actions of the JMP, CALL, and INT instructions are as follows.

JMP	Saves all registers in the outgoing task state segment. Loads the TR with the selector for the TSSD of the incoming task. Loads all registers from the incoming task state segment. Marks the incoming task state segment busy. Marks the outgoing task state segment not busy. Sets the TSF to 1. Clears the NFL of the incoming task. Leaves the NFL of the outgoing task alone.
CALL	Saves all registers in the outgoing task state segment. Loads TR with the selector for the TSSD of the incoming task. Loads all registers from the incoming task state segment. Leaves the outgoing task state segment marked busy. Marks the incoming task state segment busy. Sets the TSF to 1. Leaves the NFL of the outgoing task alone. Sets the NFL of the incoming task to 1.
INT	Saves all registers in the outgoing task state segment. Loads TR with the selector for the TSSD of the incoming task. Loads all registers from the incoming task state segment. Leaves the outgoing task state segment marked busy. Marks the incoming task state segment busy. Sets the TSF to 1. Leaves the NFL of the outgoing task alone. Sets the NFL of the incoming task to 1.

The user program returns from a task with the IRET instruction. If the nested task flag of the outgoing task is 1, the 80286 performs a task switch return. If the NFL of the outgoing task is 0, the 80286 performs a normal interrupt return. The actions of the IRET instruction is as follows:

IRET	Saves all registers in the outgoing task state segment. Loads TR with the selector for the TSSD of the incoming task. Loads all registers from the incoming task state segment. Marks the outgoing task state segment not busy. Clears the NFL of the outgoing task. Leaves the NFL of the incoming task alone.

Cross-References

80286 registers
TSS

Name

Rules for creating and
using names in commands

Syntax

first-character [*following-character*]*

Where:

first-character	is any alphabetical character (a-z), an underscore (_), a question mark (?), or an at sign (@).
following-character	is any alphabetical character (a-z), an underscore (_), an at sign (@), a dollar sign ($), a question mark (?), or the numbers 0-9. The first 40 characters are significant. The maximum length is 255 characters.

Discussion

Names are either keywords that are predefined by the I²ICE system, symbols created by you in programs, or symbols created by you in I²ICE commands while debugging those programs. All names currently in memory reside in the virtual symbol table. Refer to the *I²ICE™ System User's Guide* for details. With the I²ICE system you can create and change debug object names and manipulate user program symbol names when required.

This section explains how the I²ICE system uses debug object names and user program names.

Creating I²ICE™ System Names

Some I²ICE commands use *name* to label debug objects. When *name* is included in I²ICE command syntax, replace *name* with any alpha-numeric name you want, according to the syntax rules in the preceding Syntax section and the semantic rules that follow. The following I²ICE system naming rules are similar to most high-level language rules.

- Uppercase and lowercase letters are equivalent. Thus, var1 and VAR1 are the same name. The first 40 characters in a name are significant. If a program symbol has more than 40 characters, the extra characters are ignored when the symbol is read in from the object file.
- You can use the dollar sign to break up symbol names. The dollar sign is ignored by the system when it is combined with other letters or numerals. Thus, the I²ICE system recognizes PROCONE and PROC$ONE as the same name. A dollar sign in a name is different from the dollar sign pseudo-variable that signifies the current execution point.
- The underscore is significant. PROCTWO and PROC_TWO are different names.

How the I²ICE™ System Uses Program Names

The I²ICE system uses translator-generated names as symbols. Labels, procedure names, and modules names are all symbols.

In particular, Pascal and FORTRAN compilers use decimal numbers as labels. The I²ICE system appends a leading at sign (@) to Pascal and FORTRAN labels, to convert them to names. Thus if your Pascal or FORTRAN program has a label 12, refer to this label as @12.

How Names Appear in the Symbol Table

When a name is referenced in a command, the system looks up the name in the virtual symbol table to determine whether it is a keyword, a debug object, or program symbol. The tables are searched in order, with keywords first, debug objects next, and program symbols last.

Keywords are predefined and cannot be changed or removed. (See the Keywords entry in this encyclopedia for a list of the I²ICE system keywords.)

You define the names of debug objects (debug variables, literals, debug registers, and debug procedures). The system does not permit a debug object to have the same name as a keyword.

Program symbols are loaded with the program code. No checking is done, so a program symbol may duplicate a keyword or debug object name. The double-quote operator (") forces the system to look in the program symbol table for a reference. Therefore, a symbolic reference must include the double-quote operator when the symbol name duplicates a keyword or debug symbol name. (Refer to the Symbolic reference entry in this encyclopedia for more information on the double-quote operator.)

Cross-References

Keywords
Symbolic References

NAMESCOPE

Displays or sets the current NAMESCOPE for symbolic references

Syntax

NAMESCOPE [= *address*]

Where:

NAMESCOPE	displays the reference address (pointer) that determines the set of visible program objects.
address	changes the reference address. The *address* evaluates to type POINTER.

Discussion

The NAMESCOPE pseudo-variable is a pointer to an address in your program. The I²ICE system uses the NAMESCOPE address as a reference point to determine the amount of qualification required to identify an object in the program.

A fully-qualified reference to a symbol includes the module name and the names of all procedures that enclose the symbol in order from outer-most to inner-most. Since 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. The system looks up a partially qualified reference as follows:

1. The inner-most program block enclosing the NAMESCOPE address is determined, and the symbols defined in that block are checked.
2. If the symbol is not found, the next enclosing block is searched. This procedure is repeated until the symbol is found or until the search fails in the outer-most block, the module enclosing the NAMESCOPE address.

When you load a program, the NAMESCOPE pseudo-variable is set to the address of the prologue of the main module. The NAMESCOPE pseudo-variable changes in three cases:

- When it is set with the NAMESCOPE command.
- When a program is reloaded, the NAMESCOPE pseudo-variable is setto the new main module address.
- When a break is executed, the NAMESCOPE pseudo-variable is changed to the execution point ($).

When you set NAMESCOPE to an mtype value other than POINTER, that value becomes NAMESCOPE's selector; NAMESCOPE's offset becomes zero.

NOTE

The NAMESCOPE command does not work with ASM86.

Examples

The following examples assume that the user program has two modules with enclosed procedures and variables, structured as follows:

```
adder (MODULE)
    operand_count (PROCEDURE)
        number_of_ops (BYTE VARIABLE)
    error_check (PROCEDURE)
        error_number (BYTE VARIABLE)
display_character (MODULE)
    char_check (PROCEDURE)
        char_count (BYTE VARIABLE)
    output_char (PROCEDURE)
```

1. Access the variable number_of_ops from the adder module:

```
*operand_count.number_of_ops
3
```

2. Access the variable char_count in a different module. You need a fully qualified reference, which includes the module name and all enclosing procedure names.

```
*:display_character.char_check.char_count
26
```

3. To allow a short (partially-qualified) reference to char_count, change NAMESCOPE as shown.

```
*NAMESCOPE = :display_character.char_check
*char_count
26
```

4. Display the NAMESCOPE pseudo-variable's current location as a POINTER value. Symbols are displayed if available.

```
*NAMESCOPE
0100:001AH:display_character
```

Cross-References

$
Address

NUMTOSTR

A function that converts an expression into ASCII code

Syntax

NUMTOSTR (*expression*)

Where:

NUMTOSTR	converts *expression* into its ASCII representation. The conversion is displayed on the terminal but does not alter memory. The current number base is used for conversion.

Example

1. Display the variable VAR1 as an ASCII string.

```
*BASE = 10T
*DEFINE CHAR var1 = NUMTOSTR(1.23E-3 * 4.32E+5)
*var1
5.31360000000000000E+2
```

Cross-Reference

Expression

OFFSETOF

A function that returns the offset of a pointer value

Syntax

OFFSETOF (*pointer*)

Where:

pointer	is any program variable, debug variable, function, or expression of mtype POINTER.

Discussion

A pointer contains selector (segment) and offset values used to calculate an address. The OFFSETOF function returns the offset portion of a pointer.

Examples

1. Display the segment offset of the pointer value 200:100:

```
*BASE = 16T
*OFFSETOF(200H:100H)
100
```

2. Display the segment offset of the pointer value for CS:IP:

```
*BASE = 16T
*$
1FC4:345DH
*OFFSETOF ($)
345D
```

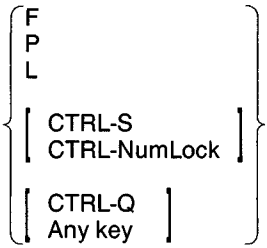
Cross-References

Address
POINTER

Paging

Controls terminal display speed

Syntax



Where:

F	(fast) is the default. New data is written to the screen continuously.
P	(page) writes one screen full of data to the terminal at a time.
L	(line) writes a single line to the terminal at a time.
CTRL-S	(stop) halts terminal display on a Series III host.
CTRL-NumLock	halts the terminal display on an IBM PC host.
CTRL-Q	resumes the terminal display halted by CTRL-S on a Series III host.
Any key	resumes the terminal display halted by CTRL-NumLock on an IBM PC host.

Default

F

Discussion

Some I²ICE commands (such as HELP and WRITE) display more information than will fit on a single screen. With these screen display controls you can halt information before it scrolls off the screen. When a long display is in fast (F) mode, entering P (page mode) halts a full screen of information.

The screen display controls F, P, and L are effective only when the I²ICE system is displaying to the screen. The CTRL-S and CTRL-Q work all the time. If you enter CTRL-S when the I²ICE system expects a command, the I²ICE system will neither accept commands nor echo characters until you enter CTRL-Q.

NOTE

The Series IV host does not recognize CTRL-S and CTRL-Q.

Partition

An address or a range
of addresses

Syntax

```
{ address
  address TO address
  address LENGTH number-of-items }
```

Where:

number-of-items	is a number or an expression that evaluates to a positive integer.

Discussion

A partition is a single address or a range of addresses. Whenever a range of addresses is required instead of a single address, it is specified with either the TO or LENGTH keywords in the form shown in the syntax.

The TO keyword assumes byte addresses. For example:

```
*BYTE 0 TO 9          /*returns 10 bytes starting at address 0:0*/

*WORD 0 TO 9          /*returns 10 bytes in the form of five words*/
```

Note that in the form *address* TO *address* the two addresses must both be either virtual addresses (i.e., aa:bb) or absolute addresses (i.e., nnn). If they are absolute addresses, the first address must be less than the second, and the difference between the two addresses must be no more than 65,536. If they are both virtual addresses, the selector values for both addresses must be equal. If the second address offset is less than the first address offset, the selector boundary wraps around (if wrap-arounds are legal on a particular host).

The LENGTH keyword sets the range depending on the memory type of the *number-of-items*. For example:

```
*BYTE 0 LENGTH 10     /*returns 10 bytes starting at address 0:0*/

*WORD 0 LENGTH 10     /*returns 10 words or 20 bytes starting at address 0:0*/
```

The expression *number-of-items* multiplied by LENGTH (of the type in bytes) must evaluate to 65,536 or less.

NOTE

MAP *partition* adjusts the range to fit the granularity of the map.

Cross-Reference

Address
Expression

Pathname

Specifies the name and location of a file

Syntax

Series III Hosts

{ [:*device*:]*filename*
  :*device*:[*filename*] }

Series IV Hosts

/directory[*/directory*]*/*filename

Where:

device	is a device such as a disk drive, line printer, or teletype output port on which the file *filename* does, or will, reside. Typical values for *device* are as follows: F*n* - disk drive number ($0 \leq n \leq 9$T) LP - line printer TO - teletype output port
filename	is the name of the file. The *filename* can contain up to six alphanumeric characters, plus a three-number extension (e.g., myprog.003).
directory	is the name of a directory.

IBM PC Hosts

[*device*:\][*directory*\]*filename*

Where:

device	is a hard disk or floppy disk drive. Values for *device* are A, B, C, D, etc.
directory	is the name of a directory.
filename	is the name of a file. The *filename* can contain up to eight alphanumeric characters, plus up to a three-number or three-letter extension (such as MYPROG.002).

NOTE

The PC/AT and PC/XT support directory-path searching as defined by the PATH and SET commands. When you use an ISIS pathname (i.e., :Fn:), you must set :Fn: to a PC pathname with the SET command.

Examples

1. The following example opens a list file called 1st.001 on disk drive 1 (on a Series III):

   ```
   *LIST :F1:1st.001
   ```

2. The following example opens a list file called LOG.001 in a directory called ICEDIR on disk drive A.

   ```
   *LIST A:\ICEDIR\LOG.001
   ```

3. The following example echoes the terminal display to a line printer:

   ```
   *LIST :LP:
   ```

PCHECK
80286 probe specific

Pseudo-variable that requests
I²ICE system protection checking

Syntax

```
PCHECK [ =TRUE
       | =FALSE
       | =boolean-expression ]
```

Where:

PCHECK	displays the current setting (TRUE or FALSE).
TRUE	indicates that I²ICE commands can display and alter only those parts of the prototype system that would normally be accessible when using an 80286 probe in protected mode.
FALSE	indicates that the 80286 probe ignores protection rules.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default Value

TRUE

Discussion

The PCHECK pseudo-variable determines whether the I²ICE system operates with I²ICE protection checking on or off. It affects how I²ICE commands access registers and memory locations.

Memory and register access are also affected by the SEL286 pseudo-variable and the MSW. The SEL286 pseudo-variable determines whether the I²ICE system performs 8086 or 80286 address translation. The PEF in the machine status word (MSW) determines whether the 80286 is in real or protected mode.

Accessing Registers

How I²ICE commands can access 80286 registers depends on the setting of PCHECK and whether the 80286 is in real mode (PEF = 0) or protected mode (PEF = 1).

Real Mode, PCHECK = TRUE

The I²ICE commands display and alter the 8086 registers and the MSW. The REGS command does not display the LDTR, the GDTR, the TR, the IDTR, or the explicit caches of any of the registers.

Real Mode, PCHECK = FALSE

The I²ICE commands display and alter the 80286 registers. The REGS command displays only the 8086 registers and the MSW. Loading the selector part of a register can, however, change its explicit cache. You can modify the MSW and the explicit caches directly.

Protected Mode, PCHECK = TRUE

The I²ICE commands display and alter the 80286 registers. The REGS command displays the 80286 registers but does not display the explicit caches.

Modifying the selector part of a register causes the 80286 probe to automatically modify that register's explicit cache. For example, if you change the selector value in the LDTR, the 80286 probe automatically loads the new LDT descriptor into the LDTR's explicit cache. You cannot change the MSW or the explicit caches directly.

The I²ICE system performs validity checking of register assignments. For example, you cannot load the task register with a selector whose table indicator bit does not select the global descriptor table and point to a task state segment descriptor. Nor can you load a segment register with a descriptor that does not point to a segment descriptor.

Protected Mode, PCHECK = FALSE

The I²ICE commands display and alter the 80286 registers. The REGS command displays the 80286 registers along with their explicit caches.

Modifying the selector part of a register causes the 80286 probe to automatically modify that register's explicit cache. The I²ICE system does not perform validity checking of register assignments. Table 1-21 summarizes how the PCHECK pseudo-variable affects register access.

Table 1-21 Effects of the PCHECK Pseudo-Variable

R Displayed by the REGS command.
A Displayed and altered by name using the REGS command.
O Displayed by name only.
S Modifying this register also modifies its explicit cache.

Name	Length (Bytes)	Real Mode PCHECK = TRUE	Real Mode PCHECK = FALSE	Protected Mode PCHECK = TRUE	Protected Mode PCHECK = FALSE
AX	2	RA	RA	RARA	
AL	1	A	A	A	A
AH	1	A	A	A	A
BX	2	RA	RA	RA	RA
BL	1	A	A	A	A
BH	1	A	A	A	A
CX	2	RA	RA	RA	RA
CL	1	A	A	A	A
CH	1	A	A	A	A
DX	2	RA	RA	RA	RA
DL	1	A	A	A	A
DH	1	A	A	A	A
SP	2	RA	RA	RA	RA
BP	2	RA	RA	RA	RA
SI	2	RA	RA	RA	RA
DI	2	RA	RA	RA	RA
IP	2	RA	RA	RA	RA
FLAGS	2	RA	RA	RA	RA
FL	1	A	A	A	A
CS	2	RAS	RA	RAS	AS
CSBAS	4		A	O	RA
CSLIM	2		A	O	RA
CSAR	1		A	O	RA
CSSEL	2		A	OR	A
DS	2	RAS	RAS	RAS	AS
DSBAS	4		A	O	RA
DSLIM	2		A	O	RA
DSAR	1		A	O	RA
DSSEL	2		A	O	RA
SS	2	RAS	RAS	RAS	AS
SSBAS	4		A	O	RA
SSLIM	2		A	O	RA
SSAR	1		A	O	RA
SSSEL	2		A	OR	A
ES	2	RAS	RAS	RAS	AS
ESBAS	4		A	O	RA
ESLIM	2		A	O	RA
ESAR	1		A	O	RA

Table 1-21 Effects of the PCHECK Pseudo-Variable (continued)

R Displayed by the REGS command.
A Displayed and altered by name using the REGS command.
O Displayed by name only.
S Modifying this register also modifies its explicit cache.

Name	Length (Bytes)	Real Mode PCHECK = TRUE	Real Mode PCHECK = FALSE	Protected Mode PCHECK = TRUE	Protected Mode PCHECK = FALSE
ESSEL	2		A	O	RA
MSW	2	RA	RA	RO	RA
LDT	2		AS	RAS	AS
LDTSEL	2		A	O	RA
LDTBAS	4		A	O	RA
LDTLIM	2		A	O	RA
LDTAR	1		A	O	RA
GDTBAS	4		A	O	RA
GDTLIM	2		A	OR	A
IDTBAS	4		A	O	RA
IDTLIM	2		A	O	RA
TR	2		AS	RAS	AS
TRSEL	2		A	O	RA
TRBAS	4		A	O	RA
TRLIM	2		A	O	RA
TRAR	1		A	O	RA

Accessing Memory

How I²ICE commands access program memory depends on the setting of the SEL286 and PCHECK pseudo-variables. It does not depend on whether the 80286 probe is in protected or real mode.

Table 1-22 explains the 80286 memory access rules.

Table 1-22 The 80286 Memory Access Rules

SEL286	PCHECK	Memory Access
TRUE	TRUE	Use a virtual address, specifying both the selector and offset. As part of the virtual address, you can also specify the LDT selector (the offset into the GDT).
TRUE	FALSE	Use a 24-bit absolute address or a virtual address. The 80286 probe performs 80286 address translation. The virtual address must contain the selector and the offset. It may also contain the LDT selector (the offset into the GDT).
FALSE	TRUE FALSE	Use a 24-bit absolute address or a virtual address. The virtual address contains the selector and the offset. The 80286 probe performs 8086 address translation. Because SEL286 is FALSE, the I²ICE system ignores the LDT selector.

Cross-References

80286 registers
Address protection
SEL286

PHANG

8086/8088 probe specific

A pseudo-variable that enables or disables system time-out based on coprocessor activity

Syntax

PHANG [= TRUE | = FALSE | = *boolean-expression*]

Where:

PHANG	displays the setting of the coprocessor time-out function.
TRUE	reports a time-out if a coprocessor memory access exceeds one second.
FALSE	prevents coprocessor time-outs.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default Value

TRUE

Discussion

If enabled, the PHANG (coprocessor hang) command reports when memory accesses made by either the internal or external coprocessor exceed one second. The I²ICE system displays a message when time-outs occur. Enter the PHANG command at the beginning of each emulation session; it remains in affect until changed or reset.

If a hang occurs while the coprocessor has the bus, you must reset the I²ICE probe processor to regain control. Reset the I²ICE probe processor by manually resetting any external coprocessors first, then reset the I²ICE probe processor by entering the I²ICE RESET UNIT command.

Examples

1. Display the current setting:

```
*PHANG
TRUE
```

2. Change the current setting:

```
*PHANG = FALSE
```

PHANG

80186/80188 probe specific

A pseudo-variable that enables or disables system time-out based on coprocessor activity

Syntax

```
PHANG [ =TRUE
      | =FALSE
      | =boolean-expression ]
```

Where:

PHANG	displays the setting of the coprocessor time-out function.
TRUE	reports a time-out if a coprocessor memory access exceeds one second.
FALSE	prevents coprocessor time-outs.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default

TRUE

Discussion

If enabled, the PHANG (coprocessor hang) command reports when memory accesses made by the external coprocessor exceed one second. The I²ICE system displays a message when time-outs occur. Enter the PHANG command at the beginning of each emulation session; it remains in affect until changed or reset.

When the coprocessor takes control of the bus because of a hang, you must reset the I²ICE probe processor to regain control. Reset the I²ICE probe processor by manually resetting any external coprocessors before you enter the I²ICE RESET UNIT command. If the RSTEN command is enabled, the prototype resets the probe processor (but not any external coprocessors).

Example

1. Display the current setting:

```
*PHANG
TRUE
```

2. Change the current setting:

```
*PHANG = FALSE
```

PINS

Displays the state of selected microprocessor signals

Syntax

PINS [*unit-number*[,*unit-number*]* | ALL]

Where:

unit-number	is the number of the unit for which you want the pin states displayed (0, 1, 2, or 3) or an expression that evaluates to 0, 1, 2, or 3.
ALL	displays the pin states for all units.

Discussion

The PINS command displays the state of signal lines on the current probe. Each display varies according to the processor type and mode.

Note that the PINS command displays only the current status of the lines. It does not show pending non-maskable interrupts (NMIs), interrupts (INTRs), or requests/grants (RQs/GTs) if any of these pulsed signals have occurred. The PINS command shows when a signal is in a perpetually errant state (i.e., a shorted signal).

PINS continued

Tables 1-23, 1-24, and 1-25 show the values displayed by the PINS commands for the 8086/8088 probe, the 80186/80188 probe, and the 80286 probe, respectively.

Table 1-23 Values Displayed by the PINS Command for the 8086/8088 Probe

Pin	Definition
HLDACK	Acknowledges receiving the HOLD signal.
HOLD	Indicates that another master is requesting a local bus HOLD.
INTR	Is an interrupt request.
MN/MX/	Indicates minimum (1) or maximum (0) mode.
NMI	Is a non-maskable interrupt.
READY	Is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer.
RESET	Causes the processor to immediately terminate its present activity.
RQ/GT0/, RQ/GT1/	Are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle.
TEST/	Is examined by the wait-for-test instruction.

Table 1-24 Values Displayed by the PINS Command for the 80186/80188 Probe

Pin	Definition
AREADY	Is asynchronous READY.
HOLD	Indicates that another master is requesting a local bus HOLD.
INTR0, INTR1, INTR2, INTR3	Is an interrupt request.
NMI	Is a non-maskable interrupt.
RESET/	Causes the processor to immediately terminate its present activity.
SREADY	Is synchronous ready.
TEST/	Is examined by the wait-for-test instruction.

Table 1-25 Values Displayed by the PINS Command for the 80286 Probe

Pin	Definition
RST	User reset (active high).
NMI	Non-maskable interrupt (active high).
RDY/	Ready (active low).
ERR/	Error (active low).
BSY/	Busy (active low).
INT	Interrupt (active high).
PREQ	Processor extension request (active high).
HOLD	Hold (active high).

Examples

1. Display the current value of the pins using the 8086/8088 I²ICE probe in MIN mode:

```
*PINS
----  PINS FOR UNIT 00  ----
RESET=1  NMI=0    HOLD=0   HLDACK=0
TEST=0   READY=0  MN/MX=1  INTR=1
```

2. Display the current value of the pins using the 8086/8088 I²ICE probe in MAX mode:

```
*PINS
----  PINS FOR UNIT 00  ----
RESET=1  NMI=0    RQ/GT0=1  RQ/GT1=1
TEST=1   READY=1  MN/MX=0   INTR=0
```

3. Display the current value of the pins using the 80186/80188 I²ICE probe:

```
*PINS
----  PINS FOR UNIT 00  ----
TEST/=1  NMI=0   AREADY=1  SREADY=0  RESET/=0
INT0=0   INT1=0  HOLD=0
INT2=0   INT3=0
```

The slash (/) indicates that the preceding signal is active low.

4. Display the current value of the pins using the 80286 I²ICE probe:

```
*PINS
----  PINS FOR UNIT 0000  ----
RST=0 NMI=0 RDY/=0 ER/=1 BSY/=1 INT=0 PREQ=1 HOLD=0
```

POINTER
8086/8088 probe specific
80186/80188 probe specific

Displays or changes memory
as selector:offset
address pointers

Syntax

POINTER *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

POINTER *partition*	displays the memory location described by *partition* as a hexadecimal pointer.
partition	is a single address or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to type POINTER.
mtype	is any of the memory types except ASM.

Discussion

The POINTER command interprets the contents of memory as selector:offset values, overriding any type associated with the memory contents. Selector and offset are 16 unsigned bits each. Thus, POINTER .var1 displays the pointer value formed from the four bytes that begin at the address of var1, regardless of the type of var1.

When a pointer value is assigned, the system assumes that the least significant digit is the offset and the four digits adjacent to it are the code segment. Thus, a maximum of five digits are valid. However, when a pointer type is assigned to data that already exists in memory, all digits are valid.

Example

1. Display a single value:

```
*POINTER $
0020:0004H 168E:2EFAH
```

2. Display several adjacent values:

```
POINTER $ LENGTH 5
020:0006H 168E:2EFAH 16BC:0000H 1E8E:2E00H 0CEA:0002H EF00:2100H
```

3. Assign a single value:

```
*POINTER 40:04H = 12345678
*POINTER 40:04H
0040:0004H 4567:0008
```

4. Assign several adjacent values:

```
*POINTER 40:04H = 123, 2, 12345, 123456
*POINTER 0040:004H LENGTH 4
0040:0004H 0012:0003H 0000:0002H 1234:0005H 2345:0006H
```

Cross-Reference

Expression
Mtype
Partition

POINTER
80286 probe specific

Displays or changes memory as selector:offset address pointers

Syntax

POINTER *partition*
DEFINE POINTER *debug-variable-name = address*

Where:

POINTER *partition*	displays the specified partition in program memory as selector:offset pairs.
DEFINE POINTER *debug-variable-name = address*	defines an I²ICE debug variable whose memory type is POINTER.

Discussion

For the 80286 probe, a debug variable of mtype POINTER is a selector:offset pair or an LDT selector:selector:offset triplet. Program memory always displays as selector:offset pairs. Debug variables can be defined as either selector:offset pairs or LDT selector:selector:offset triplets.

Displaying Program Memory

The mtype POINTER reads memory as follows:

Address + 0	Least significant byte of the offset
Address + 1	Most significant byte of the offset
Address + 2	Least significant byte of the selector
Address + 3	Most significant byte of the selector

Defining a Debug Variable

For the 80286 probe, you can define a debug variable of mtype POINTER as one of the following:

- An absolute address - the absolute address can be up to 24 bits long.
- A selector:offset pair - this is a virtual address. The selector and offset each can be up to 16 bits long.

- An LDT selector:selector:pair triplet - each member of the triplet can be up to 16 bits long. The LDT selector provides an offset into the global descriptor table. It overrides the current selector stored in the LDT register.

Example

1. Define a debug variable called *begin*.

```
*DEFINE POINTER begin = 30:200:6
*begin
0030:0200:0006H
```

Cross-References

Address
Address translation
Mtype
Name
Partition

PORT

A pseudo-variable that displays
or changes the contents
of byte-wide I/O ports

Syntax

PORT(*port-number*) [= *data*]

Where:

PORT(*port-number*)	displays the contents of the user port specified by *port-number* in the current base. The *port-number* is a number or an expression that evaluates to a number in the current base, ranging from 0000H to 0FFFFH.
data	is any byte of data entered in the current base. Using this option writes the data to the specified I/O port.

Discussion

The PORT pseudo-variable displays the contents of I/O ports only when I/O ports are mapped to USER; PORT does not display I/O port contents when I/O ports are mapped to ICE. (The MAPIO command sets the location of I/O ports.)

If you try to write data longer than one byte (e.g., a word) using the PORT pseudo-variable, only the last byte is used. Use the WPORT command for word-length data.

Examples

1. Read an I/O port:

```
*PORT(2)
UNIT 0 PORT 2H REQUESTS BYTE INPUT (ENTER VALUE) :
```

2. Write to an I/O port:

```
*PORT(2) = 50T
UNIT 0 PORT 2H OUTPUT BYTE 32H
```

To read that port:

```
*PORT(2)
35
```

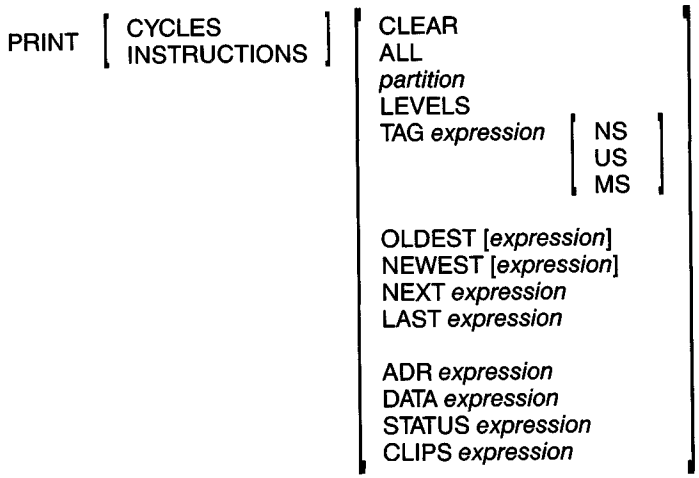
Cross-References

Expression
WPORT

PRINT

Formats and displays the contents of the trace buffer.

Syntax



Where:

PRINT	displays the next element in the trace buffer. The default display mode is INSTRUCTIONS.
CYCLES	displays the trace buffer in bus cycles. Trace buffer display is processor specific. Once set, CYCLES display mode remains in effect until you enter the PRINT INSTRUCTIONS command.
INSTRUCTIONS	displays the trace buffer in disassembled mnemonics. Trace buffer display is processor specific. INSTRUCTIONS is the default display mode. It remains in effect until you enter the PRINT CYCLES command. If the SYMBOLIC pseudo-variable is set to TRUE (the default value), then the display will include symbolic information.

CLEAR

erases the trace buffer. You can also clear the trace buffer by entering PRINT CYCLES CLEAR or PRINT INSTRUCTIONS CLEAR or by issuing a GO command with new break or trace information. A prompt is displayed if you try to print a cleared buffer.

ALL

displays the entire current contents of the trace buffer in the current default mode (CYCLES or INSTRUCTIONS). The trace buffer has 1023 usable frames. The oldest frame is number zero.

partition

specifies the range of frames to be printed. The syntax for *partition* is as follows:

[*start-frame* TO *end-frame*
start-frame LENGTH *number-of-frames*]

Refer to the Partition entry in this encyclopedia for details.

LEVELS

displays the frame numbers at which time marker (timetag) discontinuities occur. The newest trace data are at level 0. When a discontinuity occurs, the level number is changed. This indicates that timing information is no longer accurate. Refer to the TIMEBASE entry in this encyclopedia for more information on discontinuity.

TAG *expression*

searches the trace buffer for the tag value specified by *expression* and displays it in the trace buffer. Tag is a time marker (timetag) in the trace buffer. Tags are displayed in the TIME field of the CYCLES display. The next nearest tag value is displayed if the tag specified is not found.

You can specify tag in NS (nanoseconds), US (microseconds), or MS (milliseconds).

OLDEST/NEWEST
expression

displays the trace buffer from oldest to newest. The trace buffer is numbered from 0-1023 frames. The oldest frame is 0; the newest frame is 1023 (when the buffer is full). One frame is displayed if you do not specify *expression*.

PRINT continued

NEXT/LAST *expression*	displays the next or last *expression* frames relative to the current trace buffer pointer. The trace buffer pointer always points to the last frame displayed. If you did not enter a PRINT command since the last break, last is the newest frame.
expression	is the number of frames, in the current base, you want to display. One frame is displayed if you do not specify *expression*.
ADR/DATA/STATUS/CLIPS *expression*	searches the trace buffer for the item (e.g., ADR) with the value specified by *expression*. The search wraps around the trace buffer up to the current frame position. If found, the item is displayed. If not found, an error message is displayed. The ADR (address) option searches both the execution address and bus address frames.

Discussion

Trace data is stored in the trace buffer. Control trace collection with the trace register commands, GO, and TRCBUS, each discussed in this encyclopedia. The information in the trace buffer determines the amount and type of information displayed. The PRINT command displays the trace buffer of the current unit.

When emulating with trace on, the trace buffer is filled with frame numbers, addresses, processor status, clips information, and timetags. The PRINT command additionally generates disassembled instructions, symbolic program information (e.g., line numbers), level numbers, and the unit number. (Symbolic information is collected in the trace buffer if the SYMBOLIC pseudo-variable is set to TRUE. TRUE is SYMBOLIC's default value.) Display the contents of the trace buffer by using the PRINT command.

You can concatenate trace buffer data. Append new trace data by emulating with the trace buffer on, halting, and resuming emulation using the same break criteria. Concatenation prevents overwriting trace data so you can compare related emulation results. Changing the break criteria between emulations clears the trace buffer.

The trace buffer is displayed in either of two formats: INSTRUCTIONS or CYCLES. You can compare trace data in the two formats by frame number and timetag.

Displaying Trace in Instructions Mode

The INSTRUCTIONS mode displays the frame number, execution address, disassembled instruction, mnemonic, symbols, and unit number.

Two trace collection commands affect the INSTRUCTIONS trace data. Setting TRCBUS true (the default value) packs 1023 frames of execution address information into the trace buffer. Setting SYMBOLIC true (the default value) includes your source code symbols in the trace buffer.

The ADR column contains interspersed bus or execution addresses. Up to four bus cycles per line are displayed in the following format:

Bus address - Access code - Data

The two-character access code represents the origin of the I²ICE trace data. The first character represents the access type, and the second character represents processor activities.

Occasionally, an extra character appears in the trace display. An M (for memory) may appear before the instruction address. When an M appears, it means that the I²ICE system could not disassemble an instruction from the contents of the trace buffer. Instead, the contents of memory are used that may have been changed since trace was collected. Additionally, a question mark (?) may appear in the mnemonic column. When a question mark appears, it means that there is no mnemonic equivalent for the contents of that location.

Displaying Trace in Cycles Mode

The CYCLES mode displays the execution address, bus address, bus data, processor status, clip information, frame number, timetag, level, and unit number. In this display, the execution and bus addresses are separate columns. Bus addresses, bus data, and status always appear on the same line. They represent the current probe's processor bus activity. The execution-addresses line also contains clips, timetag, and level information. The clips column is blank when the input clips are not connected.

Bus status is a 16-bit hexadecimal code followed by the two-character access code. The access codes are probe-specific and are defined in the Trace buffer display entries in this encyclopedia. When displaying collected trace, only the least significant eight bits are used.

The I²ICE system has a free-running counter that counts to 2048 before wrapping around to 0. (Use the TIMEBASE pseudo-variable discussed in the TIMEBASE entry in this encyclopedia to set the time increment.) When the I²ICE system starts tracing, the value of the counter goes to the trace buffer, and the TIME column displays execution time. Frame 0 always starts at time 0.0.

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. Note that you may have lost time calibration because you do not know how many wrap-arounds occurred while the trace was interrupted.

Examples

1. The following example shows a sample 8086/8088 probe trace buffer displayed in INSTRUCTIONS mode.

```
*PRINT INSTRUCTIONS
FRAME   ADR    BYTE          MNEMONICS  OPERANDS                UNIT 0
001 000204H    FA                 CLI
002 000205H    2E8E160000         MOV SS,CS:WORD PTR 0000H
008 00020AH    BC7200             MOV SP,0072H ;+1141
00A 00020DH    2E8E1E2EBC         MOV DS,CS:WORD PTR 0BC2EH
010 000212H    EA00012100         JMP 0021H:0100H
015 000310H    8BEC               MOV BP,SP
017 000312H    FB                 STI
018 000313H    2E8D460800         LEA AX,CS:WORD PTR 0008H
01D 000318H    0E                 PUSH CS
020 000319H    50                 PUSH AX
    000390H-SW-0021H
023 00031AH    9A34003A00         CALL 003AH:0034H
    00038EH-SW-0008H   0003BCH-SW-0021H
028 0003D4H    1E                 PUSH DS
    00038AH-SW-010FH
02B 0003D5H    55                 PUSH BP
    000388H-SW-0032H
02E 0003D6H    8BEC               MOV SP,SP
    000386H-SW-0072H
030 0003D8H    8ED E0AH           MOV DS,[BP+0AH]
    000390H-SR-0021H
035 0003DBH    8B5E0B             MOV BX,[BP+0BH]
    00038EH-SR-0008H
038 0003DEH    BF0000           MOV DI,0
03A 0003E1H    BACE00             MOV DX,00CEH ;+2061
```

2. The following example shows a sample 8086/8088 probe trace buffer in unit 0 displayed in CYCLES mode.

```
*PRINT CYCLES
EXEC ADR    BUS ADR     DATA    STATUS   CLIPS FRAME  TIME LEVEL   UNIT 0
x           b 000202  d C38B s 0054 F   c     f 000
x 000202    b         d       s          c 02  f 001  0.0 nanosecs 0
x           b 000204  d FAE2 s 0054 F   c     f 002
x 000204    b         d       s          c 02  f 003  0.8 microsecs 0
x           b 000206  d F8EB s 0054 F   c     f 004
x           b 000208  d E001 s 0054 F   c     f 005
x           b 00020D  d C103 s 0054 F   c     f 006
x 00020D    b         d       s          c 83  f 007  4.4 microsecs 0
x           b 000202  d C38B s 0054 F   c     f 008
```

Cross-References

Expression
Partition
SYMBOLIC
TIMEBASE
Trace buffer display

PROC

Defines, displays, or executes
a debug procedure

Syntax

```
[DEFINE] PROC debug-procedure-name [DO
                    I²ICE commands *
                END]
```

Where:

PROC *debug-procedure-name*	displays the definition of the named procedure.
DEFINE PROC *debug-procedure-name* DO *I²ICE commands* END	defines a debug procedure.

Discussion

The following sections explain how to use debug procedures.

Defining and Executing Procedures

With procedures you can use several commands in a block structure and declare local variables. However, the procedure can be several nested blocks. The only limit on the size of procedures is the amount of memory space available.

Although a debug procedure is not executed until its name is invoked, the I²ICE system checks the syntax when the procedure is defined and determines the types of all referenced objects. Changing the type or definition of an object in the procedure before it is executed can cause errors when the procedure is executed.

Procedures can be defined within other procedures. The inner procedure is not visible to the I²ICE system until the outer procedure is executed. Once procedures become visible to the system, they are always global, even when nested inside other procedures.

NOTE

You must define debug objects before they can be referenced by a debug procedure.

Delete debug procedures with the REMOVE command.

Returning Values from Procedures

Use the RETURN function to return procedure values. The syntax of the RETURN function is as follows:

RETURN [*expression*]

An error occurs when a procedure expects a return value and does not receive one. The *expression* must be a Boolean value or an expression that evaluates to a Boolean value. Omitting *expression* halts execution of that procedure after the RETURN.

Passing Parameters in Procedures

Use the percent sign (%) in the PROC definition to tell the I²ICE system that you will furnish parameters when you invoke the debug procedure.

%*NP*	A predefined system parameter equal to the number of parameters passed in the debug procedure.
%*number*	A parameter *number* that selects that parameter from the list following the debug procedure invocation. Numbers range consecutively from 0 to 99.
%(*expression*)	Used instead of *number* but requires parentheses. Must evaluate to a number between 0 and 99.

Examples

1. Define and execute a simple procedure that averages three parameters.

```
*DEFINE PROC average = DO RETURN ((%0 + %1 + %2)/%NP) END
*average (10T,2,3)
5
```

PROC continued

2. Define, display, and execute a more complex averaging procedure. Data is supplied by the parameter list.

```
*DEFINE PROC aver = DO                          /* Define the debug procedure */
. *DEFINE INTEGER sum = 0
. *DEFINE BYTE I = 0                            /* Initialize variables */
. *COUNT %NP                  /* Count is equal to the total number of parameters */
. . *sum = sum + %(I)                           /* Add I to the sum */
. . *I = I + 1                                  /* Increment I */
. . *ENDCOUNT
. *RETURN sum / %NP                /* Return the average of all the parameters */
. *END
*
*PROC aver                                      /* Display the debug procedure definition*/
define proc AVER=do
define integer SUM=0
define byte I=0
count %NP
SUM=SUM+%(I)
I=I+1
endcount
return SUM / %NP
end
*AVER (4,5,21T)                                /* Execute the debug procedure */
+10
*AVER (-1,5)
+2
*DEFINE INTEGER i = aver (2,4,7,9)
*I                                              /* Average integers */
+5
*DEFINE REAL r = aver (1,2,3,4)
*R          /* Average real numbers. The result is truncated because SUM is an integer */
+2
```

Cross-Reference

Name

Pseudo-variable

A system-defined variable

Pseudo-variables are a cross between commands and variables. Like commands, pseudo-variables initiate operations. For instance, ports are not only displayed with the PORT pseudo-variable but also read or written. Like variables, pseudo-variables are named, have a value, can be assigned and displayed, and can be used in expressions, as shown in the following command line using the RSTEN pseudo-variable:

```
*IF NOT RSTEN THEN HALT
```

Pseudo-variables are predefined by the I²ICE system and cannot be removed. Their value range is also predefined and can only be changed within that range.

The I²ICE pseudo-variables are the following:

$
BASE
BTHRDY (probe specific)
BUSACT
COENAB (probe specific)
COREQ (80286 probe specific)
CPMODE (probe specific)
CURX
CURY
ERROR
GRANULARITY (80286 probe specific)
IORDY
MEMRDY
NAMESCOPE
PCHECK (80286 probe specific)
PHANG (8086/8088 and 80186/80188 probe specific)
PORT
QSTAT (80186/80188 probe specific)
RSTEN
SCTR
SEL286 (80286 probe specific)
SYMBOLIC
TIMEBASE
TRCBUS
UNIT
WAITSTATE
WPORT
XCTR

Pseudo-variable continued

The 8086/8088 flags and registers, the 8087 registers, the 80186/80188 flags and registers, the 80286 flags and registers, and the 80287 registers are all pseudo-variables and are listed in their respective entries in this encyclopedia.

PSTEP

Single-steps through user programs
by high-level language instructions

Syntax

PSTEP [*increment*] [FROM *address*]

Where:

PSTEP	executes by numbered high-level language statements.
increment	is an unsigned integer expression in the current base specifying the number of steps to take. The default increment is 1. The maximum increment value is 65,535T.
FROM *address*	specifies a starting address where PSTEP is to begin. The default start address is the current execution point ($). (The Address entry in this encyclopedia contains more information on addresses.)

Discussion

The PSTEP command single-steps through user programs by high-level language statements. The PSTEP command executes the next consecutive statement and halts. If the next consecutive statement is a direct call to a procedure, the PSTEP command treats the procedure call and the statements in the called procedure as a single instruction. The PSTEP command handles indirect calls as multiple instructions.

Break messages are not displayed. Use the CAUSE command to display break messages.

After PSTEP executes a line, it displays a message of the following form:

[:*module-name#line-number*]

NOTE

When you use any of the probes, stepping through an instruction that alters a segment register executes two instructions.

When you use the 8086/8088 probe, the instruction being single-stepped must not access locations 4 through 0BH. Stepping through a POPF or IRET instructions may clear the trap flag (TF) if the instruction is programmed that way. To enable single-stepping without clearing the TF, define the event register and procedure, as shown in the following example. Because PSTEP uses the hardware break facility, it may slide through an instruction.

```
*DEFINE EVTREG hstep = DO
. *XEM S0 ALWAYS BREAK END
*DEFINE PROC S = GO USING hstep
```

Any NMIs are ignored when you step using the 8086/8088 probe.

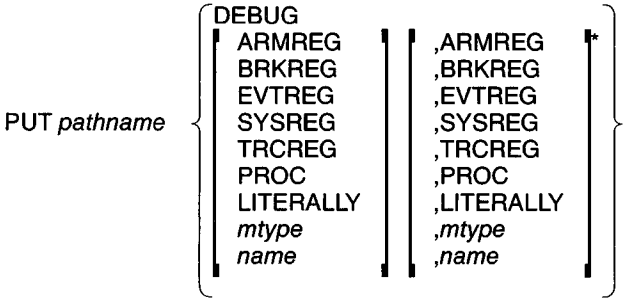
Cross-References

Address
Expression
ISTEP
LSTEP

PUT

Creates and saves system file contents from memory to file

Syntax



Where:

pathname	is the fully-qualified reference to the file into which you want to save debug objects. See the *pathname* entry in this encyclopedia.
mtype	is one of the memory types defined in the Mtype entry in this encyclopedia.
name	is the name of a debug object to be created and saved.

Discussion

The PUT command saves the definitions of debug procedures, LITERALLY definitions, debug memory types, and debug registers to a disk file. The values of debug memory types are not saved.

The PUT command creates a file to which it saves debug definitions. When the named file exists, the question "Overwrite existing file? (y or [n])" is displayed.

By using the DEBUG option, all debug objects are saved. If you specify ARMREG, BRKREG, EVTREG, SYSREG, TRCREG, PROC, or LITERALLY, the PUT command saves all debug objects of that type. If you use just the name of a debug object, only that one is saved.

A PUT file can reside on any suitable random-access device.

NOTE

Do not repeat keywords in the command. For example, the following PUT command is incorrect:

```
PUT :f1:deb.001 PROC,PROC
```

Examples

1. Create and PUT debug objects to an existing file. (If you have an IBM PC host, disregard the symbol ":f2:"; assume the file is in your current hard disk directory. To PUT the file, you would use the command: PUT debug.inc)

```
*DEFINE DWORD s_factor                    /* Create debug objects */
*DEFINE DWORD r_factor
*PUT :f2:debug.inc s_factor, r_factor
```

   Another way to save s_factor and r_factor is as follows:

```
*DEFINE DWORD s_factor
*DEFINE DWORD r_factor
*PUT :f2:debug.inc DWORD
```

2. Restore and list the debug objects from the file:

```
*INCLUDE :f2:debug.inc
*define byte I
*define byte J
*define byte K
*define integer SUM
*define pointer P
*define word X_VALUE
*define literally BASE_ADDR = 'BYTE 1000H:0H'
*define proc WHERE = EVAL $ LINE
*define dword S_FACTOR
*define dword R_FACTOR
```

Cross-References

ARMREG
BRKREG
EVTREG
LITERALLY
Mtype
Name
Pathname
PROC
SYSREG
TRCREG

QSTAT
80186/80188 probe specific

A pseudo-variable that selects
the probe configuration mode

Syntax

```
QSTAT [ =TRUE
      | =FALSE
      | =boolean-expression ]
```

Where:

QSTAT	displays the current setting.
TRUE	selects the queue status signal line configuration: QS0, QS1, and QSMD.
FALSE	selects the standard signal line configuration: ALE, WR, and RD.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default Value

FALSE

Discussion

The QSTAT command determines which signal set the 80186/80188 I²ICE probe emulates. The QSTAT configuration remains in affect until you change it with the QSTAT command.

NOTE

If your microprocessor is configured for queue status signal line, you must set the QSTAT command to TRUE before using the I²ICE system.

Displays or changes memory as 32-bit floating-point values

Syntax

REAL *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

REAL *partition*	displays the contents of memory specified by *partition* as a real number in scientific notation.
partition	is a single address, an expression that evaluates to a single address, or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to a 32-bit floating-point value for REAL.
mtype	is any of the memory types except ASM.

Discussion

The REAL command interprets the contents of memory as 32-bit floating-point values, overriding any type associated with the memory contents. Thus, REAL .var1 displays the 32-bit floating-point decimal value that begins at the address of var1, regardless of the type of var1.

Examples

In the following examples, the I²ICE system responses to the commands are shown in decimal because all real numbers are displayed in decimal, regardless of the base of the input information.

1. Display a single value:

```
*REAL $
0020:0006H +2.29701E-25
```

2. Display several adjacent values:

```
*REAL $ LENGTH 4
0020:0006H +2.29710E-25 +3.03730E-25 +1.50539E -20 +3.60534E-31
```

3. Set a single value of type REAL:

```
*REAL 40H:04H = 1234567t
```

4. Set several adjacent values:

```
*REAL 40H:04H = 123456789t, 123t, -9000.00
```

Display the values set:

```
*REAL 40H:04H LENGTH 3
0040:0004H 1.23457E+8    +1.23000E+2   -9.00000E+3
```

5. Set a range of locations to the same value:

```
*REAL 40H:04H LENGTH 10 = 0
```

6. Set a repeating sequence of values:

```
*REAL 40H:04H LENGTH 10 = 5.678, -2300, 23456, -7.567
```

7. Copy a value from one memory location to another:

```
*REAL 40H:04H = REAL $
```

8. Copy several values (block move):

```
*REAL 40H:04H = REAL $ LENGTH 10
```

9. Copy values with type conversion:

```
*REAL 40H:04H = INTEGER .var2
```

An error message is displayed if the type on the right side of the equal sign cannot be converted to the type on the left. (Refer to the Expression entry in this encyclopedia for the rules concerning type conversions.)

Cross-References

Expression
Mtype
Partition

8086/8088 Registers

8086/8088 probe specific

Displays or modifies 8086/8088 register values

Syntax

8086/8088-register [= *expression*]

Where:

8086/8088-register — displays the current value of the 8086/8088 register and is one of the keywords in Table 1-26.

expression — is an expression of the correct data type used to set a register value.

Table 1-26 8086/8088 Register Keywords

Register	Keyword	Description	Data Type
Data Registers	AX	Accumulator register pair	WORD
	AH	Accumulator high byte	BYTE
	AL	Accumulator low byte	BYTE
	BX	B register pair	WORD
	BH	B register high byte	BYTE
	BL	B register low byte	BYTE
	CX	C register pair	WORD
	CH	C register high byte	BYTE
	CL	C register low byte	BYTE
	DX	D register pair	WORD
	DH	D register high byte	BYTE
	DL	D register low byte	BYTE
Pointer/Index Registers	SI	Source index	WORD
	DI	Destination index	WORD
	BP	Base pointer	WORD
	SP	Stack pointer	WORD
Segment Registers	CS	Code segment	WORD
	DS	Data segment	WORD
	ES	Extra segment	WORD
	SS	Stack segment	WORD
Instruction Pointer	IP	Instruction pointer	WORD

Discussion

Use the 8086/8088 register keywords to display or change register values. You can display registers singly (using the keywords listed in Table 1-26) or in groups (using the REGS command). All registers are displayed in the current radix.

Example

Display and change 8086/8088 registers:

```
*REGS
----  REGISTERS FOR PROBE 0000  ----
AX=0004          BX=063A          CX=0000          DX=0002
CS=5588          DS=0188          SS=0104          ES=0000
IP=46C7          BP=0634          SP=0624          SI=0830
DI=03A2
FLAGS : ZFL PFL
*AH
00H
*CS
5588H
*IP =46C9
*IP
46C9H
```

Cross-Reference

Expression

8087 Registers

Displays or modifies
8087 register values

Syntax

8087-register [= *expression*]

Where:

8087-register — displays the current value of the register and is one of the keywords in Table 1-27.

expression — is an expression of the correct data type used to set a register value.

Table 1-27 8087 Register Keywords

8087 Register Keyword	Description	Data Type
ST0	Internal stack register 0	TEMPREAL
ST1	Internal stack register 1	TEMPREAL
ST2	Internal stack register 2	TEMPREAL
ST3	Internal stack register 3	TEMPREAL
ST4	Internal stack register 4	TEMPREAL
ST5	Internal stack register 5	TEMPREAL
ST6	Internal stack register 6	TEMPREAL
ST7	Internal stack register 7	TEMPREAL
FSW	Status word	WORD
FCW	Control word	WORD
FIA	Instruction address	DWORD
FDA	Data address	DWORD
FIO	Instruction	WORD
FTW	Tag Word	WORD

Discussion

The 8087 register keywords display or change register values. Entering any keyword alone displays the current value of that register.

NOTE

Coprocessor registers can be displayed or modified only when the coprocessor is in mode 2 (refer to the CPMODE entry in this encyclopedia) and the probe is not emulating. You must enter the GET87 command before accessing coprocessor registers.

Examples

1. Display the ST4 register:

```
*ST4
+2.3596874320856382E+00001
```

2. Change the ST0 register:

```
*ST0 = 1.04E2
```

3. Change the data address:

```
*FDA = 100034A8H
```

4. Display the instruction opcode in hexadecimal:

```
*BASE = 10H
*FIO
00A
```

Cross-References

CPMODE
Expression

80186/80188 Registers

80186/80188 probe specific

Displays or modifies
80186/80188 register values

Syntax

80186/80188-register [= *expression*]

Where:

80186/80188-register — displays the current value of the 80186/80188 register and is one of the register keywords listed in Table 1-28.

expression — is an expression (of the correct data type) used to set register values.

Table 1-28 80186/80188 Register Keywords

Register	Keyword	Description
Data Registers	AX	Accumulator register pair
	AH	Accumulator high byte
	AL	Accumulator low byte
	BX	B register pair
	BH	B register high byte
	BL	B register low byte
	CX	C register pair
	CH	C register high byte
	CL	C register low byte
	DX	D register pair
	DH	D register high byte
	DL	D register low byte
Pointer/Index Registers	SI	Source index
	DI	Destination index
	BP	Base pointer
	SP	Stack pointer
Segment Registers	CS	Code segment
	DS	Data segment
	ES	Extra segment
	SS	Stack segment
Instruction Pointer	IP	Instruction pointer

Table 1-28 80186/80188 Register Keywords (continued)

Register	Keyword	Description
Internal Register Map Index	CSCTRL	Chip select control register index range = (1-5)
	DMA0	DMA descriptors, channel 0 index range = (1-6)
	DMA1	DMA descriptors, channel 1 index range = (1-6)
	INTRPT	Interrupt controller registers index range = (1-16)
	TIMER0	Timer 0 control registers index range = (1-4)
	TIMER1	Timer 1 control registers index range = (1-4)
	TIMER2	Timer 2 control registers index range = (1-4)
Flags Register	FLAGS	All flags
Relocation Register	RELREG	Relocation register, internal peripheral control

Discussion

Use 80186/80188 register keywords to display or change register values. Entering any register keyword alone displays the current value of the 80186/80188 register.

You can display registers individually (using the keywords listed in Table 1-28) or as a group (using the REGS command). All registers are displayed in the current base.

A write to an unused or read-only internal peripheral control register is not controlled by the 80186/80188 probe. No error message will result, but the write will not occur. Reads from write-only registers will produce unpredictable data.

The internal register map index keywords in Table 1-28 are I²ICE pseudo-variables. They permit direct access to the word at that location on the chip. Each keyword must be followed by the index value in parentheses. The index values range from 1 to the maximum value listed in Table 1-28; all indexes are in the current radix. For example, the following command changes the value of the max count A register for internal timer 1.

```
*TIMER1(2) = 01000H                    /*count to 4K*/
```

The following command changes the value of the lower memory chip select register.

```
*CSCTRL(2) = 27T
```

Figure 1-14 illustrates how the keywords match the internal registers. It also cross-references the 80188/80186 register names listed in the iAPX 186 chip literature.

I²ICE™ System Keyword	80186/80188 Register Name	Offset from RELREG Value (Hexadecimal)	
RELREG	Relocation Register	FE	Relocation Register
	Unused	DC-FC	
DMAI(6)	Control Word	DA	DMA Channel 1 Control Registers
DMAI(5)	Transfer Count	D8	
DMAI(4)	Destination Pointer (upper 4 bits)	D6	
DMAI(3)	Destination Pointer	D4	
DMAI(2)	Source Pointer (upper 4 bits)	D2	
DMAI(1)	Source Pointer	D0	
	Unused	CC-CE	
DMA0(6)	Control Word	CA	DMA Channel 0 Control Registers
DMA0(5)	Transfer Count	C8	
DMA0(4)	Destination Pointer (upper 4 bits)	C6	
DMA0(3)	Destination Pointer	C4	
DMA0(2)	Source Pointer (upper 4 bits)	C2	
DMA0(1)	Source Pointer	C0	
	Unused	AA-BE	
CSCTRL(5)	MPCS Register	A8	Chip Select Control Registers
CSCTRL(4)	MMCS Register	A6	
CSCTRL(3)	PACS Register	A4	
CSCTRL(2)	LMCS Register	A2	
CSCTRL(1)	UMCS Register	A0	
	Unused	68-9E	
TIMER2(4)	Mode Control Word	66	TIMER2 Control Registers
TIMER2(3)	Unused	64	
TIMER2(2)	Maximum Count A	62	
TIMER2(1)	Count Register	60	
TIMER1(4)	Mode Control Word	5E	TIMER1 Control Registers
TIMER1(3)	Maximum Count B	5C	
TIMER1(2)	Maximum Count A	5A	
TIMER1(1)	Count Register	58	
TIMER0(4)	Mode Control Word	56	TIMER0 Control Registers
TIMER0(3)	Maximum Count B	54	
TIMER0(2)	Maximum Count A	52	
TIMER0(1)	Count Register	50	
	Unused	40-4E	
INTRPT(10)	Interrupt 3 Control Register	3E	
INTRPT(F)	Interrupt 2 Control Register	3C	
INTRPT(E)	Interrupt 1 Control Register	3A	
INTRPT(D)	Interrupt 0 Control Register	38	
INTRPT(C)	DMA 1 Control Register	36	
INTRPT(B)	DMA 0 Control Register	34	
INTRPT(A)	Timer Control Register	32	
INTRPT(9)	Interrupt Controller Status Register	30	
INTRPT(8)	Interrupt Request Register	2E	
INTRPT(7)	In-Service Register	2C	
INTRPT(6)	Priority Mask Register	2A	
INTRPT(5)	Mask Register	28	
INTRPT(4)	Poll Status Register	26	
INTRPT(3)	Poll Register	24	
INTRPT(2)	EOI Register	22	
INTRPT(1)	Unused	20	

Interrupt Control Registers (non-iRMX™ Mode)

INTRPT(10)	Unused	3E
INTRPT(F)	Unused	3C
INTRPT(E)	Level 5 Control Register (TIMER2)	3A
INTRPT(D)	Level 4 Control Register (TIMER 1)	38
INTRPT(C)	Level 3 Control Register (DMA1)	36
INTRPT(B)	Level 2 Control Register (DMA0)	34
INTRPT(A)	Level 0 Control Register (TIMER 0)	32
INTRPT(9)	Unused	30
INTRPT(8)	Interrupt Request Register	2E
INTRPT(7)	In-Service Register	2C
INTRPT(6)	Priority-Level Mask Register	2A
INTRPT(5)	Mask Register	28
INTRPT(4)	Unused	26
INTRPT(3)	Unused	24
INTRPT(2)	Specific EOI Register	22
INTRPT(1)	Interrupt Vector Register	20

Interrupt Control Registers (iRMX™ Mode)

1356

Figure 1-14 80186/80188 Internal Register Map to I²ICE™ System Keyword Cross-reference

Examples

1. Display the 80186/80188 probe registers.

```
*BASE = 16T
*REGS
----  REGISTERS FOR UNIT 00  ----
AX=0004         BX=063A         CX=0000         DX=0002
CS=5588         DS=0188         SS=0104         ES=0000
IP=46C7         BP=0634         SP=0624         SI=0830
DI=03A2         RELREG=20FF
FLAGS : ZFL PFL
*RELREG = 30FF                          /*Move internal registers to I/O space*/
*CS
5588H
*IP = 46C9
*IP
46C9H
```

80286 Registers
80286 probe specific

Displays or modifies 80286 registers

Syntax

80286-register[= *expression*]

Where:

80286-register — displays the current value of the 80286 register and is one of the register keywords given in Table 1-29.

expression — is an expression (of the correct data type) used to set an 80286 register value.

Table 1-29 The 80286 Registers

80286 Register Keyword	Description	I²ICE™ System Memory Type
AX	Accumulator register pair	WORD
AH	Accumulator high byte	BYTE
AL	Accumulator low byte	BYTE
BX	B register pair	WORD
BH	B register high byte	BYTE
BL	B register low byte	BYTE
CX	C register pair	WORD
CH	C register high byte	BYTE
CL	C register low byte	BYTE
DX	D register pair	WORD
DH	D register high byte	BYTE
DL	D register low byte	BYTE
CS	Code segment register	WORD
CSBAS*	Code segment register, base	DWORD
CSLIM*	Code segment register, limit	WORD
CSAR*	Code segment register, access rights	BYTE
CSSEL*	Code segment register, selector	WORD
DS	Data segment register	WORD
DSBAS*	Data segment register, base	DWORD
DSLIM*	Data segment register, limit	WORD
DSAR*	Data segment register, access rights	BYTE
DSSEL*	Data segment register, selector	WORD

*Displayed only when PCHECK = FALSE

Table 1-29 The 80286 Registers (continued)

80286 Register Keyword	Description	I²ICE™ System Memory Type
ES	Extra segment register	WORD
ESBAS*	Extra segment register, base	DWORD
ESLIM*	Extra segment register, limit	WORD
ESAR*	Extra segment register, access rights	BYTE
ESSEL*	Extra segment register, selector	WORD
SS	Stack segment register	WORD
SSBAS*	Stack segment register, base	DWORD
SSLIM*	Stack segment register, limit	WORD
SSAR*	Stack segment register, access rights	BYTE
SSSEL*	Stack segment register, selector	WORD
GDTBAS*	Global descriptor table, base	DWORD
GDTLIM*	Global descriptor table, limit	WORD
IDTBAS*	Interrupt descriptor table, base	DWORD
IDTLIM*	Interrupt descriptor table, limit	WORD
LDTBAS*	Local descriptor table register, base	DWORD
LDTLIM*	Local descriptor table register, limit	WORD
LDTAR*	Local descriptor table register, access rights	BYTE
LDTSEL*	Local descriptor table register, selector	WORD
TR*	Task register	WORD
TRBAS*	Task register, base	DWORD
TRLIM*	Task regsiter, limit	WORD
TRAR*	Task register, access rights	BYTE
TRSEL*	Task register, selector	WORD
FLAGS	Flags register (see Flags entry).	WORD
MSW	Machine status word (see Flags entry).	WORD
BP	Base pointer	WORD
SP	Stack pointer	WORD
IP	Instruction pointer	WORD
DI	Destination index	WORD
SI	Source index	WORD

*Displayed only when PCHECK = FALSE

Discussion

The segment registers, the task register, and the local descriptor table register each contains a selector field. The selector field identifies a descriptor (a segment descriptor, a task state segment descriptor, or a local descriptor table descriptor, respectively) and contains the requested privilege level.

Modifying the task register while in interrogation mode may affect the current task state segment.

How PICE commands access the segment registers, the task register, and the descriptor table registers depends on the protection enabled bit in the MSW and the setting of the PCHECK pseudo-variable.

The Selector Field

The segment selector points to a segment descriptor in either the local descriptor table (LDT) or the global descriptor table (GDT). The local descriptor table register (LDTR) selector and the task register (TR) selector each choose a descriptor in the GDT.

Figure 1-15 shows the bit pattern of a selector.

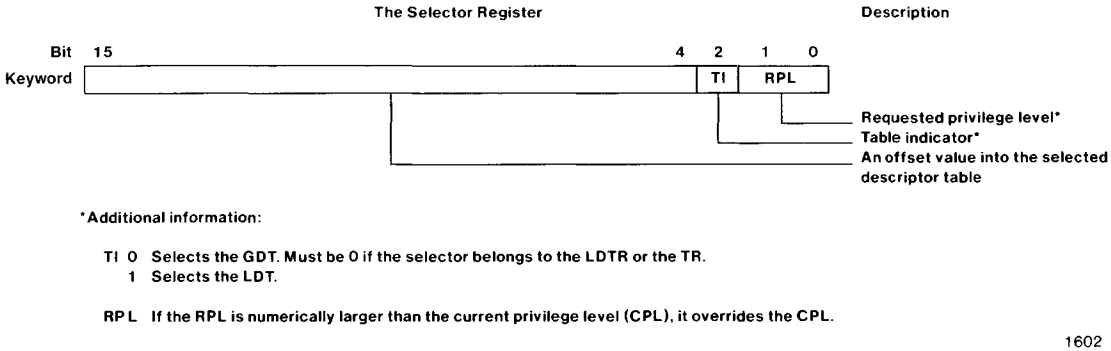


Figure 1-15 Selector Register Bit Pattern

The Task Register

The task register contains the task state segment selector and a copy of the task state segment descriptor. The task state segment selector points to the task state segment descriptor in the global descriptor table.

When a task switch operation occurs, the 80286 saves the outgoing task's state in the outgoing task's task state segment (TSS) and loads the task register with the incoming task's TSS selector and TSS descriptor (TSSD).

Modifying the Task Register with the TR Pseudo-Variable

The TR pseudo-variable represents the selector portion of the task register. Setting TR to a new value while in interrogation mode changes the current TSS. The current task becomes the outgoing task; the new task becomes the incoming task.

However, unlike a task switch operation, modifying the task register with the TR pseudo-variable does not set the task busy flag or the link field in the incoming TSS.

The task busy flag is part of the access field of the TSSD. When you switch tasks with a CALL or INT instruction, the incoming and outgoing task state segment descriptors are marked busy and the task register is loaded with the incoming task state segment selector. The 80286 then automatically updates the TR's explicit cache with the new task state segment descriptor.

The value in TR before the modification points to the outgoing TSSD, which points to the outgoing TSS. The I²ICE system updates the outgoing TSS with the current register values. The value you put into TR points to the incoming TSSD. If an error occurs while setting TR, the TSS and TR remain unchanged.

Figure 1-16 illustrates how the I²ICE system treats the TSS.

80286 Registers, 80286 Protection Mode, and Protection Checking

The protection enabled flag (PEF) in the MSW determines whether the 80286 is in protected mode (PEF = 1) or real mode (PEF = 0).

The setting of the PCHECK pseudo-variable determines whether I²ICE protection checking is on (PCHECK = TRUE) or off (PCHECK = FALSE).

The setting of the SEL286 pseudo-variable determines whether the I²ICE system performs 80286 address translation (SEL286 = TRUE) or 8086 address translation (SEL286 = FALSE).

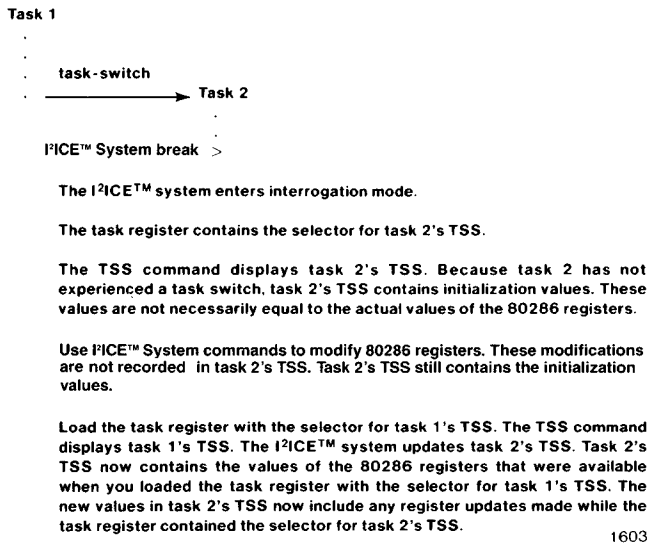


Figure 1-16 Updating the TSS by Changing the TR

The PEF and PCHECK pseudo-variables affect how you access the segment registers, the MSW, the descriptor table registers, and the task register. SEL286 and PCHECK determine how you access 80286 program memory. The results of the various pseudo-variable configurations are as follows:

Real mode, protection checking on:	PEF = 0 SEL286 = FALSE PCHECK = TRUE
Real mode, protection checking off:	PEF = 0 SEL286 = FALSE PCHECK = FALSE
Protected mode, protection checking on:	PEF = 1 SEL286 = TRUE PCHECK = TRUE
Protected mode, protection checking off:	PEF = 1 SEL286 = TRUE PCHECK = FALSE

Accessing the Segment Registers

In real mode with protection checking on, you can display and alter segment selectors. Do this with the pseudo-variable that identifies the register, not the pseudo-variable that identifies the selector field. For example, use DS, not DSSEL. In real mode, the segment base field (part of the explicit cache) tracks the selector field.

In real mode with protection checking off, you can display and alter all segment register fields. The pseudo-variable that identifies the register (for example, DS) operates differently than the pseudo-variable that identifies the selector field (for example, DSSEL). Changing the selector field with DS changes the segment register's explicit cache. In real mode, the segment base field tracks the selector field. Changing the selector field with DSSEL changes only the selector field.

In protected mode with protection checking on, you can display but not alter all segment register fields. You can explicitly alter only the selector field. You must use the pseudo-variable that identifies the register, not the one that identifies the selector field. For example, use DS, not DSSEL. Changing the selector field updates the segment register's explicit cache.

In protected mode with protection checking off, you can display all segment register fields. You can explicitly alter each field. The pseudo-variable that identifies the register (for example, DS) operates differently than the pseudo-variable that identifies the selector field (for example, DSSEL). Changing the selector field with DS automatically updates the segment register's explicit cache. The I²ICE system goes to the appropriate descriptor table and copies the selected segment descriptor. Changing the selector field with DSSEL changes only the selector field.

The Descriptor-Table Registers

The 80286 has three types of descriptor tables: the global descriptor table, the local descriptor table, and the interrupt descriptor table. The descriptor table registers are the GDTR, the LDTR, and the IDTR, respectively.

The I²ICE system treats the LDTR like a segment register. Note, however, that the TI bit in the LDTR selector field must identify the GDT (TI must be 0).

The GDTR and the IDTR do not have selector or access fields. SEL86 and PCHECK determine how you can access these registers.

In real mode with protection checking on, the pseudo-variables identifying the base and limit fields are inaccessible.

In real or protected mode with protection checking off, you can explicitly alter the base, limit, and access fields.

In protected mode with protection checking on, you can only display the base, limit, and access fields.

The Task Register

With regard to the protection rules, the I²ICE system treats the TR like a segment register. Note that in protected mode with protection checking on, the TI bit in the TR selector field must identify the GDT (TI must be 0).

Validity Checking

In real mode, the I²ICE system ignores the protection rules and does not check the validity of the selector assignments.

In protected mode with protection checking on, the I²ICE system checks the validity of the selector assignments.

In protected mode with protection checking off, the I²ICE system does not check the validity of the selector assignments.

Examples

1. Load the task register:

```
*BASE=10H;BASE                    /*Setting the default radix to hexadecimal*/
HEX
*TR=00F0
*TR
00F0
```

2. Display the registers when PCHECK = TRUE and the probes is in real mode:

```
*REGS
----  REGISTERS FOR UNIT 0  ----
AX=0064          BX=0006          CX=0000          DX=00DE
SP=03F2          BP=0007          SI=0311          DI=0030
IP=FFF0
CS=F000          DS=0000          ES=0000          SS=0000
FLAGS : none
MSW : none
```

80286 Registers continued

3. Display the registers when PCHECK = FALSE and the probe is in protected mode:

```
*REGS
----  REGISTERS FOR UNIT 0  ----
GDT=0A28        LDT=1328        IDT=1028        DT=5128
AX=0064         BX=0006         CX=0000         DX=00DE
SP=03F2         BP=0007         SI=0311         DI=0030
IP=0000
CSSEL=0024      CSBAS=000E90    CSLIM=003D      CSAR=9B
DSSEL=0028      DSBAS=001140    DSLIM=03FF      DSAR=93
ESSEL=0038      ESBAS=000E90    ESLIM=003D      ESAR=93
SSSEL=0028      SSBAS=001140    SSLIM=03FF      SSAR=93
LDTSEL=0000     LDTBAS=000D70   LDTLIM=00EF     LDTAR=7F
GDTBAS=000910   GDTLIM=02C7     IDTBAS=000BE0   IDTLIM=018F
TRSEL=0068      TRBAS=0005A0    TRLIM=002B      TRAR=FF
FLAGS :    ZFL PFL NFL
IOPL=00
MSW : MPF PEF
```

Cross-Reference

Address protection
Expression
Flags
PCHECK

80287 Registers
80286 probe specific

Displays or modifies 80287 registers

Syntax

80287-register[= *expression*]

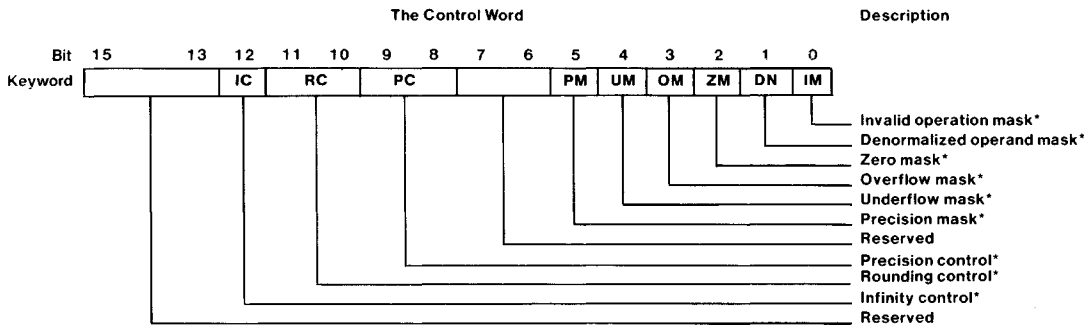
Where:

80287-register — displays the current value of an 80287 register and is one of the keywords listed in Table 1-30. Figure 1-17 shows the bit pattern of the control word. Figure 1-18 shows the bit pattern of the status word. Figure 1-19 shows the bit pattern of the tag word.

expression — is an expression (of the correct data type) that is used to set an 80287 register value.

Table 1-30 The 80287 Registers

80287 Register Keyword	Description	I²ICE™ System Memory Type
ST0	Stack register 0	TEMPREAL
ST1	Stack register 1	TEMPREAL
ST2	Stack register 2	TEMPREAL
ST3	Stack register 3	TEMPREAL
ST4	Stack register 4	TEMPREAL
ST5	Stack register 5	TEMPREAL
ST6	Stack register 6	TEMPREAL
ST7	Stack register 7	TEMPREAL
FCW	Control word	WORD
FSW	Status word	WORD
FTW	Tag word	WORD
Real Mode		
FIA	Instruction address	DWORD
FDA	Data address	DWORD
FIO	Instruction opcode	WORD
Protected Mode		
FIP	Instruction offset	WORD
FCS	Code segment selector	WORD
FDOFF	Data operand offset	WORD
FDSEL	Data operand selector	WORD



*Additional information:

IC 0 = projective
1 = affine

RC 00 = round to nearest or even
01 = round down (toward negative infinity)
10 = round up (toward positive infinity)
11 = chop (truncate toward zero)

PC 00 = 24 bits
01 = reserved
10 = 53 bits
11 = 64 bits

PM When 1, masks a precision exception
UM When 1, masks an underflow exception
OM When 1, masks an overflow exception
ZM When 1, masks a zero-divide exception
DM When 1, masks a denormalized operand exception
IM When 1, masks an invalid operation exception

1604

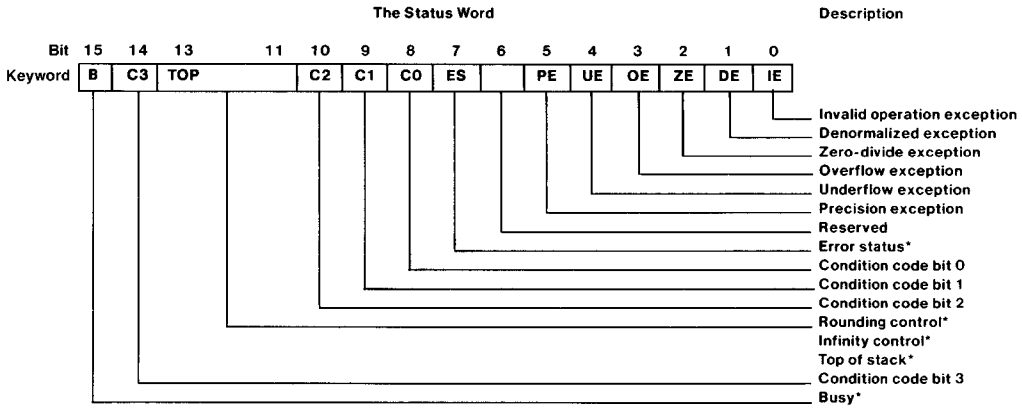
Figure 1-17 The Control Word Bit Pattern

Discussion

The 80287 is a numeric processor extension designed for use with the 80286. It extends the iAPX 286/10 architecture with floating point, extended integer, and binary coded decimal data types and adds over 50 mnemonics to the instruction set. The iAPX 286/20 (80286 with 80287) fully conforms to the proposed IEEE floating-point standard.

The 80287 runs in real or protected mode. It must run in the same mode as the 80286.

When both the 80286 and 80287 are in real mode, the combination is software-compatible with the iAPX 86/20 (8086 with 8087) except for the 8087 interrupt status bit (which is not used in the 80287). In protected mode, all 80287 references to memory for data or status must obey the 80286 memory management and protection rules.



*Additional information:

B 0 = The numeric execution unit (NEU) is idle
1 = The NEU is busy

TOP The 80287 has eight stack registers (ST0 through ST7). The TOP identifies which stack register is currently at the top of the stack. Like 80286 memory stacks, the 80287 register stack grows down. A pop stores the value from the current top register in memory, then increments TOP.

ES 0 = an unmasked exception bit is 0
1 = an unmasked exception bit is 1
The exception bits are bits 5 through 0. The control word determines whether the exception bit is masked.

1605

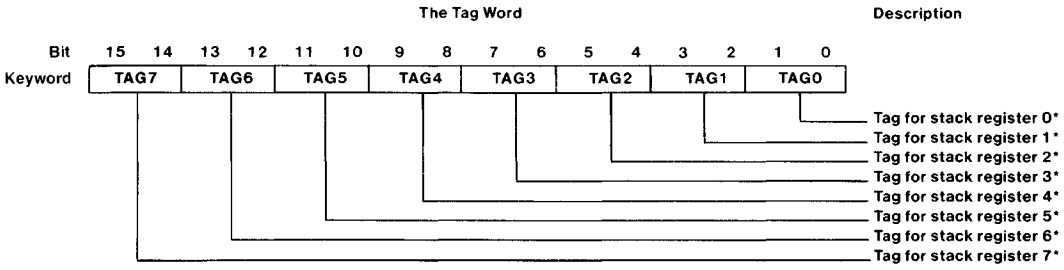
Figure 1-18 The 80287 Status Word Bit Pattern

After a reset, the 80287 is in real mode. It must execute the FSETPM instruction to enter protected mode. Once in protected mode, only a reset can return the 80287 to real mode. This reset must come from the user system.

By using the appropriate pseudo-variable, you can display the 80287 registers in either real or protected format, independent of the current 80287 mode.

Cross-References

COREQ
CPMODE
Expression



*Additional information:

The tag word marks the contents of each stack register.

00 = Valid
01 = Zero
10=Invalid or infinity
11 = Empty

1606

Figure 1-19 The Tag Word Bit Pattern

REGS

Displays selected microprocessor registers and flags set in the current unit

Syntax

```
REGS [ unit-number[,unit-number]* ]
     [ ALL                        ]
```

Where:

unit-number	is the number of the unit for which registers and flags (0, 1, 2, or 3) will be displayed or an expression that evaluates to 0, 1, 2, or 3.
ALL	displays the registers and flags for all emulating units.

Discussion

The REGS command displays the register contents of the current I²ICE probe's microprocessor in the current base. Flags are printed only when they are set. If no flags are set, the word "none" is displayed.

The I²ICE system returns an error if you specify a unit that is emulating.

80286 Probe REGS Command

Which registers the REGS command displays depends on whether the 80286 is in real or protected mode and whether you have enabled or disabled protection checking.

Real Mode

In real mode, the REGS command displays the AX, BX, CX and DX registers, the stack and base pointers, the instruction pointer, the flags register, the segment registers, and the machine status word (MSW). The PCHECK pseudo-variable does not affect this display.

Protected Mode

In protected mode, the PCHECK pseudo-variable affects the display.

When PCHECK is TRUE, the REGS command displays the same registers as in real mode and also includes the selector fields of the LDT and the TR.

REGS continued

When PCHECK is FALSE, the REGS command displays the same registers as in real mode and also includes the local descriptor table register, the global descriptor table register, the interrupt descriptor table register, the task register, and the explicit caches of the segment registers.

Displaying Flags

When a flag in the MSW or the flags register is set, the REGS command displays the flag's mnemonic. If no flags are set, the REGS command displays the word "none".

The I/O privilege level (IOPL) in the flags register appears as a separate entry. The IOPL is a number from 0 to 3, signifying the privilege level needed to execute I/O instructions.

Examples

1. The following example shows the 8086/8088 probe register display. Only the ZFL and PFL flags are displayed because they are the only flags set.

```
*REGS
----  REGISTERS FOR UNIT 0  ----
AX=4          BX=63A          CX=0          DX=2
CS=5588       DS=188          SS=104        ES=0
IP=46C7       BP=634          SP=624        SI=830
DI=3A2
FLAGS : ZFL PFL
```

2. The following example shows the 80186/80188 probe register display. Only the ZFL and PFL flags are displayed because they are the only flags set.

```
*REGS
----  REGISTERS FOR PROBE 00  ----
AX=4          BX=63A          CX=0          DX=2
CS=5588       DS=188          SS=104        ES=0
IP=46C7       BP=634          SP=624        SI=830
DI=3A2        RELREG=20FF
FLAGS : ZFL PFL
```

3. When the 80286 is in real mode and PCHECK is TRUE or FALSE, the 80286 REGS command operates as follows:

```
*REGS
----  REGISTERS FOR UNIT 0000  ----
AX=0064       BX=0006       CX=0000       DX=00DE
SP=03F2       BP=0007       SI=0311       DI=0030
IP=FFF0
CS=F000       DS=0000       ES=0000       SS=0000
FLAGS :    none
MSW :    none
```

4. When the 80286 is in protected mode and PCHECK is TRUE, the 80286 REGS command operates as follows:

```
*REGS
----  REGISTERS FOR UNIT 0000  ----
AX=0064          BX=0006       CX=0000       DX=00DE
SP=03F2          BP=0007       SI=0311       DI=0030
IP=2922
CS=0020          DS=0028       ES=0038       SS=0028
LDT=0000         TR=0068
FLAGS :  ZFL PFL NFL
IOPL=00
MSW :  MPF PEF
```

REGS continued

5. When the 80286 is in protected mode and PCHECK is FALSE, the 80286 REGS command operates as follows:

```
*REGS
----  REGISTERS FOR UNIT 0000  ----
GDT=0A28          LDT=1328          IDT=1028          DT=5128
AX=0064           BX=0006           CX=0000           DX=00DE
SP=03F2           BP=0007           SI=0311           DI=0030
IP=2922
CSSEL=0020        CSBAS=FF8240      CSLIM=551E        CSAR=9B
DSSEL=0028        DSBAS=000140      DSLIM=03FF        DSAR=93
ESSEL=0038        ESBAS=000E90      ESLIM=003D        ESAR=93
SSSEL=0028        SSBAS=000140      SSLIM=03FF        SSAR=93
LDTSEL=0000       LDTBAS=000D70     LDTLIM=00EF       LDTAR=7F
GDTBAS=000910     GDTLIM=02C7       IDTBAS=000BE0     IDTLIM=018F
TRSEL=0068        TRBAS=0005A0      TRLIM=002B        TRAR=FF
FLAGS :    ZFL PFL NFL
IOPL=00
MSW : MPF PEF
```

Cross-References

80286 Flags
80286 Registers
PCHECK

RELEASEIO

Resumes emulation
after the HOLDIO command

Syntax

RELEASEIO

Discussion

Use the RELEASEIO command to resume emulation after using the HOLDIO command.

Example

```
?RELEASEIO
```

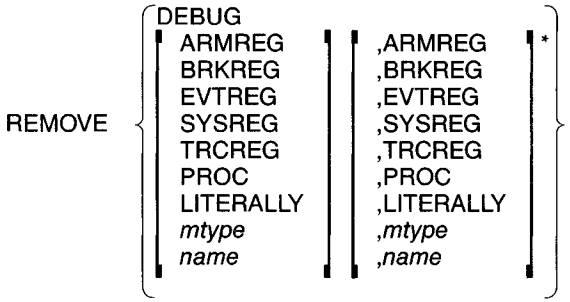
Cross-Reference

HOLDIO

REMOVE

Deletes all user program symbols
or specified debug object definitions

Syntax



Discussion

Use the REMOVE command to delete user program symbols and debug object definitions. Note that if you specify a program symbol such as REMOVE ARMREG, all defined ARMREGs are deleted. To delete a particular ARMREG, specify its name.

Example

1. Remove all user program symbols:

   ```
   *REMOVE SYMBOLS
   ```

2. Remove a single debug procedure definition:

   ```
   *REMOVE X__and__Y          /*X__and__Y is the name of a defined procedure*/
   ```

3. Remove a single debug variable:

   ```
   *REMOVE tempradix  /*tempradix is defined as a single variable within a procedure*/
   ```

4. Remove all debug procedure definitions:

   ```
   *REMOVE PROC
   ```

5. Remove all debug variables of type POINTER:

   ```
   *REMOVE POINTER
   ```

6. Remove all LITERALLY definitions:

```
*REMOVE LITERALLY
```

7. Remove a single LITERALLY definition:

```
*LITERALLY G                          /*Display the LITERALLY definition*/
Define literally G0='g'
*REMOVE g
```

Cross-References

ARMREG
BRKREG
EVTREG
LITERALLY
Mtype
Name
PROC
SYSREG
TRCREG

REPEAT

Groups and executes commands forever or until an exit condition is met

Syntax

```
REPEAT

    [ WHILE boolean-condition ]*
    [ UNTIL boolean-condition ]

    I²ICE commands

END[REPEAT]
```

Where:

REPEAT	executes commands in blocks.
I²ICE commands	executes until the test condition(s) is met. All I²ICE commands are legal except LOAD, EDIT, INCLUDE, and HELP.
WHILE *boolean-condition*	continues to execute while *boolean-condition* is true. Execution halts when the WHILE condition is false.
UNTIL *boolean-condition*	halts execution when the *boolean-condition* is true.
END[REPEAT]	ends the REPEAT block and starts execution. The optional REPEAT keyword is used to label the block type.

Discussion

A REPEAT block is executed immediately after you enter the END statement. A REPEAT block not containing WHILE or UNTIL clauses is executed forever or until aborted with CTRL-C. A REPEAT block containing WHILE or UNTIL exits when any of the test conditions are satisfied.

Example

1. The following example repeats a command:

```
*REPEAT
. *UNTIL $ = = :CMAKER#203
. *ISTEP
. *ENDREPEAT
```

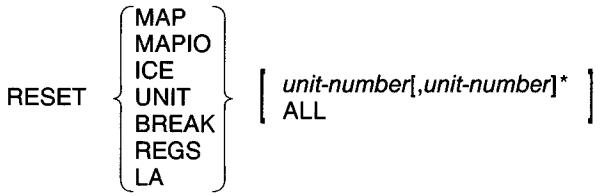
Cross-Reference

Boolean condition

RESET

Reinitializes specified functions of the I²ICE system

Syntax



Where:

MAP	restores the program memory map to its initial GUARDED state (no blocks mapped).
MAPIO	returns all I/O ports to USER status.
ICE	reloads the controlling software in the current probe.
UNIT	activates the RESET pin on the probe processor.
BREAK	clears any break conditions that were set in the current probe unit.
REGS	resets the processor registers to their default values.
LA	resets all the variables which have been set up for the logic analyzer function to their default values.
unit-number	is the number of the unit to be reset (0, 1, 2, or 3) or an expression that evaluates to 0, 1, 2, or 3.
ALL	resets all units to their default conditions.

Cross-Reference

Expression

A pseudo-variable that allows the prototype to reset the probe processor

Syntax

RSTEN [TRUE | =FALSE | =*boolean-expression*]

Where:

RSTEN	displays the current setting (TRUE or FALSE).
TRUE	lets the prototype or other external signal reset the probe processor through the reset pin during emulation.
FALSE	deactivates your connection to the reset pin.
boolean-expression	is any expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default

TRUE

Example

1. Display the current setting:

```
*RSTEN
TRUE
```

2. Disable the user reset:

```
*RSTEN = FALSE
```

3. Use RSTEN as a variable:

```
*IF NOT RSTEN THEN HALT
. *[rest of commands]
. *END
```

SASM

Loads memory with assembled 8086/8088/8087/80186/80188/80286 mnemonics

Syntax

SASM *address* = *'assembler-mnemonic'* [,*'assembler-mnemonic'*]*

Where:

address	is a single address or an expression that evaluates to a single address.
assembler-mnemonic	is an 8086/8088/8087/80186/80188/80286 instruction.

Discussion

The single line assembler (SLA) converts assembler mnemonics to machine code.

Assembler Directives

The SLA does not support assembler directives. For example, you cannot replace *assembler-mnemonic* with MY__VAR DB ?. What you enter for *assembler-mnemonic* must generate code.

Assembler Operators

The SLA does not recognize all the possible assembler operators. The instruction MOV AL,BYTE PTR [BX] is an incorrect form for the SLA because the SLA does not recognize PTR. You can still put that instruction into memory with the SLA, but you must code it as MOV AL,BYTE [BX]. In some cases, a correct form for the SLA is an incorrect form for ASM86.

The following assembler type operators are recognized by the SLA.

BYTE	specifies a number that takes one byte. The corresponding I²ICE memory type is BYTE.
WORD	specifies a 16-bit unsigned number. The corresponding I²ICE memory type is WORD.
DWORD	specifies a number that takes four bytes. The corresponding I²ICE memory type is POINTER.

QWORD	specifies a number that takes eight bytes. The corresponding I²ICE memory type is LONGREAL.
TBYTE	specifies a number that takes 10 bytes. The corresponding I²ICE memory type is TEMPREAL.
FAR	specifies that both the CS and IP take part in a JMP or CALL.
segment override prefixes	specifies that an operand is to be taken from a non-default segment (CS:, DS:, ES:, SS:).

Jumps and Calls

The SLA's control transfer instructions (jumps and calls) have different mnemonic conventions than ASM86. This section discusses the five kinds of jumps and calls: direct-short, direct-near, indirect-near, direct-far, and indirect-far.

Direct-short Jumps and Calls

The SLA does not produce a direct-short jump or a direct-short call; instead, use a direct-near jump or a direct-near call, respectively.

Direct-near Jumps and Calls

The direct-near jump and the direct-near call consist of three bytes. The first byte is E9, the opcode. The next two bytes are the difference between the current location and the destination.

The SLA uses an absolute address as the operand for a direct-near jump and a direct-near call. For example, to load absolute address 100H with a direct-near instruction that jumps to 105H, enter the following:

```
*SASM 100H = 'JMP 105H'
000100      E90200
```

This instruction skips two bytes so the relative displacement from the IP is 0002. To load absolute address 100H with a direct-near instruction that jumps to absolute address 00FCH, enter the following:

```
*SASM 100H = 'JMP 0FCH'
000100      E9F9FF
```

The relative displacement from the IP is FFF9H, which is −7 in 2's complement notation.

Indirect-near Jumps and Indirect-near Calls

The indirect-near jump and the indirect-near call consist of two bytes and possibly a 16-bit displacement. The first byte is the opcode FF, and the second byte contains the MOD field, the R/M field, and three more bits of the opcode (100Y). For example, to load absolute address 200H with an instruction that jumps to the offset contained in BX, enter the following:

```
*SASM 200H = 'JMP BX'
000200H     FFE3
```

You can get another level of indirection by using brackets ([ ]). For example, to load absolute address 200H with an instruction that jumps to the offset stored in the memory location whose offset is in BX, enter the following:

```
*SASM 200H = 'JMP [BX]'
000200H     FF27
```

Direct-far Jumps and Direct-far Calls

The direct-far jump and the direct-far call consist of five bytes. The first byte is the opcode EA, and the last four bytes contain the offset and selector of the target instruction. The SLA recognizes a direct-far jump or direct-far call by the FAR operator. For example, to load location 3:300H with an instruction that jumps to location 12:34, enter the following:

```
*SASM 3:300H = 'JMP FAR 12:34'
0003:0300H  EA34001200
```

If you leave out the selector of the target address, the SLA assumes zero. For example, JMP FAR 34H transfers control to location 00:34.

Indirect-far Jumps and Indirect-far Calls

The indirect-far jump and the indirect-fall call consist of two bytes and possibly a 16-bit displacement. The first byte is the opcode FF. The second byte contains the MOD field, the R/M field, and three more bits of the opcode (101Y). For example, to load offset 400H with an instruction that jumps to the selector and offset stored in the memory location whose offset is in BX, enter the following:

```
*SASM 400H = 'JMP DWORD [BX]'
000400H     FF2F
```

Return-far Jumps and Return-far Calls

ASM86 knows whether a procedure is a near or far return and generates the appropriate return. Because the SLA does not have this information, you must specify a near return as RET and a far return as RETFAR. For example, to load offset 500H with a far return that discards three words from the stack after returning, enter the following:

```
*SASM 500H = 'RETFAR 6'
000500H      CA0300
```

Absolute Addresses

Unlike ASM86, with the SLA you can specify an absolute address within an instruction. For example, the SLA recognizes the instruction JMP 12:34 which is a far-direct jump. ASM86 requires that you use a label or indirect jump through a register.

Like ASM86, the SLA accepts a symbol, but the SLA requires a fully-qualified symbolic reference. For example, to jump to a label within the same module and procedure, enter the module and procedure names in addition to the symbol names (e.g., JMP :mod.proc.label). To load BX with a program variable, enter MOV BX, .:mod.proc.var. The period (.) before the colon (:) is a standard I²ICE operator. It identifies the symbolic reference as resolving to the address of *var* and not the actual value of *var*.

Symbolic Addresses

The SLA accepts symbolic addresses, but, because the SLA does not use the current name scope, you must supply a fully-qualified symbolic reference, such as MOV AX,.:mod.proc-.var.

Indirect Addressing

ASM86 lets you express an indirect address in many ways. For example, the following instructions assemble to the same value.

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

The SLA accepts only the last form. The following format is the general form for an indirect address accepted by the SLA:

symbol[*basereg*][*indexreg*] + *offset*

All the parts are optional. The brackets are part of the syntax and are required. You must use options in the order shown. The following example loads offset 21:3CH with an instruction that moves the contents of the AX register to memory through an indirect address.

```
*SASM 21:3C - 'MOV .:cmaker.purchase[BX][SI] + 300H,AX'
0021:003CH   89801003
```

This instruction loads a memory location with the contents of AX, forming the address of the memory location in the following way:

1. Adds 300H to the offset of the address of the program variable *purchase* (which, as shown in the following example, is 44:10H) in the module *cmaker*.

```
*.:cmaker.purchase
0044:0010H
*DS
0044
```

2. At run time, adds the contents of BX, the contents of SI, and the sum from step 1 to get the final offset.
3. Assumes the data segment, gets the selector value from the DS register, constructs the physical address, and loads the contents of AX into the addressed memory location.

Patching Code with SASM

You can use the SLA to patch user code by replacing an instruction with a jump to patch code in an unused memory area. The final instruction in this area is a jump back to the user program.

For example, assume that the user program reads in a specified number of I/O ports and you want to read additional I/O ports. Also assume that the user program resides within the first 30K bytes of I²ICE high-speed memory and that you want to make a patch at virtual location 0021:0023H to read additional I/O ports.

Use ASM to display the initial user code.

```
*ASM :cmaker#4 to :cmaker#7
:CMAKER#4
0021:0019H    E502          IN AX,2
0021:001BH    A30C00        MOV WORD PTR 000CH,AX
#5
0021:001EH    E502          IN AX,2
0021:0020H    A30A00        MOV WORD PTR 000AH,AX
#6
0021:0023H    8B0E0C00      MOV CX,WORD PTR 000CH
0021:0027H    2BC8          SUB CX,AX
0021:0029H    890E0600      MOV WORD PTR 0006H,CX
#7
0021:002DH    8BC1          MOV AX,CX
```

Insert a jump at location 21:23:

```
*SASM 21:23 = 'JMP FAR 0:31K'
0021:0023    EA007C0000
```

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

```
*SASM 21:28 = 'NOP'
0021:0028H   90
```

Now the initial code is as follows:

```
*ASM :cmaker#4 to :cmaker#7
:CMAKER#4
0021:0019H   E502           IN AX,2
0021:001BH   A30C00MOV      WORD 000CH,AX
#5
0021:001EH   E502           IN AX,2
0021:0020H   A30A00MOV      WORD PTR 000AH,AX
#6
0021:0023H   EA007C000000   JMP 0000H:7C00H
0021:0028H   90             NOP
0021:0029H   890E0600       MOV WORD PTR 0006H,CX
#7
0021:002DH   8BC1           MOV AX,CX
```

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

```
*SASM 31K = 'IN AX,4','MOV WORD 000EH,AX'
007C00H       E504
007C02H       89060E00
*ASM 31K LENGTH 2
007C00H       E504          IN AX,4
007C02H       89060E00      MOV WORD PTR 000EH,AX
```

After adding the desired code, include the instructions you replaced at location 21:23 before jumping back to the initial code.

```
*SASM 31K+6 = 'MOV CX,WORD 000CH','SUB CX,AX','JMP 21:29'
007C06H          8B0E0C00
007C0AH          2BC8
007C0CH          EA29002100
*ASM 31K LENGTH 5
007C00H          E504          IN AX,4
007C02H          89060E00      MOV WORD PTR 000EH,AX
007C06H          8B0E0C00      MOV CX,WORD PTR 000CH
007C0AH          2BC8          SUB CX,AX
007C0CH          EA29002100    JMP 0021H:0029H
```

When the I²ICE system emulates the user program, it jumps to absolute location 007C00H, executes the patch code, then returns to the user code, as shown in the following example.

```
*GO TIL :cmaker#7
?UNIT 0 PORT 0002H REQUESTS WORD INPUT (ENTER VALUE)*1
?UNIT 0 PORT 0002H REQUESTS WORD INPUT (ENTER VALUE)*2
?UNIT 0 PORT 0004H REQUESTS WORD INPUT (ENTER VALUE)*3

*Probe 0 stopped at :CMAKER#7+3H because of execute break Trace
  Buffer Overflow
*PRINT NEWEST 13T
```

```
FRAME   ADR        BYTE    MNEMONICS  OPERANDS          UNIT 0
   3D1 0021:0019H     E502          IN   AX,2
           000002H-CI-0001H
   3D5 0021:001BH     A30C00        MOV  WORD PTR 000CH,AX
   3D8 0021:001EH     E502          IN   AX,2
           0003ECH-DW-0001H  000002H-CI-0002H
   3DC 0021:0020H     A30A00        MOV  WORD PTR 000AH,AX
   3DF 0021:0023H     EA007C0000    JMP  0000H:7C00H
           0003EAH-DW-0002H
   3E3 007C00H        E504          IN   AX,4
           000004H-CI-0003H
   3E7 007C02H        8906E000      MOV  WORD PTR 00EH,AX
   3EB 007C06H        8B0E0C00      MOV  CX,WORD PTR 000CH
           0003EEH-DW-0003H  0003ECH-DR-0001H
   3EF 007C0AH        2BC8          SUB  CX,AX
   3F1 007C0CH        EA29002100    JMP  0021H:0029H
   3F5 0021:0029H     890E0600      MOV  WORD PTR 0006H,CX
   3FA 0021:002DH     8BC1          MOV  AX,CX
           0003E6H-DW-FFFEH
FRAME   ADR        BYTE       MNEMONICS  OPERANDS   UNIT 0
   3FC 0021:002FH     99            CWD
```

The I²ICE system executes this patch in real time. Except for the JMP instructions, the program runs as if the patch code were inserted in the program. If real-time patching is not required for your application, you can implement the patch with a debug procedure.

Multiple Forms of an Instruction

If there is more than one form of an instruction (and there usually is), the SLA assembles the general form and not the shorter form. For example, consider the instruction MOV SUM,AL. ASM86 assembles this in three bytes as A200 01H, assuming that 100H is the offset of the program variable *sum*. The SLA requires a fully qualified symbolic reference for *sum* and assembles the same instruction in four bytes as 8806 0001H.

The Default Number Base

The SLA assumes the current number base, although you can override it by appending a letter to the individual number. The SLA interprets a number as binary if you append a Y, as decimal if you append a T, as hexadecimal if you append an H, and as a multiple of 1024 (decimal) if you append a K.

String Moves

The SLA accepts only the MOVSB and MOVSW mnemonics and not MOVS for string moves.

8087 INSTRUCTIONS

The SLA handles 8087 instructions differently from ASM86 as explained in the following sections.

The Stack Registers

For the SLA, specify the 8087 stack registers as ST0 through ST7 rather than ST(0) through ST(7). ASM86 accepts ST as a symbol for the top of the stack. The SLA does not recognize ST; you must enter ST0.

The ESC Mnemonic

The SLA supports all of the 8087 mnemonics except ESC.

The No-wait Mnemonics

ASM86 inserts a WAIT instruction before an 8087 instruction unless you insert an N as the second character in the 8087 mnemonic. For example, FDISI is preceded by a WAIT; FNDISI is not preceded by a WAIT. The one exception is the 8087 instruction FNOP (a no-operation) that generates a wait.

The SLA, however, is consistent when it interprets the second character of an 8087 mnemonic. The FNOP instruction does not generate a WAIT; FOP does generate a WAIT.

In addition, ASM86 does not allow some 8087 instructions to have the no-wait form. The SLA always accepts a no-wait mnemonic.

FWAIT

The SLA uses the WAIT instruction and not the FWAIT instruction.

Cross-Reference

ASM

SAVE

Saves the memory image currently in mapped memory to a file

Syntax

SAVE *pathname partition*

Where:

SAVE	saves the contents of the specified memory partition to the file specified by the pathname. The memory image is saved in 8086 OMF format. The file can be loaded with the LOAD command.
pathname	is the fully-qualified reference to the file in which you want memory values saved. The file is created if it does not exist; if it already exists, the question "Overwrite existing file? (y or [n])" will be displayed. See the Pathname entry in the *PICE™ System Reference Manual* for information on *pathname*.
partition	is the address or range of addresses in memory that has the memory values that you want saved.

Discussion

When you want to save values in mapped memory addresses, use the SAVE command. The memory image is saved in 8086 OMF format so that it can be reloaded with the LOAD command. (When you load the file, disregard the warning message: "Load module contained no starting address information.")

The SAVE command does not save symbolic information.

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.

Example

1. Save the memory values currently at addresses 18H through 0FF to the file LOAD.FIL (If you have an IBM PC host, disregard the symbol ":f2:". If the file is in your current disk directory, append to the file using the command: SAVE load.fil 18H TO 0FFH. If the file is on another drive, replace :f2: with d:, where d is the letter of the file's disk drive.)

   SAVE :F2:load.fil 18H TO 0FFH

SCTR

A pseudo-variable that assigns a value to the system event machine counter

Syntax

SCTR [= *unsigned-integer-expression*]

Where:

SCTR	displays the value of the system event machine (SEM) counter before emulation. There is no default value; SCTR is random at power on.
unsigned-integer-expression	is a number or expression that evaluates to a positive whole number in the current base.

Default

NONE (random at power on)

Discussion

The SCTR pseudo-variable displays what the value of the SEM counter will be when emulation is initiated. It does not display the current value.

You can set the SEM counter in two ways: by defining the SCTR value in an EVTREG or by using the SCTR command. If you specified a counter value in an EVTREG, executing that EVTREG with the GO command replaces any previously specified SCTR value.

The SCTR pseudo-variable is useful when the counter value needs to be changed for a new emulation or when you forget to specify it in the EVTREG definition. The SCTR command is effective only when used just before invoking an event register specification that does not specify a counter value for SEM.

Example

1. The following example shows how to set a variable SCTR for execution. This EVTREG breaks emulation seven bus cycles after the first occurrence of address 23 on the bus. You can vary this by changing SCTR.

SCTR continued

```
*SCTR = 7
*DEFINE EVTREG count_change = DO
**SEM S0 IF AT 23 THEN INCREMENT AND GOTO S1
**S1 IF ENDCNT THEN BREAK BUT ALWAYS INCREMENT END
*GO USING count_change
Probe 0 stopped at 217 because of system break
```

Cross-References

Event machines
Expression

Determines whether the 80286 probe performs 8086 address translation or 80286 address translation

Syntax

```
SEL286 [ =TRUE
       | =FALSE
       | =boolean-expression ]
```

Where:

SEL286	displays the current setting (TRUE or FALSE).
TRUE	indicates that the 80286 probe performs 80286 address translation.
FALSE	indicates that the 80286 probe performs 8086 address translation.
boolean-expression	is any expression in which the low order bit evaluates to 0 (false) or 1 (true).

Default Value

FALSE

The setting of the SEL286 pseudo-variable is also determined by the last LOAD command. When you load a program file in 8086 OMF and do not specify the SEL286 option, the SEL286 pseudo-variable becomes FALSE. When you load a program file in 8086 OMF and specify the SEL286 option, the SEL286 pseudo-variable becomes TRUE. When you load a file in 80286 OMF, the SEL286 pseudo-variable becomes TRUE.

Discussion

The 8086 address translation consists of shifting the selector field left by four bits and then adding the offset. The result is a 20-bit physical address. With 20 bits, you can address 1M byte of memory.

In 80286 address translation, the selector of the selector:offset pair provides an index into either the global descriptor table or a local descriptor table. The index is multiplied by 8 to become an offset that points to a segment descriptor. This segment descriptor contains an access field, a base address, and a limit field.

The access field identifies the type of descriptor, contains a descriptor privilege level, and identifies whether the addressed segment is in physical memory or stored on some secondary storage device.

The base address points to the base of the addressed segment. The final physical address is the sum of this base address and the offset from the selector:offset pair.

The limit field identifies the number of bytes that make up a segment. A segment can be as large as 64K-bytes.

Example

1. Set the SEL286 pseudo-variable to TRUE:

```
*SEL286 = TRUE
*SEL286
TRUE
```

Cross-References

Address translation
LOAD
Trace buffer display

SELECTOR

Displays or changes memory as 16-bit unsigned values

Syntax

SELECTOR *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

SELECTOR *partition*	displays the contents of memory specified in *partition* as a selector:offset in the current base.
partition	is a single address or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to a 16-bit unsigned value for SELECTOR.
mtype	is any of the memory types except ASM.

Discussion

The SELECTOR command interprets the contents of memory as 16-bit unsigned values, overriding any type associated with the memory contents. Thus, SELECTOR .var1 displays the first word at the address of var1, regardless of the type of var1.

The SELECTOR command displays information identical to that displayed by the WORD and ADDRESS commands. However, when SELECTOR is used as a data type within a program, it is interpreted as the code segment of an address pointer, with the instruction pointer segment assumed to be 0000.

Examples

The following examples use a hexadecimal base.

1. Display a single value:

```
*SELECTOR $
0020:0004H 2EFA
```

2. Display several adjacent values:

```
*SELECTOR $ LENGTH 10
0020:0010H 2EFA 168E 0000 72BC 2E00 1E8E 0002 00EA 2101 0000 0814 0400 0400
0020:001DH 0815 0400 72BC
```

3. Set a single value of type SELECTOR:

```
*SELECTOR 40:4 = 34AF
```

4. Set several contiguous values:

```
*SELECTOR 40:4 = 10FA, 3045, 107F
```

Display the values set:

```
*SELECTOR 40:4 LENGTH 3
0040:0004H 10FA 3045 107F
```

5. Set a range of locations to the same value (block set):

```
*SELECTOR 40:4 LENGTH 10 = 0
```

6. Set a repeating sequence of values:

```
*SELECTOR 40:4 LENGTH 10 = 1234, 5678, 9ABC, 0DEF0
```

Display the values set:

```
*SELECTOR 40:4 LENGTH 10
0040:0004H 1234 5678 9ABC DEF0 1234 5678 9ABC DEF0 1234 5678 9ABC DEF0 1234
0040:0011H 5678 9ABC DEF0
```

7. Copy a value from one memory location to another:

```
*SELECTOR 40:4 = SELECTOR $
```

8. Copy several values (block move):

```
*SELECTOR 40:4 = SELECTOR $ LENGTH 10
```

9. Copy values with type conversion:

```
*SELECTOR 40:4 = BYTE .var2
```

An error message is displayed if the type on the right side of the equal sign cannot be converted to the type on the left. (Refer to the Expression entry in this encyclopedia for the rules concerning type conversions.)

Cross-References

Expression
Mtype
Partition

SELECTOROF

A function that returns
the selector portion
of a pointer

Syntax

SELECTOROF (*pointer*)

Where:

pointer	is any program variable, debug variable, expression, function, or other object of mtype POINTER.

Discussion

A pointer contains selector (segment) and offset values used to calculate an address. The SELECTOROF function returns the selector portion of a pointer.

Examples

1. Display the selector of the address 200:100:

```
*SELECTOROF(200H:100H)
200
```

2. Display the selector of the current execution point ($):

```
*$
1FC4:345DH                    /*Display the current execution address*/
*
*SELECTOROF ($)               /*Display the selector of $ */
1FC4
```

Cross-References

Expression
Mtype
POINTER

SHORTINT

Displays or changes memory as 8-bit signed values

Syntax

SHORTINT *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

SHORTINT *partition*	displays the contents of memory specified in *partition* as a short integer in decimal.
partition	is a single address or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to an 8-bit signed value for SHORTINT.
mtype	is any of the memory types except ASM.

Discussion

The SHORTINT command interprets the contents of memory as 8-bit signed values, overriding any type associated with the memory contents. Thus, SHORTINT .var1 displays the integer that begins at the address of var1, regardless of the type of var1. If the most significant nibble of the unsigned data comprising the INTEGER is 8 through F, the value is interpreted as a negative number and displayed as the 2's complement of the unsigned data.

Note that the I²ICE system always displays values for signed-integer memory types as decimal numbers, regardless of the selected number base.

Examples

The base is hexadecimal in the following examples.

1. Display a single value:

```
*SHORTINT 40:4
0040:0004 +31
```

2. Display several adjacent values:

```
*SHORTINT $ LENGTH 5
0020:0006H -7 +29 -72 +16 + +0
```

3. Set a single value of type SHORTINT:

```
*SHORTINT 40:4 = 25
```

4. Set several adjacent values:

```
*SHORTINT 40:4 = 12, 0AB, 3
```

Display the values set (you can set memory locations to signed integer values using a hexadecimal base, but the I²ICE system displays the values in decimal):

```
*SHORTINT 40:4 LENGTH 3
0040:0004 +12 -55 +03
```

5. Set a range of locations to the same value (block set):

```
*SHORTINT 40:4 length 10 = 0
```

6. Set a repeating sequence of values:

```
*SHORTINT 40:4 LENGTH 10 = 12, 56, 0BC, 0F0
```

7. Copy a value from one memory location to another:

```
*SHORTINT 40:4 = SHORTINT $
```

8. Copy several values (block move):

```
*SHORTINT 40:4 = SHORTINT $ LENGTH 10
```

9. Copy values with type conversion:

```
*SHORTINT 40:4 = BYTE .var2
```

An error message is displayed if the type on the right side of the equal sign cannot be converted to the type on the left. (Refer to the Expression entry in this encyclopedia for the rules concerning type conversions.)

Cross-References

Expression
Mtype
Partition

The following steps outline how to prepare your program for debugging with the I²ICE system.

1. Generate source code.
2. Translate (compile or assemble) the source code. Suitable translators for the 8086/8088 and 80186/80188 I²ICE probes are as follows:

PL/M-86	Version 2.3 or greater
PASCAL-86	Version 2.0 or greater
ASM86	Version 2.0 or greater
FORTRAN-86	Version 2.0 or greater
C	Version 2.1 or greater

NOTE

To produce 186/188 instructions from PL/M-86 or ASM-86 translators, use the MOD 186 control.

When compiling, use the DEBUG control to generate the symbol table. Also use the OPTIMIZE (0) compiler control. With other optimization levels, the compiler may perform cross-statement optimization. The resulting output can be confusing when specifying breakpoints for debugging.

With ASM86, use the TYPE control to get symbolic type information. For Pascal and PL/M, TYPE is the default compiler control.

3. Link all object modules and library routines to resolve references and detect type mismatches and other relocation errors.
4. Locate the linked file (using LOC86) to produce an absolute object module.

Steps 5 through 7 invoke the I²ICE debugger.

5. Invoke the I²ICE software with the I2ICE command (described in this encyclopedia and in the *I²ICE™ System User's Guide*).

6. Map memory and I/O with the MAP and MAPIO commands (described in this encyclopedia).
7. Load the located program into mapped memory with the LOAD command (described in this encyclopedia).

NOTE

The I²ICE system LOAD command does not handle overlays. You cannot use the I²ICE system to load and debug files containing overlays.

The PSCOPE debugger requires load time locatable code (LTL) as input. The I²ICE system requires absolute code as input. Programs configured for PSCOPE will not load under the I²ICE system.

Cross-References

I2ICE
LOAD
MAP
MAPIO

I²ICE™ System User's Guide

Software requirements
80286 probe specific

The following steps outline how to prepare your 80286 program for debugging with the I²ICE system.

1. Generate source code.
2. Translate (compile or assemble) the source code. Suitable translators for the 80286 I²ICE probe are as follows:

BND286	Version 3.0 or greater
BLD286	Version 3.0 or greater
PL/M-286	Version 2.5 or greater
PASCAL-286	Version 3.1 or greater
ASM286	Version 1.1 or greater
FORTRAN-286	Version 3.0 or greater
C-286	Version 3.0 or greater

NOTE

— When compiling, use the DEBUG control to generate the symbol table. Also use the OPTIMIZE (0) compiler control. With other optimization levels, the compiler may perform cross-statement optimization. The resulting output can be confusing when specifying breakpoints for debugging.

— With ASM286, use the TYPE control to get symbolic type information. For Pascal and PL/M, TYPE is the default compiler control.

NOTE

— The I²ICE system does not support Pascal-286 and FORTRAN-286 array size greater than 64K.

3. Bind all object modules and library routines to resolve references and detect type mismatches and other relocation errors.
4. Locate the linked file (using BLD286) to produce an absolute object module.

Software requirements (80286) continued

Steps 5 through 7 invoke the I²ICE debugger.

5. Invoke the I²ICE software with the I2ICE command (described in this encyclopedia and in the *I²ICE™ System User's Guide*).

6. Map memory and I/O with the MAP and MAPIO commands (described in this encyclopedia).

7. Load the located program into mapped memory with the LOAD command (described in this encyclopedia).

NOTE

The I²ICE LOAD command does not handle overlays. You cannot use the I²ICE system to load and debug files containing overlays.

Cross-References

I2ICE
LOAD
MAP
MAPIO

I²ICE™ System User's Guide

STACK

Displays elements from the top of the stack

Syntax

STACK [*expression*]

Where:

STACK	displays one element from the top of the stack.
expression	is a positive number or an expression that evaluates to a positive number that specifies the number of elements to be displayed.

Examples

1. Display one element from the top of the stack:

```
*STACK
003A:0004H 0302
```

2. Display five elements from the top of the stack:

```
*STACK 5
003A:0004H 0302 000E 00A2 0002 0302
```

Cross-Reference

Expression

STATUS

Displays the current setting
of selected debug environment conditions

Syntax

```
STATUS [ unit-number[,unit-number]* ]
       [ ALL                        ]
```

Where:

unit-number	is the number of the unit (0, 1, 2, or 3) for which the status will be displayed or an expression that evaluates to 0, 1, 2, or 3.
ALL	displays the status of all units.

Discussion

The STATUS command displays the settings of certain probe-specific debug variables. The current probe is the source of the display.

USERMODE and PROBETYPE are labels, not variables. PROBETYPE identifies which processor is the source of the displayed information. For the 80286 probe, USERMODE indicates whether the probe is executing in REAL or PROTECTED mode.

Tables 1-31, 1-32, and 1-33 explain the values displayed by the STATUS command for the 8086/8088 probe, the 80186/80188 probe, and the 80286 probe, respectively.

Table 1-31 Values Displayed by the STATUS Command for the 8086/8088 Probe

Value	Description
BTHRDY	When TRUE, the probe's microprocessor recognizes the AND of the READY from wherever memory is mapped and the READY signal from the prototype. When FALSE, the probe's microprocessor takes ready from wherever memory is mapped. (Refer to the BTHRDY entry in this encyclopedia for more information.)
BUSACT	When TRUE, an emulating program times out when bus inactivity exceeds one second. When FALSE, an emulating program does not time out when bus inactivity exceeds one second. (Refer to the BUSACT entry for more information.)
COENAB	When TRUE, the coprocessor is enabled. When FALSE, the coprocessor is disabled. (Refer to the COENAB entry for more information.)
CPMODE	When 1, an external coprocessor runs only when the probe's microprocessor is emulating. When 2, an external coprocessor runs all the time. (Refer to the CPMODE entry for more information.)
IORDY	When TRUE, an emulating program times out when an I/O access time exceeds one second. When FALSE, an emulating program does not time out when an I/O access time exceeds one second. (Refer to the IORDY entry for more information.)
MEMRDY	When TRUE, an emulating program times out when a memory access time exceeds one second. When FALSE, an emulating program does not time out when a memory access time exceeds one second. (Refer to the MEMRDY entry for more information.)
PHANG	When TRUE, an emulating program times out when a coprocessor memory access exceeds one second. When FALSE, an emulating program does not time out when a coprocessor memory access exceeds one second. (Refer to the PHANG entry for more information.)
PROBETYPE	Identifies the probe.
RSTEN	When TRUE, specifies that an external signal can reset the probe processor. RSTEN has meaning only in emulation mode; it has no effect in interrogation mode. When FALSE, specifies that the prototype's connection to the RESET pin has no effect. (Refer to the RSTEN entry for more information.)
TRCBUS	When TRUE, collects both execution and bus information into the trace buffer. When FALSE, collects only execution information into the trace buffer. (Refer to the TRCBUS entry for more information.)

Table 1-32 Values Displayed by the STATUS Command for the 80186/80188 Probe

Value	Description
BTHRDY	When TRUE, the probe's microprocessor recognizes the AND of the READY from wherever memory is mapped and the READY signal from the prototype. When FALSE, the probe's microprocessor takes ready from wherever memory is mapped. (Refer to the BTHRDY entry in this encyclopedia for more information.)
BUSACT	When TRUE, an emulating program times out when bus inactivity exceeds one second. When FALSE, an emulating program does not time out when bus inactivity exceeds one second. (Refer to the BUSACT entry for more information.)
COENAB	When TRUE, the coprocessor is enabled. When FALSE, the coprocessor is disabled. (Refer to the COENAB entry for information.)
CPMODE	When 1, an external coprocessor runs only when the probe's microprocessor is emulating. When 2, an external coprocessor runs all the time. (Refer to the CPMODE entry for information.)
IORDY	When TRUE, an emulating program times out when an I/O access time exceeds one second. When FALSE, an emulating program does not time out when an I/O access time exceeds one second. (Refer to the IORDY entry for more information.)
MEMRDY	When TRUE, an emulating program times out when a memory access time exceeds one second. When FALSE, an emulating program does not time out when a memory access time exceeds one second. (Refer to the MEMRDY entry for more information.)
PHANG	When TRUE, an emulating program times out when a coprocessor memory access exceeds one second. When FALSE, an emulating program does not time out when a coprocessor memory access exceeds one second. (Refer to the PHANG entry for information.)
PROBETYPE	Identifies the probe.
QSTAT	When TRUE, selects the queue status signal line configurations: QS0, QS1, and QSMD. When FALSE, selects the standard signal line configuration: ALE, WR, and RD. (Refer to the QSTAT entry for information.)
RSTEN	When TRUE, specifies that an external signal can reset the probe processor. RSTEN has meaning only in emulation mode; it has no effect in interrogation mode. When FALSE, specifies that the prototype's connection to the RESET pin has no effect. (Refer to the RESET entry for more information.)
TRCBUS	When TRUE, sends both execution and bus information to the trace buffer. When FALSE, collects only execution information into the trace buffer. (Refer to the TRCBUS entry for more information.)

Table 1-33 Values Displayed by the STATUS Command for the 80286 Probe

Value	Description
BTHRDY	When TRUE, the probe's microprocessor recognizes the AND of the READY from wherever memory is mapped and the READY signal from the prototype. When FALSE, the probe's microprocessor takes ready from wherever memory is mapped. (Refer to the BTHRDY entry in this encyclopedia for more information.)
BUSACT	When TRUE, an emulating program times out when bus inactivity exceeds one second. When FALSE, an emulating program does not time out when bus inactivity exceeds one second. (Refer to the BUSACT entry for more information.)
COENAB	When TRUE, the 80286 probe microprocessor recognizes its HOLD and HLDA signals. When FALSE, the 80286 probe microprocessor does not recognize its HOLD and HLDA signals. (Refer to the COENAB entry for more information.)
COREQ	When TRUE, the 80286 probe microprocessor recognizes its PEREQ and PEACK signals. When FALSE, the 80286 probe microprocessor does not recognize its PEREQ and PEACK signals. (Refer to the COREQ entry for more information.)
CPMODE	When 1, an external coprocessor runs only when the probe's microprocessor is emulating. The probe's microprocessor recognizes its PEREQ, PEACK, HOLD, and HLDA lines only during emulation. When 2, an external coprocessor runs all the time. The probe's microprocessor recognizes its PEREQ, PEACK, HOLD, and HLDA lines during both emulation and interrogation. (Refer to the CPMODE entry for more information.)
IORDY	When TRUE, an emulating program times out when an I/O access time exceeds one second. When FALSE, an emulating program does not time out when an I/O access time exceeds one second. (Refer to the IORDY entry for more information.)
MEMRDY	When TRUE, an emulating program times out when a memory access time exceeds one second. When FALSE, an emulating program does not time out when a memory access time exceeds one second. (Refer to MEMRDY entry for more information.)
PCHECK	When TRUE, you can display and alter only those parts of the prototype system that would normally be accessible under the 80286 protection mode. When FALSE, the I²ICE system ignores most of the protection rules. (Refer to the PCHECK entry for more information.)
PROBETYPE	Identifies the probe.
RSTEN	When TRUE, specifies that an external signal can reset the probe processor. Reset enable has meaning only in emulation mode; it has no effect in interrogation mode because system reset is ignored in interrogation mode. When FALSE, specifies that the prototype's connection to the RESET pin has no effect. (Refer to the RSTEN entry for more information.)

Table 1-33 Values Displayed by the STATUS Command for the 80286 Probe (continued)

Value	Description
SEL286	When TRUE, the 80286 probe performs 80286 address translation. When FALSE, the 80286 probe performs 8086 address translation. (Refer to the SEL286 entry for more information.)
TRCBUS	When TRUE, collects both execution and bus information into the trace buffer. When FALSE, collects only execution information into the trace buffer. (Refer to the TRCBUS entry for more information.)
USERMODE	Reflects the state the processor was in when emulation was last halted. When in real mode, the protection enabled flag in the MSW is 0. When in protected mode, the protection enabled flag in the MSW is 1.

Examples

1. Display the initial settings for an 8086/8088 I²ICE probe in the current unit:

```
*STATUS
---  STATUS FOR UNIT 0000  ----
PROBETYPE=86  RSTEN=TRUE  TRCBUS=TRUE  BTHRDY=FALSE
COENAB=TRUE   CPMODE=1    MEMRDY=TRUE  IORDY=TRUE
BUSACT=TRUE   PHANG=TRUE
```

2. Display the settings for the 80186/80188 I²ICE probe in the current unit:

```
*STATUS
---  STATUS FOR UNIT 0000  ----
PROBETYPE=186  RSTEN=TRUE  TRCBUS=TRUE  BTHRDY=FALSE
COENAB=TRUE    CPMODE=1    MEMRDY=TRUE  IORDY=TRUE
BUSACT=TRUE    PHANG=TRUE  QSTAT=FALSE  INTICE=FALSE
```

3. Display the settings for a 80286 I²ICE probe in unit 2:

```
*UNIT=2
*STATUS
---  STATUS FOR UNIT 0002  ----
PROBETYPE=286  RSTEN=TRUE  SEL286=TRUE  PCHECK=TRUE
COENAB=TRUE    CPMODE=1    COREQ=TRUE   TRCBUS=TRUE
BTHRDY=FALSE   BUSACT=TRUE MEMRDY=TRUE  IORDY=TRUE
USERMODE=REAL
```

Cross-References

Address protection
BTHRDY
BUSACT
COENAB
C0REQ
CPMODE
IORDY
MEMRDY
PCHECK
PHANG
QSTAT
RSTEN
SEL286
TRCBUS

Strings

Character strings for use
as variables and displays

Syntax

$$\left\{ \begin{array}{l} \textit{'character[character]*'} \\ \textit{string-reference} \end{array} \right\}$$

Where:

'character[character]'*	is one or more characters enclosed in apostrophes. A string is stored as ASCII (byte) values.
string-reference	can be characters enclosed in apostrophes, a string expression using the CONCAT, NUMTOSTR, or SUBSTR function, or a reference to a CHAR type debug variable. *string-reference* includes any object within the PICE command language that is type CHAR.

Discussion

Strings contain one or more characters enclosed in apostrophes ('). To specify an apostrophe within a string, use two apostrophes. For example, the string 'WHAT''S UP?' is displayed as WHAT'S UP?.

The maximum length of a string is 254 characters, not counting the delimiters ('). Strings that are adjacent and separated by one or more logical blanks (space, tab, or carriage return/line feed) are concatenated to form a single string. With this feature you can break a string definition over a line boundary. The uppercase or lowercase status of the characters in the string is preserved. The null string consists of just the two delimiters (''). The CI, CONCAT, SUBSTR, and NUMTOSTR commands produce strings as results. Refer to the entries on each of these functions for examples.

Examples

1. The simplest reference is a string enclosed by apostrophes. After the following command executes, D (or d) is an abbreviation for the command word DEFINE.

```
*DEFINE LITERALLY d = 'DEFINE'
```

2. Another common way to refer to a string is with a debug variable of type CHAR.

```
*DEFINE CHAR msg1 = 'Do you want to break?'
```

3. Use the WRITE command to invoke the debug variable msg1 and display the message.

```
*WRITE msg1
Do you want to break?
```

4. The ASM type is also type CHAR; the string is the disassembled instruction or instructions. For example, suppose the current instruction is as follows:

```
*ASM $
0020:0005H 2E8E160000      MOV   SS,CS:WORD PTR 0000H
```

To save the disassembly of the current instruction, use it as a string reference (note that entering CHAR stat displays the address as well as the character string):

```
*DEFINE CHAR stat = ASM $
*stat
2E8E160000      MOV   SS,CS:WORD PTR 0000H
```

Cross-References

CI
CONCAT
NUMTOSTR
STRLEN
STRTONUM
SUBSTR

STRLEN

A function that returns the number of characters in a string

Syntax

STRLEN (*string-reference*)

Where:

string-reference	is a string reference that can be characters enclosed in apostrophes, a string expression using the CONCAT, NUMTOSTR, or SUBSTR function or a reference to a type CHAR debug variable.

Examples

1. Return the number of characters in the string "hello":

```
*STRLEN ('hello')
5
```

2. Return the number of characters in the debug variable "temp":

```
*DEFINE CHAR temp = 'hello'
*STRLEN (temp)
5
```

Cross-Reference

Strings

STRTONUM

A function that converts a string to a numeric value

Syntax

STRTONUM (*string-reference*)

Where:

string-reference	can be characters enclosed in apostrophes, a string expression using the CONCAT, NUMTOSTR, or SUBSTR function, or a reference to a type CHAR debug variable.

Example

1. In the following example, the STRTONUM function converts a string to a variable and forces it into the variable type.

```
*DEFINE REAL var2 = STRTONUM('1234.567E-2')
*var2
1.23457E+1
```

Cross-Reference

Strings

SUBSTR

Substring function that returns
a portion of a string

Syntax

SUBSTR (*string-reference, start, length*)

Where:

string-reference	can be characters enclosed in apostrophes, a string expression using the CONCAT, NUMTOSTR, or SUBSTR function, or a reference to a type CHAR debug variable.
start	is an expression with a value from 1 through 254 that specifies the index of the first character in the substring.
length	is an expression that specifies the number of characters required by the substring.

Discussion

With the SUBSTR function you can observe portions of a string. The SUBSTR function returns the substring *length* long, starting at the character indexed by start. If the index is out of range, the null string (a blank) is returned.

Examples

1. Suppose the opcode field of a disassembled instruction is a four-character field starting at position 20. You can test the field with the following command.

```
*IF SUBSTR (ASM $, 20, 4) = = 'MOV ' THEN WRITE 'I''ve found it'
.*END
I've found it
```

2. If *start* is valid but *length* is larger than the remaining characters in the string, all of the rest of the string is returned.

```
*SUBSTR ('abcdef', 3, 15)
cdef
```

Cross-References

Expression
Strings

SYMBOLIC

A pseudo-variable that enables or disables symbolic display in the trace buffer

Syntax

```
SYMBOLIC [ TRUE
         | =FALSE
         | =boolean-expression ]
```

Where:

SYMBOLIC	displays the current setting.
TRUE	permits the symbolic display of information in the trace buffer with the PRINT INSTRUCTIONS command.
FALSE	prohibits the display of symbolic information in the trace buffer with the PRINT INSTRUCTIONS command.
boolean-expression	is an expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default

TRUE

Symbolic references

References to program addresses and variables

Syntax (eight forms)

1. References to program modules:

   :module-name

2. References to program labels:

   [*:module-name.*][*procedure-name.*]* *label-name*

3. References to procedures:

   [*:module-name.*][*procedure-name.*]* *procedure-name*

4. References to line numbers:

   [*:module-name*] *#line-number*

5. References to variables:

   [*:module-name.*][*procedure-name.*]* *variable-name*

6. References to array variables:

   [*:module-name.*][*procedure-name.*]* *variable-name* [*expr* [, *expr*]*]

7. References to fields in a record or structure:

   [*:module-name.*][*procedure-name.*]* *record-name.field-name*[*.field-name*]*

8. Changing the value of a variable:

   variable-reference = expression

Where:

module-name *procedure-name* *label-name* *variable-name* *record-name* *field-name*	are names of program objects that follow the rules for identifiers.
line-number	is one or more decimal digits.
[*expr* [, *expr*]*]	is a list of one or more expressions identifying an element in an array. The list is enclosed in brackets (the required pair of brackets is underlined to distinguish them from the inner brackets indicating the optional part of the reference).
variable-reference	is a reference to a variable, array variable, or field in a record or structure.
expression	converts, if necessary, to the type of the variable in the *variable-reference*.

Discussion

The user program symbol table contains the names of all objects in the program, including the type and (for some objects) the length of each object. A symbolic reference identifies an object by name. When a symbolic reference is used in a command or expression, the value corresponding to the object is returned. The value returned depends on the type of the object. This section reviews the kinds of symbolic references and the values they represent. This section also discusses two special operators used with symbolic references, the double-quote operator and the dot operator.

User Symbol Table

The I²ICE system reads in information about the program symbols from the object file named in the LOAD command, unless symbol information is suppressed by a NOSYMBOLS or NOLINES option. To make this information available in the object file, use the DEBUG control on the compiler or assembler invocation. (Refer to the compiler or assembler manual for details.) Compilers also generate line numbers. Line number information is read in with the user program when the information is available.

Symbolic references continued

The I²ICE system organizes the symbol and line number information into a user symbol table. The user symbol table preserves scope and type information specified in the program file. When the type of a variable cannot be determined from the file, the system assigns it to null type. Operations that require type information are invalid with null types.

Names

All symbolic references except line numbers involve the names of objects. In the I²ICE command language, the character sets that identify objects are referred to as *name*. Command keywords, debug object names, and program symbols are all *name*. Refer to the Name entry in this encyclopedia for more information.

Compilers and assemblers use *name* for module names, procedure names, labels, and variable names in the source programs. Pascal and FORTRAN labels are decimal numbers in the source program. Language compilers append a leading at sign (@) to Pascal and FORTRAN labels, converting them to names. So, if a Pascal or FORTRAN program has a label 12, refer to this label as @12.

References to Program Addresses

References to modules, procedures, labels, and line numbers represent addresses in the program. The value returned is of type POINTER. In each case, the value represents the address of the first executable instruction in the module, in the procedure, at the label, or at the line number.

Modules

A module is identified with a leading colon (:). For example, if tca is the name of a module, the reference is as follows:

```
*:tca
1CA7H:0000H
```

The POINTER value returned is the first executable address in the module.

Line Numbers

A line number reference is the line number preceded by a number sign (#). The module name may be required as qualification. For example:

```
*#14
1CA7H:001AH

*:tcainv#134
1CA7H:04D9H
```

The NAMESCOPE pseudo-variable determines whether the line number reference requires a module name as a qualification. The NAMESCOPE pseudo-variable contains an address; if the address is within the desired module, you can omit the module name. Initially and after emulation breaks, NAMESCOPE contains the execution address. You can set NAMESCOPE to any executable address.

Procedures

A procedure reference is the name of the procedure. If necessary, the reference should include the module name and the names of any procedures that enclose the given procedure.

The NAMESCOPE variable determines when the procedure reference requires qualification (module name and outer procedure names). You can omit *module-name* if NAMESCOPE is within the desired module. If the NAMESCOPE address is within any enclosing procedures, these qualifiers can be omitted as well. For example:

```
*inner_procedure
0100:0200H

*:mod1.outer_procedure.inner_procedure
0100:0200H
```

The first example is valid only if NAMESCOPE is within procedure outer_procedure in module mod1.

Labels

Labels identify statements within procedures; typically, a label is used as the object of a GOTO statement or to control transfer statements within the procedure. The NAMESCOPE variable determines the qualifiers required to identify a label. For example:

```
*start_over
0100:0234H

:mod1.first_procedure.start_over
0100:0234H
```

The first example is valid only if NAMESCOPE is within procedure first_procedure in module mod1.

References to Program Variables

You can use a reference to a program variable to display the contents of the variable, to use the contents as an operand in an expression, or to change the contents of the variable. A reference to a program variable is valid if the variable is active at the current execution point and if it has a type that fits the context of the reference. The following sections detail the kinds of variables programs can contain, with the corresponding references.

Static Variables

The simplest kind of variable is a static variable representing a single scalar quantity. Static variables are always active and so can always be accessed. The address in NAMESCOPE determines the kind of qualification required to identify a variable, as discussed previously in the Procedures and Labels sections. The value returned by the reference depends on the type of the variable.

For example, assume the current program has symbols as shown by the following DIR command:

```
*DIR :calculatepi
DIR of :CALCULATEPI
MEMORY . . . . . array [?] of byte
DENOMINATOR . . . . . word
ELEMENT . . . . . real
I . . . . . word
PI . . . . . real
SGN . . . . . integer
TERMS . . . . . word
DONE . . . . . label
```

After executing then breaking within this module, references to any of the variables are valid. For example:

```
*pi
3.14159

*terms
5000
```

Dynamic Variables

Dynamic variables are based variables or stack-resident variables. The operating system memory manager allocates space for based dynamic variables as required during run-time. Stack-resident variables are allocated to the stack instead of fixed memory locations. Examples of stack-based variables are parameters in procedures, local variables in PL/M REENTRANT procedures, and all local variables in Pascal procedures.

The form of a reference to a dynamic variable is exactly the same as for a static variable. The difference is that if the execution point is not within the procedure that defines the variable, the variable is not active. By contrast, static variables are always active. An error results if you try to access a variable that is not active.

NOTE

When compiling a procedure, the compiler inserts a sequence of code called the prologue at the beginning of the procedure. The prologue reserves space for stack-based variables needed by the procedure. For stack-based variables to be fully active following a break within a procedure, the break must occur after the prologue.

For example, suppose the program has a procedure named start with a parameter i. If the execution point is not inside this procedure, the value of i is undefined. The following example shows the initial load of the program and the display of the program symbols for the initial module. Then it demonstrates that the symbol i is not active even when the correct qualifier is included. Finally it shows that after emulation breaks within the procedure (and past the prologue), the symbol i is valid.

```
*LOAD :F1:tca.86
*ISTEP 3                                    /*Execute past the prologue*/
*DIR :tca
DIR of :TCA
MEMORY . . . . . array [?] of byte
FINISH . . . . . procedure
START  . . . . . procedure
  DATE . . . . . array [9] of byte
  I  . . . . . byte
  TIME . . . . . array [22] of byte
  TIMESTAMP  . . . . . array [22] of byte
  STATUS . . . . . word
  SYSTEMID . . . . . array [22] of byte
MAIN . . . . . procedure
  COVERAGECLASSPTR . . . . pointer
  COVERAGEDEPTHPTR . . . . pointer
  RESULTPATHPTR  . . . . . pointer
  RESULTTYPEPTR  . . . . . pointer
  USERPATHPTR  . . . . . pointer
```

```
*EVAL $ PROCEDURE
:TCA.MAIN
*i
ERROR .12: Symbol not known in current context
*start.i
ERROR .11: Symbol currently not active
*GO TIL :tca.start
[Break at :tca.start]
*ISTEP                                  /* ISTEP to get past the prologue */
*i
8
```

Array Variables

An array consists of elements of a given type. To access an individual element, the reference specifies the index or indexes of the element. For example, suppose the array named date has nine elements and is defined (in PL/M) as an array of bytes. Then you can access any element with a reference such as the following:

```
*date[0]
8
```

Suppose also that element date[0] contains the length of the data portion of the array; in other words, the data elements are numbered 1 through the value of date[0] (in this case 8). The following debug procedure displays all the elements as bytes:

```
*DEFINE PROC dsply = DO
. *DEFINE BYTE index = 1
. *COUNT date[0]
. . *date[index]
. . *index = index + 1
. . *END
. *END
*dsply
48
54
47
49
49
47
56
50
*
```

To continue one step further, suppose date is really an array of ASCII characters. The next example displays the character values and also illustrates the use of the dot operator to specify the address of a variable (more details on the dot operator appear later in this section).

```
*CHAR .date[1] LENGTH date[0]
1C4FH:056DH '06/11/82'
```

Pascal arrays can have more than one dimension; the reference to an element in such an array must have the required number of indexes. For example, if array variable big_pascal_array has four dimensions, a reference might be as follows:

```
*big_pascal_array[10, 2, 34, 80]
126
```

Structure Variables

Variables with compound elements are called structures in PL/M and records in Pascal. The individual elements of a structure are called fields. A reference to a field value gives the structure name, then the field name with a dot (.) to separate the names. The name scope determines the amount of additional qualification required, just as for other variables.

For example, suppose a PL/M program has a structure declared as follows:

```
DECLARE house STRUCTURE (
                          stories BYTE
                          rooms BYTE
                          bathrooms BYTE);
```

After the program is executed so that the variable fields have taken on values (and assuming name scope is inside the module and procedure containing the structure), you can access the fields with references such as the following:

```
*house.stories
2

*house.rooms
10

*house.bathrooms
4
```

Compound Variables

The program can contain compound forms such as arrays of arrays, arrays of structures, and structures of arrays. The rules for references to these compound forms combine the rules discussed so far.

Symbolic references continued

As an example, suppose a PL/M program contains a structure defined as follows:

```
DECLARE table(9) STRUCTURE(
    option(10) BYTE ) data(
        8, 'CONTROLS ',
        7, 'MODULES ',
        5, 'LINES ',
        5, 'PROCS ',
        5, 'COUNT ',
        7, 'NOCOUNT ',
        4, 'LIST ',
        4, 'SAVE ',
        5, 'MERGE ',
                    );
```

References to this structure have forms such as the following:

```
*table[0].option[0]
8
```

To display an entire option in this structure, note that the first element in each row gives the number of ASCII characters in the option. For example:

```
*CHAR table[0].option[1] LENGTH table[0].option[0]
1DFEH:010FH 'CONTROLS'
```

The following debug procedure displays the entire structure:

```
*DEFINE PROC showtable = DO
. *DEFINE BYTE index = 0
. *COUNT 9T                                    /* Length of the table in decimal */
. . *CHAR .table[index].option[1] LENGTH table[index].option[0]
. . *index = index + 1
. . *END
. *END
*showtable
1DFEH:010FH 'CONTROLS'
1DFEH:0119H 'MODULES'
1DFEH:0123H 'LINES'
1DFEH:012DH 'PROCS'
1DFEH:0137H 'COUNT'
1DFEH:0141H 'NOCOUNT'
1DFEH:014BH 'LIST'
1DFEH:0155H 'SAVE'
1DFEH:015FH 'MERGE'
```

Based Variables and Pointer Variables

A based variable is referenced through another variable called its pointer variable. The pointer variable contains the address of the based variable. The PL/M definition of a pointer and its based variable might be as follows:

```
DECLARE optionptr POINTER;
DECLARE option BASED optionptr BYTE;
```

The executable part of the program then assigns a value to the based variable by setting the pointer to the desired address, then assigning the value with PL/M statements such as the following:

```
optionptr = old_option;
option = 42;
```

After this code is executed, the value of the pointer variable is the address of the variable. Assume the following addresses and contents for the two variables:

Variable	Address of Variable	Contents of Variable
old_option	1D00H:0874H	0
optionptr	1D00H:0420H	1D00H:0874H
option	1D00H:0874H	42

The following examples show references and displays for these variables. Note that with the I²ICE system you can refer to a based variable directly:

```
*option
42

*optionptr
1D00H:0874H

*BYTE optionptr
1D00H:0874H 42

*BYTE .option
1D00H:0874H 42
```

(The dot operator causes the system to return the address of the variable rather than the contents, as discussed later in this section.)

Note that for PL/M, the symbol table entry describing the pointer variable does not contain any information about the type of the based variable. To display the contents of the based variable, specify the type with a keyword such as BYTE, as in the previous examples.

Symbolic references continued

Changing the Value of a Variable

To make debugging easier, you can change the value of a program variable from the terminal. The value assigned is converted to the type of the variable. For example:

```
*:calculatepi.terms = 10000T
*
```

The variable must be active to receive a value. You cannot change the address corresponding to a module, procedure, label, or line number to a new value.

Double-Quote Operator

Keywords, debug object names, and user program symbols are all names. The emulator does not let you define a debug object with the same name as a keyword, but no checking is performed on the user symbols as they are loaded. Thus a user symbol may duplicate a keyword or debug object name. (For a list of PICE keywords, see the Keywords entry in this encyclopedia.)

In case of a duplication, keywords and debug object names have precedence over user symbols. Thus, if your program has a procedure named exit, the keyword EXIT masks out that symbol. The following command produces an error:

```
*GO TIL exit
        ↑ syntax error
```

To avoid conflict, precede the symbol with a double-quote operator ("). The double-quote operator forces the system to look up the entry as a user program symbol. The double-quote operator makes the following command valid:

```
*GO TIL "exit
[break at :mod1#534]
```

Dot Operator

The dot operator (.) can precede any of the references to program variables described previously. The effect is to return the address of the variable instead of the value of the variable. The dot operator is used whenever an address is required, as when using a type keyword to override the variable type or setting a breakpoint on a data address.

Thus, if your program has a BYTE variable named temp__variable in procedure getchar, the following reference returns the contents of the variable:

```
*getchar.temp__variable
19
```

However, the following reference uses the dot operator to return the location of the variable:

```
*.getchar.temp_variable
011AH:0056H
```

The dot operator should precede the outermost qualifier, including the module name if it is used. For example:

```
*.:tca.start.i
1DFEH:0125H
```

Cross-References

Address
Expression
Keywords
Name

SYSREG

Defines a register that contains system break specifications

Syntax

DEFINE SYSREG *name* = [SYSTRIG | SYSARM | SYSDARM] *system-specification* [CALL *dproc*]

Where:

DEFINE SYSREG *name* = *system-specification*	creates a system register called *name*. Specifying *system-specification* after the equal sign (=) defines the break criteria. The System specification entry in this encyclopedia describes this syntax.
name	is the name of the system register you are creating.
SYSTRIG	specifies that when the *system-specification* is met, any I²ICE units enabled are triggered and perform the programmed action.
SYSARM	specifies that when the *system-specification* is met, any I²ICE units enabled are armed to perform the programmed action.
SYSDARM	specifies That when the *system-specification* is met, any I²ICE units enabled are disarmed.
system-specification	defines the break criteria. The System specification entry in this encyclopedia defines *system-specification*.
CALL *dproc*	calls the debug procedure named when *system-specification* is met. The called debug procedure must return either TRUE (meaning a break is to occur) or FALSE (meaning emulation is to continue). CALL *dproc* is never activated when a SYSTRIG, SYSARM, or SYSDARM is specified.
dproc	is the name of the debug procedure you want to call when *system-specification* is met.

Discussion

The SYSREG command defines breaks on operand access, operand data, logic clips, system breaks, and coprocessor cycles.

Read the System specification entry in this encyclopedia to familiarize yourself with the terms used in the following discussion.

When to Use SYSREGs

The two ways to use a system specification are to state it in the GO command or to use a debug register called a SYSREG (system register) in the GO command.

Use system registers (SYSREGs) to define processor state specifications. The state is any of the following items expressed as addresses or data:

- READ from memory
- WRITE to memory
- INPUT from an input port
- OUTPUT to an output port
- FETCH an instruction

NOTE

A break specification is not a system specification. The BRKREG, ARMREG, EVTREG, and GO commands define *break-specification* breakpoints.

How to Specify SYSREGs

Note that you can optionally enclose the entire specification following the equal sign in a DO-END block.

Specifying Addresses and Data

The DEFINE SYSREG command distinguishes between address and data values. All *bus-data* items are prefixed with the keyword IS. All *bus-address* items are prefixed with the keyword AT. (The syntax for *bus-data* and *bus-address* is defined in the System specification entry.)

Processor States

Processor state items are lists of break conditions (e.g., READ). Lists can be ORed together to form compound break conditions. For example:

READ OR WRITE AT *address*

Furthermore, you can combine an ORed list with a logically ANDed list. For clarity, you can insert the optional keyword WITH as a reminder that a logical AND is being specified. For example, the following two commands are the same:

```
*READ OR WRITE AT address FULLBUF
*READ OR WRITE AT address WITH FULLBUF
```

Using the Optional Call

When emulation stops because of a SYSREG that includes a CALL, the CALL transfers control to the named debug procedure. The debug procedure must return a Boolean value (TRUE or FALSE). If TRUE, emulation stops and a break message is displayed. If FALSE, emulation continues.

NOTE

Emulation halts if a Boolean value is not returned or if there is an error in the called debug procedure. An error message indicates that the halt was not caused by a normal execution break.

Manipulating SYSREGs

Manipulate a SYSREG by referring to it by name. You can manipulate SYSREGs in the following ways:

- Create a SYSREG with the DEFINE command
- Delete a SYSREG from memory with the REMOVE command
- List SYSREG names with the DIR command
- Save (or restore) a SYSREG to (or from) a file with the PUT or APPEND (or INCLUDE) commands
- Display a SYSREG with the SYSREG command
- Execute a SYSREG with the GO USING command

- Use a SYSREG as part of the DEFINE ARMREG specification
- Modify a SYSREG with the editor

NOTE

Defining new break specifications using an old SYSREG name destroys the old definition in memory. An error results if you try to assign a SYSREG name to any other debug object in memory.

Restoring a saved SYSREG that has the same name as a SYSREG in memory overwrites the latter.

An error occurs if you try to restore a saved SYSREG that has the same name as another debug object in memory.

Because SYSREGs are referred to by name, you can reuse break specifications without re-entering them. The GO command allows SYSREG lists. By combining SYSREGs, you can switch breakpoints in a GO statement by changing SYSREG names.

Using SYSREGs with Multiple Units

The keywords SYSTRIG, SYSARM, and SYSDARM indicate actions caused by system registers. Other units in the I²ICE system must be enabled to respond to a system action. The unit causing the system action is also affected if it is enabled. Refer to the ENABLE and SYSTEM entries in this encyclopedia for details.

Restrictions

System registers can contain any number of specifications, limited only by memory. The GO command's ability to execute them is limited by the number of word recognizers available.

Word recognizers are the programmable portion of the internal execution state machine. They compare user match specifications with conditions on the bus they monitor. When a match occurs, the state machine halts emulation. Refer to the Event machine entry in this encyclopedia for details.

Word recognizer use is governed internally. You cannot know precisely how many word recognizers the I²ICE system uses in any given specification. A good rule of thumb is that one- or two-range (partition) specifications or four-location specifications are the upper limit.

The I²ICE system indicates when the word recognizer limit is exceeded.

SYSREG continued

Example

1. The following example shows how to define a SYSREG to trigger a system break when bus address 105H is read and contains the value 45H. In addition, the value of the input logic clips must be 0010X01H.

```
*DEFINE SYSREG sys = DO
**SYSTRIG READ AT 105H IS 45H WITH CLIPS 0010X01H
**END
*
```

Cross-References

ENABLE
Event machines
Name
SYSTEM
System specification

SYSTEM

Sets the initial state of
the system arming functions

Syntax

SYSTEM { ARM / DISARM }

Where:

SYSTEM ARM	sets the initial system state to armed.
SYSTEM DISARM	sets the initial system state to disarmed.

Default

SYSTEM ARM

Discussion

The SYSTEM ARM/DISARM command is issued before entering an emulation using the SYSTRIG function. The condition is reinitialized to this state each time a GO command is executed. You should set the initial state to SYSTEM DISARM when system arming functions will be used in the debug session. Otherwise, you should set the initial state to SYSTEM ARM.

Example

1. The following example defines an arm register that arms the system when line 12 of module TEST is executed, triggers the current probe when line 25 is executed, and then disarms the current probe.

```
*UNIT = 0
*DEFINE ARMREG seq = DO
**ARM SYSARM AT :test #12
**TRIG :test #25 END
*SYSTEM DISARM
*GO FROM :top_module USING seq
Probe 0 stopped at :test #25+2 because of execute break
```

System specification

Defines system specifications
for execution control commands

The term *system-specification* has a special meaning in the syntax of I²ICE system commands. The execution control commands (i.e., GO, ARMREG, SYSREG, TRCREG, and EVTREG) use this term in their syntax definitions.

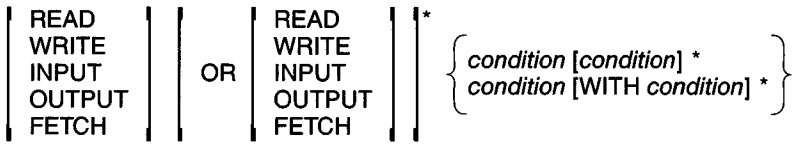
The *break-specification*, defined in the Break specification entry in this encyclopedia, refers to object code addresses. The *system-specification*, defined syntactically in the following Syntax section, is essentially everything else, including bus addresses, bus data, probe processor status, trace buffer full, and the optional external logic clips input (if attached).

You can categorize break or trace information into *break-specification* or *system-specification*. Only certain keywords apply in each case, which cuts down on the number of keywords to be recalled per situation.

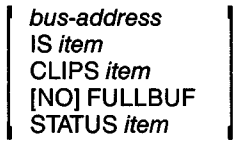
This entry describes how to specify the syntax for each branch of the *system-specification* term.

When you construct the control command line, substitute the syntax shown in the Syntax section for *system-specification*.

Syntax



condition is



bus-address with READ, WRITE, or FETCH is

```
[ AT masked-constant [, masked-constant] * ]
[ AT partition                             ]
```

bus-address without READ, WRITE, or FETCH is

```
[ AT masked-constant [, masked-constant] * ]
[ AT [OUTSIDE] partition                   ]
```

item is

```
[ expression [,expression]*                ]
[ masked-constant [,masked-constant]*      ]
[ [OUTSIDE] partition                      ]
```

Where:

condition — qualifies the READ, WRITE, INPUT, OUTPUT, or FETCH. If you do not include a READ, WRITE, INPUT, OUTPUT, or FETCH, the I²ICE system assumes the OR of all the possibilities.

bus-address — is the address of an opcode or an operand (data). Addresses refer to the emulating processor's bus. Qualifying the bus address with the processor's *status* distinguishes between opcode fetches and data accesses. For example:

```
DEFINE SYSREG sys = FETCH AT address

DEFINE SYSREG sys = READ AT address
```

Instruction bytes are prefetched into the processor queue but might not be executed. Branch instructions flush the queue.

Fetched instructions appear in the PRINT CYCLES trace buffer display in the BUS ADR column. Instructions are interspersed with operand data.

The following *bus-address* options are valid:

AT *address-one*
AT OUTSIDE *address-start* LENGTH 50
AT *address-one, address-two, address-three*
AT *.data*
AT OUTSIDE *.databegin* TO *.dataend*
AT *.start* LENGTH 0BH
AT X0X1101Y
AT OUTSIDE 45H TO 50H
AT CS:3000
AT 0AXXFH

When the system specification is a READ, WRITE, or FETCH, you cannot specify the *bus-address* to be OUTSIDE a *partition*.

The BUS ADR column of the PRINT CYCLES trace display indicates addresses and data.

OUTSIDE — tells the I²ICE system to recognize all addresses other than those in the *partition* (a logical NOT function).

item — is either an expression, a partition, or a masked constant.

IS *item* — specifies data or an instruction read (from memory or an I/O port) or written (to memory or an I/O port).

CLIPS *item* — specifies one of the eight input or one of the two output logic clips supplied with the I²ICE system. The input clips are displayed with the CLIPSIN command. The output clips are set with the CLIPSOUT command. Refer to the CLIPSIN and CLIPSOUT entries in this encyclopedia for details.

Use the eight input clips to qualify system-specification breakpoints. Using masked constants simplifies CLIPS breakpoint specifications. Specifying a CLIPS *item* breakpoint when the input clips are not attached results in the message "No clips module".

[NO] FULLBUF — specifies a break or trace or both when the trace buffer is full. The keyword NO is a logical NOT. The trace buffer is a system-specification condition. Stopping emulation or tracing on the buffer full condition prevents the buffer from being entirely overwritten.

STATUS *item*	refers to the state of the probe processor. See the PRINT entry in this encyclopedia for more information about processor status.
partition	is a bounded range of contiguous addresses. A symbolic reference to a module or procedure is a partition.
expression	represents a single address or list of addresses. Addresses are numbers or symbolic references.
masked-constant	is a binary or hexadecimal number with one or more locations set to a don't-care condition. Replacing numbers with an uppercase or lowercase X tells the I²ICE system to accept any number as a valid match. Masked constants represent 32 bits. If the base is binary, each X represents one bit. If the base is hexadecimal, each X represents four bits. Unspecified leading bits are filled with zeros.

Discussion

A status item can appear in an ORed list only once. For example, READ OR READ is illegal. The WITH keyword logically ANDs conditions.

Refer to the Expression, Masked constant, and Partition entries in this encyclopedia for details on *expression, masked-constant*, and *partition*, respectively.

When specifying bus addresses, you need to know Intel's iAPX architecture. The 16-bit iAPX microprocessors access memory a word at a time. They access memory as words beginning on even addresses. If the word exists at an odd address, the 16-bit iAPX microprocessor performs two memory accesses, both at even addresses.

For example, assume that the word AB12H is stored at the even address FC00H. Memory is arranged as follows:

```
FC00    12
FC01    AB
FC02    ??
FC03    ??
```

When you read the word at FC00H, one memory access occurs. The data bus contains AB12. Assume that your program reads that word. You can define a system debug register that triggers a break when the read occurs.

```
*DEFINE SYSREG even = READ IS 0AB12H
```

Now assume that the word AB12 is stored at the odd address FC01 and that FC00H and FC03 both contain 90H. Memory is arranged as follows:

```
FC00    90
FC01    12
FC02    AB
FC03    90
```

When you read FC01H, two memory accesses occur. The memory bus contains 1290 the first time and 90AB the second time. The previous system debug register no longer causes a break. Because the word AB12 is at an odd address, it does not appear on the data bus as a complete word. You can still break on that condition, but you must use an event debug register.

```
*DEFINE EVTREG odd = DO
**SEM S0 IF READ AT 0FC01H IS 12XXH
**          THEN GOTO S1
**       S1 IF READ AT 0FC02H IS 0XXABH
**          THEN BREAK
**          ELSE GOTO S0
**END
```

Use the FETCH keyword carefully. For example, assume that you want to specify a break when the emulating microprocessor fetches the instruction beginning at statement #13 in the module cmaker.

```
*GO TIL FETCH AT :cmaker#13
```

If that instruction is at an odd address, the break might not occur. The 16-bit microprocessors do word fetches from even addresses, with one exception. The first fetch after a program transfer to an odd address obtains a byte. If you program transfers control to an odd address, the fetch is from that odd address.

Examples

1. The following example shows the syntax for a DO statement containing *system-specification* options.

```
DEFINE SYSREG x = DO                          /* Debug register named x */
status
bus-address                                   /* Partial construction */
CLIPS item                    /* The CLIPS keyword is entered as is */
END                                           /* Matches the DO */
```

2. The following example shows the same DO statement with the *system-specification* options defined.

```
*DEFINE SYSREG x = DO
**READ
**AT :mod1.procA
**CLIPS 110XXY
**END
*
```

Cross-References

Expression
Masked constant
Partition

TEMPREAL

Displays or changes memory
as 80-bit floating-point values

Syntax

TEMPREAL *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

TEMPREAL *partition*	displays the contents of memory in the *partition* as a temporary real number in scientific format.
partition	is a single address or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to an 80-bit floating-point value for TEMPREAL.
mtype	is any of the memory types except ASM.

Discussion

The TEMPREAL command interprets the contents of memory as 80-bit floating-point decimal values, overriding any type associated with the memory contents. Thus, TEMPREAL .var1 displays the 80-bit floating-point value that begins at the address of var1, regardless of the type of var1.

Examples

The following examples assume a decimal number base.

1. Display a single value:

```
*TEMPREAL $
0020:0006H +3.365797667020075OE-199
```

2. Display several adjacent values:

```
*TEMPREAL $ LENGTH 3
0020:0006H +3.3657976670200750E -199 -0.0507276084731473SE+3623
0020:0020H +0.0368425402359838E +4856
```

3. Set a single value of type TEMPREAL:

```
*TEMPREAL 40H:4H = 120.344444444444
```

4. Set several adjacent values:

```
*TEMPREAL 40H:4H = 123456789012345678 9t, -23456.67890, 5
```

Display the values set:

```
*TEMPREAL 40H:4H LENGTH 3
0040:0004H +1.2345678901234567 89E +18 -2.3456678900000000000E+4
0040:0018H       +5.0000000000000000
```

5. Set a range of locations to the same value:

```
*TEMPREAL 40H:4H LENGTH 10 = 0
```

6. Set a repeating sequence of values:

```
*TEMPREAL 40H:4H LENGTH 10 = 5.678, -2300, 23456, -7.567
```

7. Copy a value from one memory location to another:

```
*TEMPREAL 40H:4H = TEMPREAL $
```

8. Copy several values (block move):

```
*TEMPREAL 40H:4H = TEMPREAL $ LENGTH 10
```

9. Copy values with type conversion:

```
*TEMPREAL 40H:4H = EXTINT .var2
```

An error message is displayed if the type on the right side of the equal sign cannot be converted to the type on the left. (Refer to the Expression entry in this encyclopedia for the rules concerning type conversions.)

Cross-References

Expression
Mtype
Partition

TIMEBASE

A pseudo-variable that sets the trace counter source and increment and formats the trace buffer timetag

Syntax

TIMEBASE = *integer* { NS | US | MS }

Where:

integer	is a number or an expression that evaluates to a positive whole number. The *integer* controls the amount of time between increments of the timebase counter. The *integer* ranges from 200 nanoseconds (NS) through 6,553 microseconds (US), in multiples of 100 NS.
NS US MS	is the time measurement suffix for *integer*; NS is nanoseconds, US is microseconds, and MS is milliseconds.

Default

200 NS

Discussion

The TIMEBASE command uses the timebase counter and increment settings to calculate and save timetag information. Timetag is part of the trace buffer displayed with the PRINT CYCLES command.

The TIMEBASE command controls a free-running counter (timebase counter). By specifying a timebase increment value, you control how often the counter increments by one. The current counter value is transparent to the user. However, the timebase increment and count value determine the timetag in the PRINT CYCLES trace buffer display.

The timebase counter starts at zero, counts until it wraps around and then starts over. Wrap-arounds are transparent. Timetag is calculated as follows: (timebase increment * timebase counter). The value of the timebase counter is saved every time an instruction is executed.

The TIMEBASE command defines the amount of time between increments for all units. The current unit is the source of the timebase counter. Traced events between I²ICE probes are synchronized because they use the same TIMEBASE setting.

NOTE

Trace buffer timing is not synchronized with the Intel logic timing analyzer (iLTA) timing information.

The effect of the TIMEBASE command is important in two instances: when tracing is continuous and when trace is switched on and off.

Continuous Tracing

Using TIMEBASE to set up timetag information is straightforward when tracing is continuous. You can accurately compare traces collected from one or more probes without interruption. Timebase counter wrap-arounds are transparent.

Interrupted Tracing

Turning the trace on, off, and back on again can create synchronization problems. A discontinuity occurs if the timebase counter wraps around while trace is off. Wrap-arounds with trace off occurs when the timebase counter value, which is free running, has been incremented 2048 times before trace is resumed. For example, suppose TIMEBASE = 1US was set. A trace discontinuity occurs if the length of time between turning the trace off and turning it back on is longer than (1US * 2048).

A trace discontinuity is displayed in the PRINT cycles trace buffer under the heading LEVEL. The newest trace data collected is always at level 0. A trace discontinuity increments the level count by one.

Only trace data at level 0 is comparable for timing synchronization between probes. Trace data timing information is comparable for any single level, but only for each probe considered individually.

Example

1. The following example shows how to set TIMEBASE parameters and shows the trace buffer display that results using the 8086/8088 probe. The timebase counter source is the trace board in unit 3 and the increment, referenced by all probes, is 200 microseconds.

```
*UNIT = 3
*TIMEBASE = 200 us
*GO
*PRINT CYCLES ALL
```

```
EXEC ADR  BUS ADR   DATA   STATUS  CLIPS FRAME  TIME       LEVEL UNIT (
x         b 000202 d C38B s 0054 CF c     f 000
x 000202  b        d      s         c 02  f 001        0 nanosecs   0
x         b 000204 d FAE2 s 0054 CF c     f 002
x 000204  b        d      s         c 02  f 003      0.8 microsecs  0
x         b 000206 d F8EB s 0054 CF c     f 004
x         b 000208 d E001 s 0054 CF c     f 005
x         b 00020A d C103 s 0054 CF c     f 006
x 00020A  b        d      s         c 83  f 007      4.4 microsecs  0
x         b 000202 d C38B s 0054 CF c     f 008
```

Cross-Reference

Expression

Trace buffer display
8086/8088 probe specific

The PRINT command (discussed in the PRINT entry of this encyclopedia) displays the contents of the trace buffer. The INSTRUCTIONS option displays the trace buffer in disassembled mnemonics, and the CYCLES option displays the trace buffer in bus cycles.

INSTRUCTIONS mode shows bus activity and execution within the emulating probe processor. The display of execution within the probe processor shows the frame number, execution address, instruction opcode, mnemonic, and unit number, as shown in the following example:

```
FRAME    ADR       BYTE     MNEMONICS OPERANDS   UNIT 0
001      000204H   FA         CLI
```

The following example shows how bus activity is displayed:

```
000390H-SW-0021H
```

The format of the bus activity display is as follows:

bus address-access code-data

Where:

bus address	is a bus address.
access code	is a two-character access code representing the origin of the I²ICE trace data. The first character represents the access type, and the second character represents processor activities. Table 1-34 defines the access codes.
data	is the I²ICE trace data.

Table 1-34 8086/8088 INSTRUCTIONS Mode Access Codes

Access Type	Code	Processor Activity	Code
Segment		Fetch instruction	F
Extra segment (ES)	E	Read memory	R
Stack segment (SS)	S	Write memory	W
Code segment (CS)	C	Input from I/O port	I
Data segment (DS)	D	Output from I/O port	O
		Halt	H
Coprocessor		Interrupt acknowledge	A
RQ/GT0	0		
RQ/GT1	1		

CYCLES mode displays the execution address, bus address, bus data, processor status, clip information, frame number, timetag, level, and unit number. The status column in CYCLES mode contains a 16-bit hexadecimal bus status code followed by a two-character access code. Table 1-35 defines the two-character access codes.

Table 1-35 8086/8088 CYCLES Mode Access Codes

Bits			Function	Code
2	1	0		
0	0	0	Interrupt acknowledge	A
0	0	1	Input from I/O port	I
0	1	0	Output from I/O port	O
0	1	1	Halt	H
1	0	0	Fetch instruction	F
1	0	1	Read memory	R
1	1	0	Write memory	W
Bits			**Function**	**Code***
2	1	0		
0	0	0	Extra segment (ES)	E
0	0	1	Stack segment (SS)	S
0	1	0	Code segment (CS)	C
0	1	1	Data segment (DS)	D

NOTE

When bit 5 = 0, bits 1 through 4 are interpreted as access codes; when bit 5 = 1, a coprocessor bus cycle is indicated and bit 3 then indicates the coprocessor number.

Bit 6: Status of TEST pin.
Bit 7: System event machine (SEM) in state 3
(XLINK) when bit 7 = 1.
Bits 8-15: Unused.

Trace buffer display (8086/8088) continued

Examples

1. The following example shows a sample 8086/8088 probe trace buffer displayed in INSTRUCTIONS mode.

```
*PRINT INSTRUCTIONS ALL
FRAME    ADR    BYTE          MNEMONICS  OPERANDS                UNIT 0
001 000204H    FA             CLI
002 000205H    2E8E160000     MOV SS,CS:WORD PTR 0000H
008 00020AH    BC7200         MOV SP,0072H ;+114
00A 00020DH    2E8E1E2EBC     MOV DS,CS:WORD PTR 0BC2EH
010 000212H    EA00012100     JMP 0021H:0100H
015 000310H    8BEC           MOV BP,SP
017 000312H    FB             STI
018 000313H    2E8D460800     LEA AX,CS:WORD PTR 0008H
01D 000318H    0E             PUSH CS
020 000319H    50             PUSH AX
    000390H-SW-0021H
023 00031AH    9A34003A00     CALL 003AH:0034H
    00038EH-SW-0008H     0003BCH-SW-0021H
028 0003D4H    1E             PUSH DS
    00038AH-SW-010FH
02B 0003D5H    55             PUSH BP
    000388H-SW-0032H
02E 0003D6H    8BEC           MOV SP,SP
    000386H-SW-0072H
030 0003D8H    8ED E0AH       MOV DS,[BP+0AH]
    000390H-SR-0021H
035 0003DBH    8B5E0B         MOV BX,[BP+0BH]
    00038EH-SR-0008H
038 0003DEH    BF0000         MOV DI,0
03A 0003E1H    BACE00         MOV DX,00CEH ;+206
```

2. The following example shows a sample 8086/8088 probe trace buffer displayed in CYCLES mode.

```
EXEC ADR   BUS ADR    DATA   STATUS  CLIPS FRAME   TIME  LEVEL     UNIT 0
x          b 000202   d C38B s 0054 CF c    f 000
x 000202   b          d      s         c 02 f 001    0.0 nanosecs   0
x          b 000204   d FAE2 s 0054 CF c    f 002
x 000204   b          d      s         c 02 f 003    0.8 microsecs  0
```

```
x          b 000206  d F8EB s 0054 CF c    f 004
x          b 000208  d E001 s 0054 CF c    f 005
x          b 00020D  d C103 s 0054 CF c    f 006
x 000200   b         d      s         c 83 f 007  4.4 microsecs 0
x          b 000202   d C38B s 0054 CF c    f 008
```

Cross-Reference

PRINT

Trace buffer display
80186/80188 probe specific

The PRINT command (discussed in the PRINT entry in this encyclopedia) displays the contents of the trace buffer. The INSTRUCTIONS option displays the trace buffer in disassembled mnemonics, and the CYCLES option displays the trace buffer in bus cycles.

INSTRUCTIONS mode shows bus activity and execution within the emulating probe processor. The display of execution within the probe processor shows the frame number, execution address, instruction opcode, mnemonic, and unit number, as shown in the following example:

```
FRAME    ADR    BYTE          MNEMONICS  OPERANDS        UNIT 0
3E6 000016H    EA10000000      JMP  0000H:0010H
```

The following example shows how bus activity is displayed:

```
000010H-W-10A1H
```

The format of the bus activity display is as follows:

bus address-access code-data

Where:

bus address	is a bus address.
access code	is a two-character access code representing the origin of the I²ICE trace data. The first character represents the access type, and the second character represents processor activities. Table 1-36 defines the access codes.
data	is the I²ICE trace data.

Table 1-36 80186/80188 INSTRUCTIONS Mode Access Codes

Access Type Character	Code	Device Activity	Code
DMA channel 0	0	Fetch instruction	F
DMA channel 1	1	Read memory	R
Coprocessor	C	Write memory	W
Normal CPU activity	(blank)	Input from I/O port	I
		Output from I/O port	O
		Halt	H
		Acknowledge interrupt	A

CYCLES mode displays the execution address, bus address, bus data, processor status, clips information, frame number, timetag, level, and unit number. The status column in CYCLES mode display contains a 16-bit hexadecimal bus status code followed by a one-character access code. Table 1-37 defines the access code.

Table 1-37 80186/80188 CYCLES Mode Access Codes

Bits			Function	Code
2	1	0		
0	0	0	Interrupt acknowledge	A
0	0	1	Input from I/O port	I
0	1	0	Output from I/O port	O
0	1	1	Halt	H
1	0	0	Fetch instruction	F
1	0	1	Read memory	R
1	1	0	Write memory	W

NOTE

Bit 3 = 1	DMA channel 0 bus cycle
Bit 4 = 1	DMA channel 1 bus cycle
Bit 5 = 1	Coprocessor bus cycle
Bit 6 = 1	Bus cycle with lock asserted
Bit 7 = XLINK	The Event machines entry in this encyclopedia describes XLINK.

Examples

The examples in this section are based on the following assembly language program:

```
0000:0010H  A11000        MOV AX,WORD PTR 0010H
0000:0013H  A31000        MOV WORD PTR 0010H,AX
0000:0016H  EA10000000    JMP 0000H:0010H
0000:001BH  90            NOP
0000:001CH  90            NOP
0000:001DH  90            NOP
```

1. The following example shows a sample 80186/80188 probe trace buffer displayed in INSTRUCTIONS mode.

Trace buffer display (80186/80188) continued

```
FRAME   ADR         BYTE         MNEMONICS  OPERANDS                    UNIT 0
3E6 000016H         EA10000000     JMP   0000H:0010H
        000010H- W-10A1H
3EC 000010H         A11000         MOV   AX,WORD PTR 0010H
3EF 000013H         A31000         MOV   WORD PTR 0010H,AX
        000010H- R-10A1H
3F2 000016H         EA10000000     JMP   0000H:0010H
        000010H- W-10A1H
3F8 000010H         A11000         MOV   AX,WORD PTR 0010H
3FB 000013H         A31000         MOV   WORD PTR 0010H,AX
        000010H- R-10A1H 000010H- W-10A1H
```

2. The following example shows a sample 80186/80188 probe trace buffer displayed in CYCLES mode.

```
EXEC ADR  BUS ADR   DATA   STATUS  CLIPS FRAME   TIME          LEVEL UNIT 0
x 000013 b        d      s         c    f 3EF  408.2 microsecs  0
x        b 000010 d 10A1 s 0005 R   c    f 3F0
x        b 000016 d 10EA s 0004 F   c    f 3F1
x 000016 b        d      s         c    f 3F2  409.2 microsecs  0
x        b 000010 d 10A1 s 0006 W   c    f 3F3
x        b 000018 d 0000 s 0004 F   c    f 3F4
x        b 00001A d 9000 s 0004 F   c    f 3F5
x        b 00001C d 9090 s 0004 F   c    f 3F6
x        b 000010 d 10A1 s 0004 F   c    f 3F7
x 000010 b        d      s         c    f 3F8  411.6 microsecs  0
x        b 000012 d A300 s 0004 F   c    f 3F9
x        b 000014 d 0010 s 0004 F   c    f 3FA
x 000013 b        d      s         c    f 3FB  413.0 microsecs  0
x        b 000010 d 10A1 s 0005 R   c    f 3FC
x        b 000016 d 10EA s 0004 F   c    f 3FD
x        b 000010 d 10A1 s 0006 W   c    f 3FE
```

Cross-Reference

PRINT

Trace buffer display
80286 probe specific

The PRINT command (discussed in the PRINT entry in this encyclopedia) displays the contents of the trace buffer. The INSTRUCTIONS option displays the trace buffer in disassembled mnemonics, and the CYCLES option displays the trace buffer in bus cycles.

INSTRUCTIONS mode shows bus activity and execution within the emulating probe processor. The display of execution within the probe processor shows the frame number, execution address, instruction opcode, mnemonic, and unit number, as shown in the following example:

```
FRAME   ADR    BYTE        MNEMONICS OPERANDS       UNIT 0
3A1     976H   8EC0        MOV       ES,AX
```

The following example shows how bus activity is displayed:

```
0005A6H- R-007CH
```

The format of the bus activity display is as follows:

bus address-access code-data

Where:

bus address — is a bus address.

access code — is a two-character access code representing the origin of the I²ICE trace data. The first character represents the access type, and the second character represents processor activities. Table 1-38 defines the access codes.

data — is the I²ICE trace data.

Table 1-38 Access Code in the Trace Buffer Display

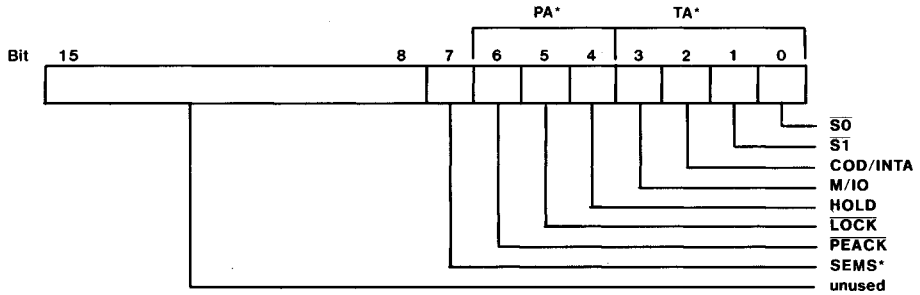
Processor Activity	Code: First Character		Code: Second Character
Processor extension activity	C	Instruction fetch	F
Locked instruction	L	Read from memory	R
HOLD asserted	H	Write to memory	W
Normal CPU activity	blank	Input from I/O port	I
		Output to I/O port	O
		Halt	H
		Acknowledge interrupt	A

Trace buffer display (80286) continued

In CYCLES mode, the trace buffer displays a 16-bit status word. The upper eight bits (bits <15-8>) are zero. The status word decodes into the two-character access code (the access codes are defined in Table 1-38). Figure 1-20 shows the decoding of the status word.

Note that breakpoints can be set to the occurrence of any given bus cycle type by using the status word. For example, the following command causes the probe to break on any interrupt acknowledge cycle:

```
GO FROM $ TIL STATUS IS 0X0000Y
```



*Additional information:

SEMS	When 1, signifies that the system event machine is in state 3 (SLINK=1).	
PA	xOx	Locked instruction, decoded as L.
	x11	HOLD asserted, decoded as H.
	010	Processor extension activity, decoded as C.
	110	Normal CPU activity.
	x	Don't-care bit.
TA	0000	Acknowledge interrupt, decoded as A.
	0100	Halt, decoded as H.
	0101	Memory read, decoded as R.
	0110	Memory write, decoded as W.
	1001	Input from I/O port, decoded as I.
	1010	Output from I/O port, decoded as O.
	1101	Instruction fetch, decoded as F.

2277

Figure 1-20 The 80286 Status Word Bit Pattern

Examples

The examples in this section are based on the following PL/M source code.

```
FLOATING_POINT_MATH_TEST:
DO;
/* This program performs a few math routines for ICE86A emulation of 8087 and emulator */

DECLARE  ERROR(10)  BYTE  INITIAL(0,0,0,0,0,0,0,0,0,0);
DECLARE  (A,B)  REAL;
DECLARE  PT  WORD  INITIAL(.A);
DECLARE  RESULT  BASED  PT(1)  WORD;
DECLARE  I  BYTE;
DECLARE  REAL_ST_BUF(100)  BYTE;
DECLARE  FERROR  WORD  INITIAL(0);
DECLARE  LOOP_COUNT  BYTE  INITIAL(0);

CALL  INIT$REAL$MATH$UNIT;
CALL  SET$REAL$MODE(33EH);

/*  Summation  */
A,B=0.0;
DO  I=1  to  5;
    B=B+1000.0;
    A=A+B;
END;
IF  A  <>  15000.0  THEN  ERROR(1)=1;

/*  Anti-summation  */
B=0.0;
DO  I=1  TO  5;
    B=B+1000.0;
    A=A-B;
END;
IF  A  <>  0.0  THEN  ERROR(2)=1;

/*  Negative  summation  */
A,B=0.0;
DO  I=1  TO  5;
    B=B-1000.0;
    A=A+B;
END;
IF  A  <>-15000.0  THEN  ERROR(3)=1;

/*  Anti-negative  summation  */
B=0.0;
```

Trace buffer display (80286) continued

```
DO  I=1  TO  5;
    B=B+1000.0;
    A=A+B;
END;
IF  A  <>  0.0  THEN  ERROR(4)=1;

/*  DOUBLE  */
A=0.0;
B=100.0;
DO  I=1  TO  10;
    A=A+B;
    B=A;
END;
IF  A  <>  51200.0  THEN  ERROR(5)=1;

/*  Divide  and  subtract  */
DO  I=1  TO  10;
    A=A  /  2.0;
    B=B-A;
END;
IF  B  <>  A  THEN  ERROR(6)=1;
IF  A  <>  50.0  THEN  ERROR(7)=1;

/*  Factorial  */
A=1.0;
B=2.0;
DO  I=1  TO  9;
    A=A  *  B;
    B=B+1.0;
END;
IF  RESULT(0)  <>  7COOH  OR  RESULT(1)  <>  4A5DH  THEN  ERROR(8)=1;

/*  Anti-factorial  */
DO  I=1  TO  10:
    B=B-1.0;
    A=A  /  B;
END;
IF  RESULT(0)  <>  0  OR  RESULT(1)  <>  3F80H  THEN  ERROR(9)=1;

LOOP_COUNT=LOOP_COUNT+1;

CALL  SAVE$REAL$STATUS(@REAL_ST_BUF);

COMPLETED:  HALT;

END;
```

Disassemble the source code by entering the ASM command.

```
*ASM $ LENGTH 10
0020:0006H  FA                  CLI
0020:0007H  2E8E1600            MOV  SS,CS:WORD PTR 0000H
0020:000CH  BC1000              MOV  SP,0010H ; + 16T
0020:000FH  2E8E1E0200          MOV  DS,CS:WORD PTR 0002H
0020:0014H  EA0A002100          JMP  (#10)0021H:000AH
0020:0019H  008BECFB            ADD  [BP+DI+0FBECH],CL
0021:000DH  9A00005F00          CALL  005FH:0000H
:FLOATING_POINT_MATH_TEST#11
0021:0012H  B83E03              MOV  AX,033EH ; +830T
0021:0015H  50                  PUSH  AX
0021:0016H  9BD96EFE            FLDCW     WORD PTR [BP-02H]
0021:001AH  58                  POP  AX
.12
0021:001BH  9BDD060000          FLD       DWORD PTR 0000H
0021:0020H  9BDD162400          FST       DWORD PTR 0024H
0021:0025H  9BDD1E2000          FSTP       DWORD PTR 0020H
0021:002AH  9B                  FWAIT
.13
0021:002BH  C606360001          MOV  BYTE PTR 0036H,1
```

Trace buffer display (80286) continued

1. Display the trace buffer in INSTRUCTIONS mode (for brevity, this display is truncated after frame 41H):

```
*PRINT INSTRUCTIONS ALL
FRAME    ADR          BYTE              MNEMONICS  OPERANDS        UNIT 0
001 000206H          FA                   CLI
004 000207H          2E8E160000           MOV  SS,CS:WORD PTR 0000H
        000200H- R-005EH
008 00020CH          BC1000               MOV  SP,0010H ; +16T
00B 00020FH          2E8E1E0200           MOV  DS,CS:WORD PTR 0002H
        000202H- R-0055H
010 000214H          EA0A002100           JMP  (#10)0021H:000AH
:FLOATING_POINT_MATH#10
013 0021:000AH       8BEC                 MOV  BP,SP
015 0021:000CH       FB                   STI
018 0021:000DH       9A00005F00           CALL  005FH:0000H
        0005EEH- W-0021H
01D 0005F0H          CB                   RET        ; FAR
        0005ECH- W-0012H  0005ECH- R-0012H  0005EEH- R-0021H
:FLOATING POINT_MATH#11
023 0021:0012H       B83E03               MOV AX,033EH ; +830T
025 0021:0015H       50                   PUSH  AX
026 0021:0016H       9BD96EFE             FLDCW      WORD PTR [BP-02H]
        0005EEH- W-033EH
02E 0021:001AH       58                   POP  AX
        0000F8H- 0-6ED9H
:FLOATING_POINT_MATH#12
030 0021:001BH       9BD9060000           FLD        DWORD PTR 0000H
        0005EEH-CR-033EH  0000FAH- 0-033EH  0005EEH- R-033EH
0000F8H- 0-06D9H
        0000FCH- 0-001CH  0000FCH- 0-0021H 000550H-CR-0000H
        .....etc.
```

2. Display the trace buffer in CYCLES mode (for brevity, this display is truncated after frame 41H):

```
*PRINT CYCLES ALL
EXEC ADR  BUS ADR   DATA   STATUS  CLIPS FRAME   TIME      LEVEL  UNIT 0
x         b 000206 d 2EFA s 00ED  F c     f 000
x 000206  b        d      s         c 06  f 001   0.0 nanosecs    0
x         b 000208 d 168E s 006D  F c     f 002
x         b 00020A d 0000 s 006D  F c     f 003
x 000207  b        d      s         c 06  f 004   1.4 microsecs   0
x         b 00020C d 10BC s 006D  F c     f 005
x         b 00020E d 2E00 s 006D  F c     f 006
x         b 000200 d 005E s 0065  R c     f 007
x 00020C  b        d      s         c 06  f 008   3.0 microsecs   0
x         b 000210 d 1E8E s 006D  F c     f 009
x         b 000212 d 0002 s 006D  F c     f 00A
x 00020F  b        d      s         c 06  f 00B   5.2 microsecs   0
x         b 000214 d 0AEA s 006D  F c     f 00C
x         b 000216 d 2100 s 006D  F c     f 00D
x         b 000202 d 0055 s 0065  R c     f 00E
x         b 000218 d 0000 s 006D  F c     f 00F
x 000214  b        d      s         c 06  f 010   8.0 microsecs   0
x         b 00021A d EC8B s 006D  F c     f 011
x         b 00021A d EC8B s 006D  F c     f 012
x 00021A  b        d      s         c 06  f 013  10.4 microsecs   0
x         b 00021C d 9AFB s 006D  F c     f 014
x 00021C  b        d      s         c 06  f 015  11.0 microsecs   0
x         b 00021E d 0000 s 006D  F c     f 016
EXEC ADR  BUS ADR   DATA   STATUS  CLIPS FRAME   TIME      LEVEL  UNIT 0
x         b 000220 d 005F s 006D  F c     f 017
x 00021D  b        d      s         c 06  f 018  12.4 microsecs   0
x         b 000222 d 3EB8 s 006D  F c     f 019
x         b 000224 d 5003 s 006D  F c     f 01A
x         b 0005EE d 0021 s 0066  W c     f 01B
x         b 0005F0 d 00CB s 006D  F c     f 01C
```

Trace buffer display (80286) continued

```
x 0005F0 b        d       s          c 06 f 01D   15.4 microsecs   0
x        b 0005EC d 0012  s 0066  W  c    f 01E
x        b 0005EC d 0012  s 0065  R  c    f 01F
x        b 0005EE d 0021  s 0065  R  c    f 020
x        b 000222 d 3EB8  s 006D  F  c    f 021
x        b 000224 d 5003  s 006D  F  c    f 022
x 000222 b        d       s          c 06 f 023   19.8 microsecs   0
x        b 000226 d D99B  s 006D  F  c    f 024
x 000225 b        d       s          c 06 f 025   20.0 microsecs   0
x 000226 b        d       s          c 06 f 026   20.6 microsecs   0
x        b 000228 d FE6E  s 006D  F  c    f 027
x        b 0005EE d 033E  s 0066  W  c    f 028
x        b        d       s          c 06 f 029   21.8 microsecs   0
x        b 00022A d 9B58  s 006D  F  c    f 02A
x        b 00022C d 06D9  s 006D  F  c    f 02B
x        b 00022E d 0000  s 006D  F  c    f 02C
x        b 000230 d D99B  s 006D  F  c    f 02D
EXEC ADR  BUS ADR   DATA   STATUS  CLIPS FRAME   TIME      LEVEL  UNIT 0
x 00022A b        d       s          c 06 f 02E   25.0 microsecs   0
x        b 0000F8 d 6ED9  s 006A  0  c    f 02F
x 00022B b        d       s          c 06 f 030   26.0 microsecs   0
x        b 0005EE d 033E  s 0025 CR  c    f 031
x        b 0000FA d 033E  s 006A  0  c    f 032
x        b 0005EE d 033E  s 0065  R  c    f 033
x        b 000232 d 2416  s 006D  F  c    f 034
x        b        d       s          c 06 f 035   28.8 microsecs   0
x        b 000234 d 9B00  s 006D  F  c    f 036
x        b 000236 d 1ED9  s 006D  F  c    f 037
x        b 000238 d 0020  s 006D  F  c    f 038
x        b 00023A d C69B  s 006D  F  c    f 039
x        b 0000F8 d 06D9  s 006A  0  c    f 03A
```

```
x          b 0000FC d 001C s 006A  0 c     f 03B
x          b 0000FC d 0021 s 006A  0 c     f 03C
x          b 000550 d 0000 s 0025 CR c     f 03D
x          b 0000FA d 0000 s 006A  0 c     f 03E
x          b 000552 d 0000 s 0025 CR c     f 03F
x          b 0000FA d 0000 s 006A  0 c     f 040
x          b 0000FC d 0000 s 006A  0 c     f 041
```

Cross-Reference

PRINT

TRCBUS

A pseudo-variable that controls the collection of bus information in the trace buffer

Syntax

TRCBUS { =TRUE | =FALSE | =*boolean-expression* }

Where:

TRCBUS	displays the setting, either TRUE or FALSE.
TRUE	collects both execution and bus information into the trace buffer.
FALSE	collects only execution addresses into the trace buffer.
boolean-expression	is an expression in which the low-order bit evaluates to 0 (false) or 1 (true).

Default

TRUE

Discussion

The I²ICE system normally collects both execution and bus information in the trace buffer. Typically, three times as many bus cycles are executed as execution cycles. If bus activity is of no particular interest, you can set TRCBUS to FALSE, filling all 1023 usable trace buffer frames with just execution frames.

Display the trace buffer with the PRINT INSTRUCTIONS or PRINT CYCLES command.

Cross-Reference

PRINT
Trace buffer display

TRCREG

Defines a register that contains user program tracing specifications

Syntax

DEFINE TRCREG *name* = { *break-specification* | [SYSTRACE] *system-specification* }

Where:

DEFINE TRCREG *name*	creates a debug trace register called name. A *break-specification* or [SYSTRACE] *system-specification* following the equal sign (=) defines the trace collection criteria. The Break specification and System specification entries in this encyclopedia describe the syntax in detail.
name	is the name of the debug trace register you want to create.
SYSTRACE	specifies that when the *system-specification* is met, any I²ICE units properly enabled are triggered and trace according to the defined criteria. Do not specify SYSTRACE on any unit that also has SYSARM, SYSDARM, or SYSTRIG specified.

Discussion

The I²ICE system normally records (traces) all probe processor bus information when TRCBUS is TRUE. Only execution addresses are traced when TRCBUS is FALSE. The TRCREGs are programmed to selectively collect trace information. By using this debug register, you can specify conditions to the probe that control when trace information is collected.

Defined trace registers (TRCREGs) can only be activated using the optional TRACE keyword in the GO command.

You can optionally enclose a TRCREG specifications in a DO-END block.

Manipulating TRCREGs

You manipulate a TRCREG by referring to its name. You can manipulate TRCREGs in the following ways:

- Create a TRCREG with the DEFINE command

- Delete a TRCREG from memory with the REMOVE command
- List TRCREG names with the DIR command
- Save a TRCREG to a file with the PUT or APPEND commands
- Restore a TRCREG from a file with the INCLUDE command
- Display a TRCREG with the TRCREG command
- Execute a TRCREG with the GO USING command
- Modify a TRCREG with the editor

Restrictions

The TRCREGs may contain any number of specifications. The GO command's ability to execute the specifications, however, is limited by the number of word recognizers available.

Word recognizers are the programmable portion of the internal execution state machine. They compare your match specifications with conditions on the bus they monitor. When a match occurs, the state machine halts emulation. Refer to the Event machines entry in this encyclopedia for details.

Word recognizer use is governed internally. You cannot know precisely how many word recognizers are used in any given specification. A good rule of thumb is one- or two-range (partition) specifications or four-location specifications are the upper limit.

The I²ICE system returns an error when the word recognizer limit is exceeded.

NOTE

Defining new trace specifications using an old TRCREG name destroys the old definition in memory. An error results if you try to assign a TRCREG name to any other debug object in memory.

Restoring a saved TRCREG that has the same name as a TRCREG in memory overwrites the latter.

An error occurs when you try to restore a saved TRCREG that has the same name as any other debug object in memory.

Because TRCREGs are referred to by name, you can reuse break specifications without reentering them. The GO command allows TRCREG lists. You can switch breakpoints in a GO statement by changing TRCREG names.

When using 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

Trace Buffer

Trace information is collected in a 1024-frame buffer in either of two frame types, bus frames or execution frames. A bus frame contains bus addresses, data, and processor status. Bus frames are collected for each bus cycle (read, write, input, output, and fetch). An execution frame contains execution addresses, clips, and timetags. Execution frames are collected every time an instruction is popped from the processor's queue and executed. Trace frames are displayed with the PRINT command.

Example

1. The following example shows how to define a trace register to cause all enabled units to trace when the data value 32H is read from location 106H of the current unit.

```
*DEFINE TRCREG trace_it = DO
**SYSTRACE READ AT 106H IS 32H
**END
*GO FOREVER TRACE trace_it
```

Cross-References

Break specification
Event machines
Name
PRINT
System specification
Trace buffer display

TSS
80286 probe specific

Displays the current task state segment

Syntax

TSS [(*expression*)]

Where:

TSS	displays the current task-state segment.
expression	represents a 16-bit selector value. This option overrides the current selector stored in the task register.

Discussion

The selector, if specified, selects the descriptor table (either the GDT or the LDT) and an offset into the table. If the selector is not specified, the I²ICE system assumes the selector stored in the task register (TR). When you include a selector, that selector must identify the GDT (bit 2 must be 0).

Example

1. Display the current task state segment:

```
*TSS
LINK=0280  SP0 =0400  SS0 =0028  SP1 =0000  SS1 =0000
SP2 =0000  SS2 =0000  IP  =3592  FL  =0046  AX  =0000
CX  =0024  DX  =001C  BX  =FF56  SP  =0400  BP  =0006
SI  =FF5A  DI  =0030  ES  =0038  CS  =0020  SS  =0028
DS=0028  RLDT=0000
```

Note that the selector part of the local descriptor table register is called RLDT when the task state segment is displayed.

Cross-Reference

80286 registers
Expression
Multitasking

UNIT

A pseudo-variable that displays or changes the current default unit

Syntax

UNIT [= *unit-number*]

Where:

UNIT	displays the current default unit.
unit-number	changes the default unit. The *unit-number* is an expression that evaluates to 0, 1, 2, or 3.

Default

0

Discussion

The UNIT command changes the default unit for the system. All commands are directed to the default (current) unit unless a command is specifically referred to another unit with the backslash (\) control.

The default unit is always numbered 0. The chassis located first in the system cable chain is the default unit.

Example

1. Set I²ICE chassis number 1 as the default unit:

```
*UNIT = 1
```

Cross-Reference

Expression

UNITHOLD

Causes the I²ICE system to pause while the user cable is moved

Syntax

UNITHOLD [*unit-number*[,*unit-number*]* | ALL]

Where:

UNITHOLD	suppresses the error displayed when the user cable is disconnected from a running unit. Entering any character restores normal error detection.
unit-number	is a number of the unit you want to hold (0, 1, 2, or 3) or an expression that evaluates to 0, 1, 2, or 3.
ALL	holds all units.

Discussion

In software (loopback) mode, the user cable is plugged into the loopback socket on the probe buffer box. In hardware mode, the user cable is plugged into the user prototype. Normally, the system issues an error message if the user cable is not connected to either the user system or to the software-only socket on the buffer box. With the UNITHOLD command you can change prototypes and switch between hardware and software modes. Use the UNITHOLD command before you disconnect the user cable while the I²ICE software is running.

Terminal interaction pauses while UNITHOLD is active. Enter any character to restore normal error detection.

UNITHOLD causes the 80286 probe to 3-state all signals on the user cable. For the 8086/8088 and 80186/80188 probes, UNITHOLD 3-states most of the signals on the user cable. This effect of the UNITHOLD command permits the safe transfer of the user cable between the loopback socket and the target system while I²ICE system power remains on.

Cross-Reference

Expression

VERSION

Displays the version numbers of the host and probe software

Syntax

VERSION [*unit-number*[,*unit-number*]* | ALL]

Where:

VERSION	displays the host software version number.
unit-number	is the number of the probe whose software version number you want to display (0, 1, 2, or 3) or an expression that evaluates to 0, 1, 2, or 3.
ALL	displays the version number of all probes' software.

Discussion

The VERSION command displays the version numbers of the software for the host and any probes connected to the system. If a probe is emulating when you enter the VERSION command, the system returns a message that the probe is emulating rather than returning the version number.

WAIT

A function that suspends command execution during emulation

Syntax

```
WAIT
```

Where:

WAIT	returns a device code indicating which I²ICE probe or iLTA caused the break.

Discussion

The WAIT function prevents the I²ICE system from accepting terminal commands until any emulating unit executes a breakpoint or the iLTA (Intel logic timing analyzer) completes data collection.

You must enter a WAIT command for every unit in operation. For example, if you have two probes emulating, enter two WAIT commands.

When a break occurs, the WAIT function returns the device code, in the current base, of the unit causing the break. The codes and their definitions are listed in Table 1-39.

Table 1-39 Decimal Device Codes for the WAIT Function

Code	Definition
0	I²ICE Chassis 0
1	I²ICE Chassis 1
2	I²ICE Chassis 2
3	I²ICE Chassis 3
4	iLTA Chassis 0
5	iLTA Chassis 1
6	iLTA Chassis 2
7	iLTA Chassis 3
255	No device emulating

NOTE: These table values are displayed in the current base.

The WAIT function depends on a break or trigger event to execute commands. This is particularly useful in debug procedures written for multiple unit control.

NOTE

When specifying a WAIT inside a debug procedure, the WAIT command must be followed by the CAUSE command for the display to be properly formatted.

Example

1. This example defines a debug procedure, named runmany, that starts two units emulating the same program. The procedure runmany suppresses the break message with a REPEAT loop until both probes have halted emulation. The REPEAT loop will execute as long as there is at least one probe emulating.

```
*DEFINE PROC runmany = DO
. *UNIT = 0;
. *GO TIL exit_point                    /*Begin emulating on unit 0*/
. *UNIT = 1
. *GO TIL exit_point                    /*Begin emulating on unit 1*/
. *REPEAT
. . *UNTIL (WAIT = = 255T)              /*Wait until both probes have broken*/
. . *CAUSE                              /*Required CAUSE command*/
. . *ENDREPEAT
. *END
*runmany                                /*Execute PROC*/
```

WAITSTATE

A pseudo-variable that controls
the number of memory wait-states
inserted by the I²ICE system

Syntax

WAITSTATE [= *expression*]

Where:

WAITSTATE	displays the current setting.
expression	is any numeric expression that evaluates to a BYTE value from 0 to 15 (decimal).

Default

0

Discussion

Program memory can run with no wait-states (i.e., no extra clock cycles), or you can specify the number of wait-states to simulate slower memories. The initial value is zero wait-states. Setting WAITSTATE to a value other than zero affects all program memory, including memory mapped to USER. If *expression* is greater than 15, WAITSTATE is set to 15 and a message is displayed to tell you that WAITSTATE is set to 15.

Examples

The following examples assume decimal base.

1. Display the current setting:

```
*WAITSTATE
0
```

2. Change the setting:

```
*WAITSTATE = 10
```

3. Use WAITSTATE as a variable:

```
*WAITSTATE = WAITSTATE + 5
```

Cross-Reference

Expression

WORD

Displays or changes memory
as 16-bit unsigned values

Syntax

WORD *partition* [= *expression* [, *expression*]* | = *mtype partition*]

Where:

WORD *partition*	displays the contents of memory in *partition* as a WORD in the current base.
partition	is a single address or a range of addresses specified as *address* TO *address* or *address* LENGTH *number-of-items*.
expression	converts to a 16-bit unsigned value for WORD.
mtype	is any of the memory types except ASM.

Discussion

The WORD command interprets the contents of memory as 16-bit unsigned values, overriding any type associated with the memory contents. Thus, WORD .var1 displays the first two bytes at the address of var1, regardless of the type of var1.

The information displayed by the WORD command is identical to that displayed by the ADDRESS and SELECTOR commands. However, when the memory type WORD is used as a data type in a program, it is interpreted as a 16-bit unsigned value. Both the ADDRESS and SELECTOR types, in that context, are interpreted as segments of address pointers.

NOTE

The I²ICE system writes a word value in two byte values (uses two bus cycles).

Examples

The following examples assume a hexadecimal base.

1. Display a single value:

```
*WORD $
0020:0004H 2EFA
```

2. Display several adjacent values:

```
*WORD $ LENGTH 6
0020:0004H 2EFA 168E     0 72BC 2E00 1E8E
```

3. Set a single value of type WORD:

```
*WORD 40:4 = 34AF
```

4. Set several contiguous values:

```
*WORD 40:4 = 10FA, 3045, 107F
```

Display the values set:

```
*WORD 40:4 LENGTH 3
0040:0004H 10FA 3045 107F
```

5. Set a range of locations to the same value (block set):

```
*WORD 40:4 LENGTH 10 = 0
```

6. Set a repeating sequence of values:

```
*WORD 40:4 LENGTH 10 = 1234, 5678, 9ABC, 0DEF0
```

Display the values set:

```
*WORD 40:4 LENGTH 6
0040:0004H 1234 5678 9ABC DEF0 1234 5678
```

7. Copy a value from one memory location to another:

```
*WORD 40:4 = WORD $
```

8. Copy several values (block move):

```
*WORD 40:4 = WORD $ LENGTH 10
```

9. Copy values with type conversion:

```
*WORD 40:4 = BYTE .var2
```

An error message occurs if the type on the right side of the equal sign cannot be converted to the type on the left. (Refer to the Expression entry in this encyclopedia for the rules concerning type conversions.)

Cross-References

Expression
Mtype
Partition

WPORT

A pseudo-variable that displays or changes the contents of word-wide I/O ports

Syntax

WPORT(*port-number*) [= *data*]

Where:

WPORT(*port-number*)	displays the contents of the specified word-wide I/O port in the current base. The *port-number* is a number or expression that specifies one of the I/O ports in the range 0000H to OFFFFH.
data	writes a word of data to the specified port.

Discussion

If the I/O reference (*port-number*) is mapped to an 80186/80188 internal peripheral control register, the register is transparently accessed. There is no protection against writing a read-only control register.

Note that the I²ICE system displays the output word in hexadecimal regardless of the current radix.

Examples

1. Read a word-wide port:

```
*WPORT(0123H)
0A12
```

2. Write a word-wide port:

```
*WPORT(0123H) = 5556
```

Cross-References

Expression
PORT

WRITE

Displays and formats character strings and numerical expressions

Syntax

WRITE [SCREEN (*x,y*)] [USING(*'format-item* [,*format-item*]* *'*)] *write-list*

Where:

WRITE *write-list*	displays the items in *write-list* to the terminal. The *write-list* consists of *write-items*, separated by commas. The *write-items* are ASCII character strings or numerical expressions. The ASCII character strings must always be enclosed in apostrophes (*'*). The number of *write-items* is limited only by the size of the line buffer.
SCREEN (*x,y*)	writes the *write-list* to a specified (*x,y*) coordinate location on the display screen. After the write, the cursor returns to its previous location.
(*x,y*)	are the screen coordinates. The upper left corner is location (0,0). The columns (*x*) are numbered 0 to 79. Rows (*y*) are numbered 0 to 24 on the Intel Series III/IV and IBM PC terminals. The *x* is any integer or expression that evaluates to 0 through 79. The *y* is any integer or expression that evaluates to 0 through 24.
USING (*'format-item*, [*format-item*]* *'*)	formats the display according to a list of one or more *format-items*.
format-item	is one of the following:
n	is a decimal number specifying the width of the output field. If *n* = 0 and the radix is binary or hexadecimal, the length used is the normal display length of the item without padding or truncation. If the radix is decimal, 0 specifies that the output be left-justified. If you choose any other number, that number directly determines the maximum width of any display item.
m.n	specifies the width of the output field for a real number. The *m* is the total number of characters, including the decimal point. The *n* is the number of digits to the right of the decimal point. Both *m* and *n* are entered in decimal.

[*n*]C	moves the output pointer to column *n*. The next item, if any, is written from that point. Columns are numbered 1 to 80. The C (without *n*) moves the pointer to the next column. The *n* is in decimal.
[*n*]X	writes *n* spaces between items. The X (without *n*) writes one space.
H	writes numerical items in hexadecimal (overrides the default number base).
T	writes numerical items in decimal (overrides the default number base).
Y	writes numerical items in binary (overrides the default number base).
.	terminates a format string (optional). If the list contains undisplayed items, they remain undisplayed.
>	terminates a format string and specifies that a carriage return and line feed are not to be issued following the write. If the list contains undisplayed items, they remain undisplayed.
&	terminates a format string and specifies that the write output buffer is not to be flushed at the end of the write. Later writes are added to the one in the buffer. If the list contains undisplayed items, they remain undisplayed.
"text"	inserts ASCII text in the format. In this context only, you must enclose *"text"* in quotation marks (").

Discussion

The WRITE command is most often used in procedures to add explanatory text to returned values or to write returned values in a more useful form, such as a table.

The WRITE command displays a maximum of 200 bytes of data. An ASCII character is one byte of data. Spaces, carriage returns, and line feeds count as characters. The byte content of numerical expressions depends on the memory type required to store the number. If you try to write more than 200 bytes of data, an exclamation point (!) is displayed at the end of the line and you cannot see the rest of the information.

Unless specified in the format string by the continuation symbol (&), the information in the write buffer is deleted at the end of every write. Even if you specify the continuation symbol, the write buffer is deleted on a return from a procedure.

WRITE continued

If the *write-list* contains more items than are specified by the format string, the format string is reused from the beginning, until all *write-items* are displayed according to the format.

Examples

1. Write a character string to the display screen:

```
*WRITE 'hello'
hello
```

2. Format a display:

```
*DEFINE BYTE a = 45
*DEFINE BYTE b = 67
*DEFINE BYTE c = 22
*WRITE USING ('2,H,2x') a,b,c
45 67 22
```

3. The following example shows the WRITE USING option. The procedure SQUAREIT squares a number the user specifies at the time the procedure is called (%0).

```
*DEFINE PROC squareit = DO
. *WRITE USING ('"The square of",X,T,0,X,"is ",X,T,0') %0, %0*%0
. *END
```

Calling the procedure and specifying the number:

```
*squareit(7)
The square of 7 is 49
```

4. The following procedure SQR squares a series of numbers from 0 to a number (%0) the user specifies at the time the procedure is called. The display format is set up as a table with headers.

```
*DEFINE PROC sqr = DO
. *WRITE USING ( ' "Number",20C,"Square" ')
. *DEFINE BYTE b = 0
. *REPEAT
. . *UNTIL b = = %0 + 1
. . *WRITE USING ( '2X,T,2,22C,T,3 ') b,b*b
. . *b = b + 1
. . *END
. *END
*sqr(5)

Number    Square
   0          0
   1          1
   2          4
   3          9
   4         16
   5         25
```

Cross-References

Expression
Strings

XCTR

A pseudo-variable that assigns a value to the execution event machine counter

Syntax

XCTR [= *unsigned-integer-expression*]

Where:

XCTR	displays the value of the execution event machine (XEM) counter prior to emulation. There is no default value; XCTR is random at power on.
unsigned-integer-expression	is a number or expression that evaluates to a positive whole number in the current base.

Discussion

The XCTR command displays what the value of the system event machine (SEM) counter will be when emulation is initiated. It does not display the current value.

There are two methods to set the system event machine (SEM) counter: when defining an EVTREG or by using the XCTR command. If a counter value is specified in an EVTREG invoked with a GO command, the EVTREG value replaces any previously specified XCTR value.

The XCTR command is useful when you must change the counter value from a new emulation or you forget to specify it in the EVTREG definition. The XCTR command is effective only when used just before invoking an event register specification that does not specify a counter value for the execution event machine (XEM).

Example

1. The following example shows how to set XCTR for execution. The EVTREG breaks emulation five execution addresses after the first occurrence of execution address 12.

```
*XCTR = 5
*DEFINE EVTREG count_change = DO
**XEM S0 IF 12 THEN INCREMENT AND GOTO S1
**S1 IF ENDCNT THEN BREAK BUT ALWAYS INCREMENT END
*GO USING count_change
Probe 0 stopped at :module #17 because of execute break
```

Cross-Reference

Expression

2 Error Messages

The five classes of errors that the I²ICE system reports are as follows:

WARNING	The I²ICE system takes no action and command processing continues. Warnings advise you of a possible error condition.
ERROR	The I²ICE system stopped processing the current command. The prompt reappears, indicating that you should try again. Memory may be altered.
SEVERE ERROR	The I²ICE system closes all INCLUDE files and returns a prompt to the terminal. Memory may be altered.
FATAL ERROR	Non-recoverable error. Control returns to the host operating system. Memory may be altered.
INTERNAL ERROR	Indicates an internal software problem. You should contact Intel's service organization. Memory may be altered.

All error messages have the following format:

[device-number] severity-level # number
message [*]

Where:

device-number	is the probe number P86, P186, or P286. When *device-number* is not present, the error pertains to the host development system.
severity-level	is warning, error, severe error, fatal error, or internal error.
# number	is the decimal error message number.
message	is the text of the error. If the ERROR command is set to FALSE, the error message display is suppressed.

Messages followed by a [*] have extended messages. The extended message is displayed on-line with the HELP command (see the HELP entry in Chapter 1).

Note that error numbers are duplicated; that is, a host error can have the same number as a probe error.

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 . . .

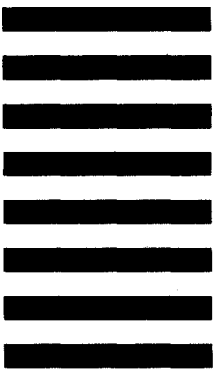
This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

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

0688/5K/0885/SOS/AD