intel®

DEBUG-88
USER'S MANUAL

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

Order Number: 121758-003

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

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

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.

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.

The following are trademarks of Intel Corporation and its affiliates and may be used to identify Intel products:

AEDIT	iDIS	Intellink	MICROMAINFRAME
BITBUS	iLBX	iOSP	MULTIBUS
BXP	i_m	iPDS	MULTICHANNEL
COMMputer	iMMX	iRMX	MULTIMODULE
CREDIT	Insite	iSBC	Plug-A-Bubble
i	int_el	iSBX	PROMPT
I^2ICE	int_elBOS	iSDM	Ripplemode
iATC	Intelevision	iSXM	RMX/80
ICE	int_eligent Identifier	Library Manager	RUPI
iCS	int_eligent Programming	MCS	SYSTEM 2000
iDBP	Intellec	Megachassis	UPI

A985/783/ 4K DD

REV.	REVISION HISTORY	DATE
-001	Original issue.	11/82
-002	Alter preface.	1/83
-003	Incorporate change package.	6/83

PREFACE

This manual provides operating instructions for DEBUG-88, a symbolic debugger for use with 8086/8088 programs executed on the Intellec Series IV Microcomputer Development System.

Operation of this system requires version 1.0 or later of the iNDX operating system.

This manual has seven chapters and two appendixes:

- Chapter 1, "Introduction," introduces the DEBUG-88 symbolic debugger.
- Chapter 2, "Simplified Operation," presents easy to follow instructions for a limited set of DEBUG-88 commands that you can enter to obtain quick "hands-on" experience in using the debugger.
- Chapter 3, "Command Format Notation," describes the character set, editing rules, classes of expressions, symbol names, operators and logical and semantic rules employed by DEBUG-88.
- Chapter 4, "Utility Commands," describes the DEBUG-88 file management commands.
- Chapter 5, "Execution Commands," describes the DEBUG-88 commands used to transfer control between DEBUG-88 and user programs, to specify operational mode and to set breakpoints.
- Chapter 6, "Simple Commands," describes the remaining DEBUG-88 simple commands, i.e., those used to set and display registers, flags, memory locations, and the stack.
- Chapter 7, "Compound Commands," describes the DEBUG-88 commands that allow nesting of other commands.
- Appendix A, "Error Messages," lists in numerical order the error messages that are produced by improper use of DEBUG-88.
- Appendix B, "Example of a DEBUG Session," presents a log of a debug session for a sample program.

Related Publications

For more information on the Series IV Microcomputer Development System, see the following manuals:

- *Intellec Series IV Microcomputer Development System Overview*, 121752
- *Intellec Series IV Operating and Programming Guide*, 121753
- *Intellec Series IV ISIS-IV User's Guide*, 121880

Notational Conventions

UPPERCASE	Characters shown in uppercase must be entered in the order shown. You may enter the characters in uppercase or lowercase.

italic	Italic indicates a meta symbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:
filename	Is a valid name for the part of a *pathname* that names a file.
pathname	Is a valid designation for a file; in its entirety, it consists of a *directory* and a *filename.*
pathname1, pathname2, ...	Are generic labels placed on sample listings where one or more user-specified pathnames would actually be printed.
directory-name	Is that portion of a *pathname* that acts as a file locator by identifying the device and/or directory containing the *filename.*
system-id	Is a generic label placed on sample listings where an operating system-dependent name would actually be printed.
V*x.y*	Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.
[ ]	Brackets indicate optional arguments or parameters.
{ }	One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.
{ } . . .	At least one of the enclosed items must be selected unless the field is also surrounded by brackets, in which case it is optional. The items may be used in any order unless otherwise noted.
|	The vertical bar separates options within brackets [ ] or braces { }.
...	Ellipses indicate that the preceding argument or parameter may be repeated.
[,...]	The preceding item may be repeated, but each repetition must be separated by a comma.
punctuation	Punctuation other than ellipses, braces, and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered: `SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')`
`input lines`	In interactive examples, user input lines are printed in white on black to differentiate them from system output.
`<cr>`	Indicates a carriage return.

CONTENTS

TABLES

FIGURES

CHAPTER 1
INTRODUCTION

DEBUG-88 provides symbolic debugging of user developed 8086/8088 programs. The DEBUG-88 English language command set enables you to:

- Initialize DEBUG-88 and load your program's Symbol Names and Line Numbers in the DEBUG-88 symbol table and clear all break registers.
- Set starting and stopping points for execution of your program.
- Execute your program in either continuous or single step mode.
- Set, display and alter 8086/8088 Registers, Flags, Memory Locations and Stack Content.
- Display the contents of Memory Locations as Disassembled Instructions.
- Search for and display User Defined Program Labels and Line Numbers.

CHAPTER 2
SIMPLIFIED OPERATION

This chapter presents easy to follow instructions that allow you to enter a simplified subset of the full DEBUG-88 commands at the console exactly as they are shown in the following examples. Subsequent chapters provide detailed explanations of the complete DEBUG-88 command set. Those chapters will explain the full power and flexibility available to you with DEBUG-88. Readers unfamiliar with the use of a debugger should first master the simplified commands given in this chapter. After you have become comfortable with the use of the simplified command set, proceed to the later chapters. More sophisticated users may prefer to skip over this chapter and instead go directly to the more detailed sections of this manual.

ENTRY	RESULT
`D88`	Invokes DEBUG-88. System responds by sending the following message to the console: Series IV DEBUG 8O88, Vx.y, where *x*, *y* is replaced by the current version and level numbers of DEBUG. The system prompt, an asterisk (*), now appears each time DEBUG-88 requests you to enter a command.
`*LOAD "name"`	Loads symbol names and line numbers for your program. For ***name***, enter the filename of your program. Note that ***double quotes*** must surround the filename.
`*REG`	Displays the present contents of all registers.
`*FLAG`	Displays the present contents of all flags.
`*STACK 10`	Displays ten decimal words from the top of the stack. Other integer values may be substituted for 1O.
`*BR`	Displays the present values of both Break Registers (BR0 and BR1). Break registers hold a user program address at which, when reached, user program ceases and control returns to the debugger.
`*BR0=value`	Sets Break Register 0 equal to the new value. ***value*** can be an address or a symbol name, i.e., code address not data port. Note: ensure that the breakpoint is at an address that will be reached.
`*BR1=value`	Sets Break Register 1 equal to the new value. ***value*** can be an address or a symbol name. Note: ensure that the breakpoint is at an address that will be reached.
`*GR=TIL BR0`	Activates Break Register 0.
`*GO`	Starts execution of your program. Execution terminates when BR0 is reached.
`*STEP`	Executes your program in single step fashion.

(CONTROL-D) Simultaneously depress the Control Key and the D key	Suspends execution of your program and returns control to DEBUG-88. Use this escape in case you fail to reach a breakpoint.

A very useful routine is one that displays each successive memory location in disassembled form, i.e., one that displays the Assembly Language Mnemonic Code for the instruction.

*ASM CS:IP	Display instructions as a mnemonic.

CHAPTER 3
COMMAND FORMAT NOTATION

Command Categories

Invocation and the Utility Commands

D88	Invoke DEBUG-88.
EXIT	Unconditionally exit DEBUG-88.
LIST	Create an output listing of user-DEBUG-88 interaction.
LOAD	Load user generated program code.

Execution Commands

BR	Set/Display the contents of one or more Break Registers.
GR	Set/Display the GO Register, activate Break Points.
GO	Transfer control and execute user program.
STEP	Execute user program in Single Step Mode.
PSTEP	Execute user program in Single Step Mode and side step any CALLs.

Simple Commands

Set/Display Commands

FLAG	Set/Display the contents of one or more Flags.
REGISTER	Set/Display the contents of one or more Registers.
STACK	Set/Display the contents of the Stack.
ASM	Display memory as a disassembled instruction.
EVALUATE	Evaluate the variable expression as an integer or a pointer.
EVALUATE SYMBOL	Compare the variable expression against user assigned symbols and symbols created by the Define command.
EVALUATE LINE	Compare the variable expression against user assigned line numbers.
LINE	Display all Line Numbers found in the load file.
MODULE	Display all Module Names found in the load file.

SYMBOL	Display all Symbols found in the unnamed module created by the Define command and all symbols found in the load file.
RADIX	Establish a Radix (Numerical Base) to be used by other commands. Display the current radix value.

Symbol Manipulation Commands

DEFINE	Place a newly defined symbol in the Unnamed Module.
DOMAIN	Set the module in which to begin a search for user defined symbols.
REMOVE	Remove an existing user defined symbol from the Unnamed Module.

Compound Commands

COUNT	Set up a loop to be executed, at most, the specified number of times.
IF	Provide for conditional execution of subsequent commands.
REPEAT	Set up an infinite loop.

Character Set

The valid character set for the DEBUG-88 command language consists of the ASCII characters 20 hex to 7E hex, inclusive, shown in table 3-1.

In addition, the characters Return and Escape are accepted.

Tokens possessing meaning to DEBUG-88 consist of one or more valid characters. Entering invalid characters results in Error Messages.

Invoking DEBUG-88

To invoke DEBUG-88 enter the characters "D88." The system responds by outputting to the console the reply:

```
SERIES IV DEBUG-88, Vx.y
```

where

x and *y*	represent the current version and level numbers of DEBUG-88.

Entering Commands

Use your system console device to communicate interactively with DEBUG-88. DEBUG-88 displays an asterisk (*) in the left margin as the system prompt. The prompt indicates that DEBUG-88 is awaiting a user input. Each input line can contain up to a maximum of 120 characters. Terminate each input by depressing RETURN. The symbol ‹cr› will be used in this manual to represent the return key.

Continuation Lines

If you need to extend an input line, use the ampersand (&) as a line continuator. Characters on a given line that are entered after (i.e., to the right of) the ampersand are disregarded. Two consecutive asterisks (**) are used to designate the beginning of a continuation line. For example:

```
*GO FROM BUG3 &
**TILL .LAB2
```

is equivalent to

```
*GO FROM BUG3 TILL .LAB2
```

Comments

Comments may be appended to any input command line. New users of DEBUG-88 will find that using comments significantly increases their understanding. To append a command line, enter a semicolon (;) after the command. DEBUG-88 ignores all characters that appear between the semicolon and the carriage return (RETURN). It is also acceptable to enter a line that consists entirely of comments by entering a semicolon as the first character of the line. Note that an ampersand following a semicolon is treated as part of a comment and not as a line continuator. If on a given command line you wish both to extend to a second line and to include comments on the original line, be sure to enter the ampersand before (i.e., to the left of) the semicolon.

Line Editing

If while entering a command (or a comment) you strike an incorrect key, backspacing the cursor (with the rubout key) over the entered character will erase that character. With the cursor in the position you desire, simply strike the key for the correct character.

Interrupting Program Execution

You can unconditionally escape from execution of the user program and return to DEBUG-88 by simultaneously striking the Control key and the D key (CONTROL-D). Striking CONTROL-C, on the other hand, provides an unconditional escape from the user program to the operating system.

Error Conditions and Error Messages

Syntax errors that appear in a command line cause an error message to appear on the console device. Error messages are in the form:

```
ERROR XX
```

where

XX is a decimal number that defines the error.

Refer to Appendix A for a numerical list of all DEBUG-88 error messages.

Expressions

Expressions are entered as arguments in command lines. The expressions specify either numeric values or Boolean conditions. Each expression evaluates to a number that can be either of two types—

- Pointer—a pair of sixteen bit unsigned integers separated by a colon. The first (i.e. left) integer of the pair is the base address; the second is the displacement from the base.
- Word—a single sixteen bit unsigned integer. A word can be considered as a special case of the pointer, i.e., one having a base equal to zero.

DEBUG-88 provides unsigned integer arithmetic. It does not provide signed arithmetic. The arithmetic operations are performed on the offsets, with the result taking as its base the base of the leftmost term.

The following are examples of typical expressions—

- Expressions that consist of only a single value:

  3
  0FFFFH
- Expressions that contain operands, operators and parentheses:

  2 + 3
  174/4
  (127 + 44)/20
  0100H + 00FFH
- Expressions that contain symbols:

  .SYMBA − 2
  .SYMBA + 3

Operands

You can reference the following types of operands—

- Numeric Constants
- Command Keywords
- Keyword References
- Memory References
- Port References
- Symbolic References
- Statement References
- String Constants

Numeric Constants

A numeric constant produces a fixed, unsigned, 16-bit integer value that consists of the digits 0 through 9 and A through F (i.e., the supplementary hexadecimal digits). Hexadecimal numbers starting with A-F require a prefix of 0 (zero) to distinguish them from symbols. The suffix H that follows the digits indicates the hexadecimal base. If on entering a digit you do not append an H (or an h), hexadecimal is still assumed as a default condition unless a different base was explicitly set earlier. The other bases (with corresponding suffixes shown in parenthesis) are: Decimal (T), Octal (Q) and Binary (Y). Numerical constants that exceed the largest representable 16-bit unsigned value are truncated to the 16 least significant bits.

Table 3-1. ASCII Printing Characters and Codes (20H-7EH)

Character	Hex Code	Character	Hex Code	Character	Hex Code
Space(SP)	20	@	40	`	60
!	21	A	41	a	61
"	22	B	42	b	62
#	23	C	43	c	63
$	24	D	44	d	64
%	25	E	45	e	65
&	26	F	46	f	66
'	27	G	47	g	67
(	28	H	48	h	68
)	29	I	49	i	69
*	2A	J	4A	j	6A
+	2B	K	4B	k	6B
,	2C	L	4C	l	6C
−	2D	M	4D	m	6D
.	2E	N	4E	n	6E
/	2F	O	4F	o	6F
0	30	P	50	p	70
1	31	Q	51	q	71
2	32	R	52	r	72
3	33	S	53	s	73
4	34	T	54	t	74
5	35	U	55	u	75
6	36	V	56	v	76
7	37	W	57	w	77
8	38	X	58	x	78
9	39	Y	59	y	79
:	3A	Z	5A	z	7A
;	3B	]	5B	{	7B
<	3C	/	5C	|	7C
=	3D	[	5D	}	7D
>	3E		5E	~	7E
?	3F		5F		

Operators

Operators define the arithmetic or the operation to be performed on one or more operands. Operators are of two types: unary—those that require only one operand, and binary—those that require two operands. Table 3-2 summarizes the operators and lists their precedence (order of operation) level. Operators are executed from lowest numbered precedence to highest.

The Arithmetic Operators are Unary + and −, and Binary +, −, *, /, MOD, and :.

Unary + and −. Unary + is the default condition. Unary − produces the 2's complement modulo 65536 (i.e., −N evaluates to 65536−N). Unary − does not apply to pointers.

Binary + and −. Binary + applies both to pointers and to words. When a word is added to a pointer, the pointer displacement is summed with the word. This produces a result in modulo 65536 form (i.e., bits above Bit 16 are dropped). Binary + does not affect the base value of the pointer. When two pointers are added, the operation is applied to the offsets and the result takes the base of the lefthand term.

Binary − applies both to pointers and to words. The result of the binary − operation is the arithmetic difference of the two operands. When both operands are words, the result is the two's complement of their difference, modulo 65536. A negative result (i.e., −N) is treated as 65536 − N.

Table 3-2. Operators

Precedence	Operator	Operation	Result	Remarks
1	:	a:b	a:b	
2	+	+a	a	
	−	−a	−a	two's complement of a.
3	*	a*b a:b*x:y	(a*b) mod 65536 a:((b*y) mod 65536)	
	/	a/b a:b/x:y	(a/b) a:(b/y)	Division by 0 is reported as an error.
	MOD	a MOD b a:b MOD x:y	a MOD b a:(b MOD y)	Division by 0 is reported as an error.
4	+	a+b a:b+x:y	(a+b) mod 65536 a:((b+y) mod 65536)	
	−	a−b a:b−x:y	(a−b) mod 65536 a:((b−y) mod 65536)	
5	Content	Operators	See Table 3-3.	
6	=	a=b a:b=x:y	a=b a*16+b=x*16+y	All comparisons are done on unsigned quantities. If the comparison succeeds, TRUE (FFh) is returned; if not, FALSE (0h) is returned.
	>	a>b a:b>x:y	a>b a*16+b>x*16+y	
	<	a<b a:b<x:y	a<b a*16+b<x*16+y	
	<>	a<>b a:b<>x:y	a<>b a*16+b<>x*16+y	
	>=	a>=b a:b>=x:y	a>=b a*16+b>=x*16+y	
	<=	a<=b a:b<=x:y	a<=b a*16+b<=x*16+y	
7	NOT	NOT a	bitwise one's complement of a	
8	AND	a AND b	bitwise ANDing of a with b	
9	OR	a OR b	bitwise ORing of a with b	
	XOR	a XOR b	bitwise eXclusive ORing of a with b	

NOTES:

1. The notation **a:b** denotes the construction of a pointer that uses the word **a** as the base and the word **b** as the offset. The same thing happens for **x:y**.

2. In the column labelled "Operation," **a**, **b**, **x** and **y** represent unsigned 16-bit word quantities. If a data item is of byte length (e.g., of data type BYTE, BOOLEAN, etc. is used instead of a word), the byte is sign-extended with high order zeroes to make it of word length. If a pointer is used instead of a word, the pointer is converted into a word by discarding its base (e.g., 0481:02A5 becomes 02A5).

3. If a binary operator has one word operand and one pointer operand, the word operand is made into a pointer operand prior to the operation by using the word as the offset of a pointer whose base equals 0 (e.g., 5 becomes 0000:0005).

4. MOD is the Operation MOD, while mod indicates that the result is a number represented in modulo fashion, with 65536 the modulus.

When a word is subtracted from a pointer, it is subtracted from the pointer displacement only. The base of the pointer is not affected. The result of the operation is in 2's complement form. When a pointer is subtracted from another pointer, only the offsets are subtracted and the result takes the base of the lefthand term. The result is the 2's complement difference of the two displacements, modulo 65536.

Binary *, /, MOD and :. These operators apply only to word operands and produce a word result. Binary * causes multiplication of word operands. If the result exceeds the 16 bit maximum, the excessive high order bits are truncated.

Binary / causes the word on the left to be divided by the word on the right. The result is the quotient only; any remainder is lost. Thus, 5/3 evaluates to 1.

Binary MOD produces only the remainder of a division operation. Thus, 5 MOD 3 evaluates to 2.

The colon(:) causes the word on the left to be treated as a base value that will be displaced by the word on the right. The base of the pointer is shifted one hexadecimal position to the left (i.e., multiplied by 16) before the operation is carried out. Thus, 0481:22=4810 + 0022=04832.

When binary *, / and MOD are applied to two pointers, the operations are performed on the offsets and the result takes the base of the leftmost term.

Content Operators

Content operators fetch the contents of one or more memory addresses. In an expression, content operators function as unary operators. Their precedence is directly below that of binary subtraction (i.e., content operators are evaluated after subtraction but before any relational operator). Table 3-3 defines the seven content operators. .VAR refers to the address of variable VAR. If you enter .VAR, DEBUG-88 will return the address of the variable. If you enter WORD .VAR, DEBUG-88 will return the contents of the WORD pointed at by .VAR. You may consider that the single period and double period prefixes (. and ..) signify the "address of " the operator that is being requested or used.

The single exclamation mark (!) and double exclamation mark(!!) prefixes are used to explicity dereference a symbol. Dereferencing means it is *not* to be considered a reserved word. Thus, GO is a reserved word indicating a command. !GO is a user defined symbol.

Table 3-3. Content Operators

Operator	Content Returned
BYTE	1-byte unsigned integer value from the addressed location in user memory. It is displayed in unsigned form.
WORD	2-byte unsigned integer value from the addressed location in user memory. It is displayed in unsigned form.
SINTEGER	Same as BYTE, but displayed in two's complement form.
BOOLEAN	Same as BYTE, but displayed as a Boolean value (i.e., TRUE if least significant bit = 1; FALSE if it = 0.
INTEGER	2-byte integer value from addressed location in user memory. It is displayed in 2's complement form.
POINTER	4-byte pointer value from the addressed location in user memory.
PORT	1-byte value from addressed 8-bit I/O port.
WPORT	2-byte value from addressed 16-bit I/O port.

Relational Operators

The Relational Operators are: NOT, AND, OR XOR, =, >, <, <> (not equal), >= and <=. The operators carry their usual Boolean definitions.

Arithmetic and Logical Semantic Rules

The semantic rules that govern the operators are summarized in Table 3-2. All operators result in a numerical value. For Boolean operations, the least significant bit (LSB) of the result is tested. The LSB = 1 for TRUE results; the LSB = 0 for FALSE results.

Command Keywords

Command keywords are system-reserved words that are used to designate the command functions. Examples are LOAD, GO and STEP.

Keyword References

Keyword references are system-reserved words that are used to designate flags, registers, and memory. Case is unimportant in keywords; e.g., LOAD, LoAD, and load all reference the same command.

Keyword references can be used in three ways—

- In an expression to be evaluated, e.g., RAX + 5
- Alone as an entry, e.g., RAX ‹cr›
- On the left side of an equal sign, e.g., RAX = 55H ‹cr›. The Keyword Reference is assigned the new value represented by the right side of the equal sign.

If the value referenced occupies less than 16 bits, it is automatically right justified, and the higher order bits are filled with zeroes. If the value exceeds 16 bits, the higher order bits are truncated and lost.

Reserved Words

All system-reserved words are listed below. A name that does not appear in the list, or that is not a valid abbreviation of one of these names as explained in the abbreviation rule stated below, is considered to be a user generated label.

A, AFL, AH, AL, AND, ASM, AX, BH, BL, BOOLEAN, BP, BR, BR0, BR1, BX, BYTE, C, CFL, CH, CL, COUNT, CS, CX, DEFINE, DFL, DH, DI, DL, DOMAIN, DS, DX, ELSE, END, ES, EVALUATE, EXIT, F, FLAG, FOREVER, FROM, G, GO, GR, IF, IFL, INTEGER, IP, LENGTH, LINE, LIST, LOAD, MOD, MODULE, NOSYMBOL, NOT, OFL, OR, ORIF, PFL, POINTER, PORT, PS, PSTEP, R, RADIX, RAH, RAL, RAX, RBH, RBL, RBX, RCH, RCL, RCX, RDH, RDL, RDX, REAL, REGISTER, REMOVE, REPEAT, RF, S, SFL, SI, SINTEGER, SP, SREAL, SS, STACK, STEP, SYMBOL, T, TFL, THEN, TILL, TO, TREAL, UNTIL, WHILE, WORD, WPORT, WS, XOR, ZFL.

Abbreviation Rule

Reserved Words can be abbreviated to, at least, their first three characters. Thus, POI, POIN, POINT, POINTE all have the same meaning as the keyword POINTER.

Memory References

Memory locations are referenced according to their type. The format used to reference a memory location is—

type address

where

type is one of the following:

BYTE	a one byte unsigned value located at memory location *address*.

SINTEGER	a one byte signed integer.
INTEGER	a two byte integer value located at memory locations *address* and *address + 1*. An integer is displayed in two's complement form.
WORD	a two byte integer value located at memory locations *address* and *address + 1*. An integer is displayed as a 16-bit unsigned number.
POINTER	a four byte pointer value (base and displacement) located at *address*, *address + 1*, *address + 2*, and *address + 3*.
PORT	an 8-bit port value located at *address*.
WPORT	a 16-bit port value located at *address* and *address + 1*.
BOOLEAN	same as BYTE except it displays as either TRUE (any odd number, i.e., LSB=1) or FALSE (any even number, i.e., LSB=0).

Symbol Names, Symbolic References, and Symbol Table Organization

Symbol Names

A name that is not a reserved word, or a valid abbreviation of a reserved word, is considered by DEBUG-88 to be a user defined symbol name. Symbol names are composed of the valid DEBUG-88 character set with the following restrictions and extensions:

- Only the upper case and lower case alphabetic letters and the characters _, $, ?, @ can be used as the initial character of a symbol name. Numbers cannot be used as the initial character of a symbol name, although they may appear anywhere else in a symbol name. Any other characters cannot be used in symbol names.
- Case is not considered in defining a symbol name. Thus, DEBUG-88 considers Vertex to be the same symbol as VERTEX or vertex.
- The dollar sign character ($) is disregarded by DEBUG-88. Thus, $VERTEX is considered to be the same character as Vertex and as Ver$tex. The $ is therefore useful as a separator for symbol names such as Main$Data, Duplicate$Data, SumofSquares. Note that entering a space would result in DEBUG-88 defining more than one symbol. Thus, Main Data would be taken as two symbols, Main and Data.
- Symbol names must have at least one significant character. $A, A$$, and $$A are valid symbol names, while $$$ is not.
- The escape specifiers ., .., !, !! can be used to create symbol names that might otherwise conflict with reserved words. Thus, REGISTER and REG are reserved words, while .REGISTER, ..REGISTER, !REGISTER and !!REGISTER are all user defined symbol names.
- Symbol names should not exceed 40 characters in length, not including the $.

DEBUG-88 maintains a symbol table that contains all user defined symbol names and a statement number table that contains all user defined line numbers. The symbols and line numbers are loaded by the Load command. Additions to and deletions from

the unnamed module of the symbol table may be made via the Define and the Remove commands, respectively. A symbol table value that represents either the address or the numerical value is assigned to each symbol. The Symbol Type and the Value assigned are:

Symbol Type	Value
Instruction and Statement Label	Address of the Instruction
Program Variables	Address of Variable
Module Names	No Value Assigned

Modular Organization

The Symbol Table is organized on a modular basis. Symbol names are stored under the module in which they occur. The first entry in the Symbol Table is always the Unnamed Module. The Unnamed Module does not correspond to a module of the program being debugged. Rather, it contains any user defined symbol created via the Define command. The Publics Module follows the Unnamed Module. The Publics Module contains all public symbols in the system currently being debugged. Following the Publics module, the user modules are loaded in the order in which they are encountered. The symbols within a given module are likewise stored in the order in which they occur. Refer to figure 6-1 for a graphic representation of a typical Symbol table.

Programming Languages and Modular Organization

In PL/M-86 and PASCAL-86 programs, the module name is the label of a simple DO block that is not nested in any other block. In Assembly language programs, a module name is a label that is the object of a NAME directive. In FORTRAN-86, module names are program names and subprogram names.

Accessing Symbols

Symbols are accessed via the module name (if used) and the symbol name. The format is—

[*..module*] *.symbol...*

The module name is placed in brackets to indicate that it is optional. Note that module names are designated as such by the two consecutive period prefix (..). Symbol names are designated by a single period prefix (.). In searching for a symbol, the first occurrence of the symbol is always taken. The Unnamed Module is searched first. Then, the User Modules are searched in the order in which they occur. Since symbol names can occur more than once, you may need to use both a module name and a number of symbol names to access a particular symbol. You can also use module names and symbol names to speed up the accessing process. A typical example of the modules and the symbols they contain is shown below.

Module	Symbol
Unnamed Module	
Public Module	
..Blue	.X .Y .Z .Z1 .Y .R2 .Y
..Green	.X .Y .Z .Z .Y
..Three	.X .Z .X .R23 .XX .X .T2 .X
..Bigmod	.X .Z .Y .Z2

Assume you want to retrieve symbol X, which occurs in the module named Bigmod. The symbol X occurs in three other modules. You can quickly and precisely reference it as ..Bigmod .X. This designates the first occurrence of the symbol named X in module Bigmod. If you failed to designate a module name, the unnamed module would be searched for symbol X.

Next, assume you wish to retrieve the fourth occurrence of symbol X in module Three. You could designate the entire reference as—

..Three .X .Z .X .R23 .XX .X .T2 .X

A shortcut method that would still designate the fourth occurrence of .X in module Three would be—

..Three .T2 .X

which requests the first occurrence of the symbol X occurring after symbol T2 in module Three. Once you become familiar with this indexing technique, you will be able to access symbol names quickly and succinctly. The symbols in the symbol table invariably appear in the order in which they were declared in the source program. Module names can be omitted if so desired.

Statement Number References

User program locations can also be referenced by their user line number. The statement number table is organized in the same modular fashion as the symbol table. The format for accessing the line number is:

[*..module*] *# statement number*

where

statement number is an integer or an expression evaluating to an integer. A default decimal radix is assumed if no explicit radix is specified.

DEBUG-88 responds by returning a pointer value that is the absolute address of the first instruction generated for the source statement.

Examples are:

```
#45          (line number 45)
..TEST1 #12H (line number 12 hexadecimal in module TEST1)
```

Line numbers are interpreted as decimal unless explicitly stated to be otherwise.

String Constants

Single quotes are used to designate a character string. Any ASCII character can be placed within a string. Each character is assigned an 8-bit value, with the 7 low order bits equal to the ASCII code for the character, and the highest order bit set to 0.

Table 3-1 lists all printable ASCII characters and their corresponding hexadecimal codes.

CHAPTER 4
UTILITY COMMANDS

DEBUG-88 Utility Commands provide file management capabilities. The Utility Commands are D88, LOAD, LIST, and EXIT.

D88 — Invoke DEBUG-88

Entering D88 at the console invokes DEBUG-88.

Syntax

```
        WORKSPACE (ws-size)...
D88     NOSYMBOL                ... <cr> [filename [arguments...]] <cr>
        NOLINE
        NOPUBLIC
```

where

WORKSPACE	causes *ws-size* bytes of storage to be allocated by DEBUG-88 for symbol table space.
ws-size	is a decimal integer that represents the workspace size. If not specified, a default size of 10240 is assumed. If *ws-size* exceeds 65535, it is truncated to the 16 least significant bits.
NOLINE, NOSYMBOL and NOPUBLIC	are modifiers that affect the type of symbolic information available to DEBUG-88.
filename	is the user defined filename, with any *arguments* required to invoke a given program under DEBUG-88.

If more than one option is selected (e.g., WORKSPACE and NOSYMBOL), the entries are separated by a space.

Abbreviations

WORKSPACE can be abbreviated to WS, NOSYMBOL to NOS, NOLINE to NOL, NOPUBLIC to NOP. Note that in the case of the invocation, only the abbreviations shown can be used.

Description

DEBUG-88 is invoked by entering D88 at the console. The optional WORKSPACE parameter can be used to specify the number of bytes of storage to be reserved for the DEBUG-88 symbol table. When no WORKSPACE is explicitly specified, the default size of 10240 is assumed. The modifier NOLINE suppresses the loading of line numbers. NOSYMBOL suppresses the loading of translator/assembler defined symbols. Listing of any public symbols, however, is not suppressed by this modifier. To suppress the listing of public symbols, you must use the NOPUBLIC modifier. Suppressing the loading of any unneeded information conserves memory.

Examples

1. D88 ‹cr›
2. D88 WS (5620) NOS NOL NOP ‹cr›
3. D88 NOS/WORKDISK/HANDRV.86 ‹cr›

LOAD — Load 8086/8088 Object Code

The Load command allows you to load 8086 object code.

Syntax

```
                  NOLINE
LOAD "filename"   NOSYMBOL  . . .   <cr>
                  NOPUBLIC
```

where

filename	is the complete name of the file that contains your object code. Extensions to file names are not assumed. Any extension must be explicitly entered.
NOLINE, NOSYMBOL and NOPUBLIC	are modifiers that affect the type of symbolic information available to DEBUG-88.

If more than one option is selected, the entries are separated by a space.

Abbreviations

LOAD can be abbreviated to LOA, NOSYMBOL to NOS, NOSY, or NOSYM, NOLINE to NOL, NOPUBLIC to NOP. Note that, unlike the invocation, the general abbreviation rule pertains to Load and to all other DEBUG-88 commands. In subsequent examples, only a few of the possible abbreviations will be shown.

Description

The Load command allows DEBUG-88 to access the symbolic names and the line numbers of user generated programs. Execution of the Load command loads the user program code and places all user generated symbolic names and line numbers in the DEBUG-88 symbol table (unless they are suppressed by the NOS, NOL or NOP modifiers). All break registers are cleared of any previous values.

The modifier NOLINE suppresses the loading of line numbers. NOSYMBOL suppresses the loading of translator/assembler defined symbols. Listing of public symbols, however, is not suppressed. To suppress the listing of public symbols, you must use the NOPUBLIC modifier. Suppressing of unneeded information conserves memory.

Examples

```
1. * LOAD "/WORKDISK/HANDRV.86" <cr>
2. * LOAD "BIGFILE.86" NOS <cr>
3. * LOAD "/WD3/HH" NOS NOP <cr>
```

LIST — Create a Listing

The List command produces a listing of the DEBUG-88 console interaction on some specified external device.

Syntax

```
LIST ["filename"] <cr>
```

where

filename is the name you give to your output log file.

Abbreviations

LIST can be abbreviated to LIS.

Description

The List command produces a log of the DEBUG-88–console interaction. The log file is given the name specified in the command. If you do not specify a filename, the current log filename is displayed. If a log file currently exists and none is desired, the null filename (" ") can be specified.

Examples

1. ```
   * LIST "/LOGDISK/LIST.LOG" <cr>
   ```

   This creates a log under the filename LOGDISK/LIST.LOG.

2. ```
   * LIS <cr>
   ```
3. ```
   * LIST "" <cr>
   ```
   ```

EXIT — Exit DEBUG-88

The Exit command causes you to exit from DEBUG-88.

Syntax

```
EXIT
```

Abbreviations

EXIT can be abbreviated to EXI.

Description

The Exit command causes an unconditional exit from DEBUG-88. Control is returned to the operating system.

Examples

1. * EXIT <cr>
2. * EXI <cr>

CHAPTER 5
EXECUTION COMMANDS

This chapter describes the commands used to: transfer control between DEBUG-88 and the user program; specify the operational mode (single step or continuous); and, set starting and stopping points for user program execution. Generally, when using DEBUG-88 to check your program, you will want to execute particular program segments, and then halt and return control to DEBUG-88 so you can check register, memory or flag contents. The two break registers allow you to specify break points for execution of your program.

DEBUG-88 has two break registers, BR0 and BR1. BR is used to represent both break registers. Each break register can be set to an address in your program. The function of the break register is to halt execution of the user program as the address in the break register is about to be executed.

After the break registers have been set to the desired break address, they must be explicitly activated. This is done by activating the GO register (GR). The GO command is used to transfer control from DEBUG-88 to your program.

BR0, BR1 and BR — Display/Set Break Register

BR0, BR1 and BR are used to display the contents of the break registers or to set either or both break registers to a new address.

Syntax

```
BR0 ‹cr›
BR1 ‹cr›
BR  ‹cr›
```

```
                          expression
BR0 or BR1 or BR = base and displacement  ‹cr›
                   address
```

where

expression	is an arithmetic expression that evaluates to an integer.
base and displacement	is an address represented in base and displacement form.
address	is an address.

The first three commands shown display the current contents of Break Register 0, Break Register 1, or both Break Registers, respectively. The examples that follow set the Break Registers to new values.

Examples

1. * BR ‹cr›

   This command displays the contents of both Break Registers.

2. * BR1 = CS:IP + 5 ‹cr›

   When the break register is followed by an equal sign and an expression, the break address is set equal to the address of the expression or equal to the register base and displacement to the right of the equal sign. Thus, in example 2, Break Register 1 is set to the new value equal to the instruction pointer plus five. The instruction pointer is the Code Segment (CS) displaced by the Instruction Pointer (IP).

3. * BR0 = .ADDR1 ‹cr›

   This command sets Break Register 0 equal to address .ADDR1.

GR — Display/Set the Go Register

Setting the GO register activates the Break Registers.

Syntax

Display the Go Register.

```
GR ‹cr›
```

Set the Go Register.

```
        FOREVER
GR  =   TILL breakaddress [OR breakaddress]    ‹cr›
        TILL breakregister [OR breakregister]
```

where

FOREVER	disables all break points. Execution of the user program will cease only if a halt instruction is encountered or if one of the two escapes is used.
TILL	causes user program execution to continue until the break address is reached.
breakaddress	is an integer or a pointer that represents the break point.
breakregister	is one of the Break Registers (BR0 or BR1) or both Break Registers (BR).

Abbreviations

TILL can be abbreviated to TIL or T, FOREVER to FOR.

Examples

1.
```
* GR ‹cr›
```

This displays the contents of the GO register.

2.
```
* GR = TILL BR1 ‹cr›
```

This activates BR1 and deactivates BR0.

3.
```
* GR = FOREVER ‹cr›
```

This deactivates both break registers. Execution of the user program continues unless an escape code is used.

4.
```
* GR = TIL CS:IP OR ..MOD1.SCANNER ‹cr›
```

This activates BR0 (set to address pointed at by CS:IP) and BR1 (set to the address pointed at by symbol .SCANNER) in module ..MOD1. Execution of the user program continues until the first of the two break points is encountered.

GO — Transfer Control and Execute User Program

The GO command transfers control from DEBUG-88 to the user program.

Syntax

```
                    FOREVER
GO FROM address     TILL breakaddress [OR breakaddress]
                    TILL breakregister [OR breakregister]
```

where

FROM	specifies a starting address for user program execution.
FOREVER	disables all break points.
breakaddress and *breakregister*	specify the desired break points.

Abbreviations

GO can be abbreviated to G, FROM to FRO or F, TILL to TIL or T, FOREVER to FOR.

Description

The GO command transfers control from DEBUG-88 to the user program. If no starting point is explicitly specified, the value of CS:IP is taken as the starting address. Otherwise, FROM is used to specify either a single breakpoint or two breakpoints, of which the first encountered is used. TILL is used to specify a single breakpoint, TILL ... OR to specify two breakpoints. Specifying FOREVER disables all breakpoints (i.e., user program execution goes on without interruption unless an escape sequence is used).

Examples

1. `* GO TILL .LAB4 <cr>`

   Control is transferred from DEBUG-88 to the user program. Execution of the user program begins at the address pointed at by CS:IP. The user program is executed until location .LAB4 is reached, at which point control is returned to DEBUG-88.

2. `* G FROM .LABEL1 TIL BR0 OR BR1 <cr>`

   Control is transferred from DEBUG-88 to the user program. Execution begins at address .LABEL1. Execution continues until the address of Break Point 0 or the address of Break Point 1 is reached.

Specifying Operational Mode

Unless you explicitly designate otherwise, when control is transferred from DEBUG-88 to the user program, that program executes in continuous mode. It is, however, often useful to execute the user program instructions one at a time, i.e., after each instruction a break immediately occurs. Two DEBUG-88 commands are available for this purpose—STEP and PSTEP. The difference between the two lies in the manner in which they execute CALLs. STEP executes each instruction of a CALL just as it does any other instruction. PSTEP executes CALLs in regular mode, then returns to single step through the main line code.

CONTROL-D—Asynchronous Breakpoint Escape Code

The Escape Code, CONTROL-D, is used to terminate execution of the user program unconditionally and then return control to DEBUG-88 command mode from the STEP and PSTEP commands. At times, you may want to run your program without knowing in advance where to set a breakpoint that will return control to DEBUG-88. In such a case, or after using the FOREVER command, you must use the escape code to return control to DEBUG-88. To escape, depress the Control key and the D key simultaneously. The escape will not function if you have disabled keyboard interrupts.

STEP — Single Step Through All Instructions

Syntax

```
STEP [FROM address] ‹cr›
```

where

address is the starting address.

Abbreviations

STEP can be abbreviated to STE, FROM to FRO or F.

Description

The Step command initiates sequential single step execution of the user program. Each instruction is displayed in disassembled form and the user is queried by a question mark (?) that appears on the screen. If the user responds by entering a carriage return, ‹cr›, the instruction is executed and the process is repeated for the next instruction. If the user responds by entering the escape (CONTROL-D), the user is returned to DEBUG-88 command mode. If the Step command is nested in a compound command, interactive querying does not take place. Instead, the current instruction is performed in single step mode and the next command in the compound command is executed. You may explicitly designate a starting address by using FROM or you may use the default starting address of CS:IP. In the case of CALLs, each instruction of the CALL is executed. CALLs are treated no differently than other segments of code.

Examples

1. 

   Start single step execution with the address pointed at by .RCV in module ..Main.

PSTEP — Single Step Through Main Line Code, Bypassing Calls

Syntax

```
PSTEP [FROM address] ‹cr›
```

where

address is a designated address.

Abbreviations

PSTEP can be abbreviated to PS, FROM to FRO or F.

Description

The Pstep command initiates single step execution of the user program from a specified starting address or from the default starting address of CS:IP. Each instruction is displayed in disassembled form and the user is queried by a question mark (?) that appears on the screen. If the user responds by entering a carriage return, ‹ cr ›, the command is executed and the process is repeated for the next instruction. If the user responds by entering the escape (CONTROL-D), the user is returned to DEBUG-88 command mode. If the Step command is nested in a compound command, interactive querying does not take place. Instead, the current instruction is performed in single step mode and the next command in the compound command is then executed. PSTEP checks whether the instruction about to be executed is a CALL instruction. For non-CALL instructions, normal single step execution occurs. For a CALL, a breakpoint is set at the instruction immediately after the CALL and control is returned to the main line user code. Thus, CALLs are "side stepped." This is quite useful when a code segment to be debugged contains numerous and/or lengthy CALLs known to be error free. Using PSTEP rather than STEP significantly reduces debugging time.

Examples

1.
```
* PSTEP ‹cr›
```

Execute in single step mode, sidestepping CALLs starting at address CS:IP.

2.
```
* PSTEP FROM ..MAIN.RCV ‹cr›
```

Execute in single step mode, sidestepping CALLs starting at location .RCV in module ..MAIN.

CHAPTER 6
SIMPLE COMMANDS

The Simple Commands are those (other than the Utility Commands and the Execution Commands previously described) that do not allow nesting of other commands. The majority of Simple Commands display or set registers, flags, memory locations and the stack. Also included among the Simple Commands are the symbol manipulation commands, the Radix, the Instruction Disassembly and the Domain command.

DOMAIN — Symbol Table Look Up

The Domain command sets the module in which to begin a search for user symbol names.

Syntax

```
DOMAIN  ..module name <cr>
```

where

module name — is the user defined module name. Note that the double period prefix (..) must be used to identify the symbol as a module name.

Abbreviations

DOMAIN can be abbreviated to DOM.

Description

The Domain command is introduced here because its use can greatly facilitate the use of the other Simple Commands. DEBUG-88 assumes that all symbols that are not DEBUG-88 reserved words are user defined objects. The Symbol Table lists all the symbol entries in the order in which they are encountered in the user program. Figure 6-1 graphically explains the ordering of the Symbol Table. The table is broken down into modules. The first module is always the unnamed module. Any symbol table entries created using the Define command are placed in the unnamed module. All public symbols are placed in the public module. If you specified the NOPUBLIC parameter at invocation, the public module will contain no entries. The subsequent modules are named for and correspond on a one-to-one basis with each module of your program. To avoid having to search the entire symbol table from the beginning when searching for a given symbol, use the Domain command. The Domain command allows you to specify the module in which to begin the search. Thus, in example one, entering the command DOMAIN ..MODK allows you to begin a search in Module K, bypassing modules 1, 2, etc. Without the Domain command, all modules starting at module 1 would be searched in sequence.

Examples

1.
```
* DOMAIN ..MODK <cr>
```

Set the start of the symbol table search to module ..MODK.

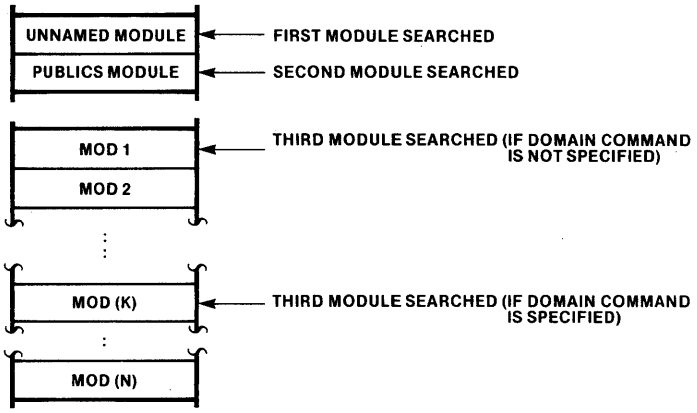

Figure 6-1. DEBUG-88 Symbol Table Structure 121758-1

Display/Set Commands

The Display Commands allow you to display the contents of the registers, the flags, memory locations and the stacks. Several entries can be placed on a single command line by separating them with commas. Memory can be displayed in machine code and in disassembled form. Set Commands are produced by placing an equal sign and the value in the command line to the right of the keyword being set.

The Display Commands are:

Display Registers
Display Flags
Display Stack
Display Symbols
Display Lines
Display Modules
Display Memory
Display Disassembled Memory

The Set Commands are:

Set Register
Set Stack
Set Memory
Set Flag

REGISTER — Display Contents of 8086/8088 Register

The Register command causes DEBUG-88 to display the contents of all 8086/8088 registers, or of a specified register or set of registers.

Syntax

```
REGISTER ‹cr›
register ‹cr›
```

where

register	is any of the following entries: 8-bit registers—RAL, RAH, RBL, RBH, RCL, RCH, RDL, RDH or 16-bit registers—RAX, RBX, RCX, RDX, SP, BP, SI, DI, SS, CS, DS, ES, IP, RF.

Abbreviations

REGISTER can be abbreviated to REG.

The initial R of any register name may be omitted.

REGISTER — Set Contents of 8086/8088 Register to a New Value

The Register command causes DEBUG-88 to set the contents of all 8086/8088 registers, or of a specified register or set of registers.

Syntax

register = *variable* ‹ c r ›

where

register	is any of the following entries: 8-bit register—RAL, RAH, RBL, RBH, RCL, RCH, RDL, RDH; or 16-bit registers—RAX, RBX, RCX, RDX, SP, BP, SI, DI, SS, CS, DS, ES, IP, RF.
variable	is a pointer, an integer or a label.

Abbreviations

Register can be abbreviated to REG. The initial R of any register name may be omitted.

Description

The register is assigned the value of the variable on the right side of the equal sign.

Examples

1. * REG ‹cr›

   The contents of all registers are displayed.

```
AX = nnnnh BX = nnnnh CX = nnnnh DX = nnnnh
CS = nnnnh SS = nnnnh DS = nnnnh ES = nnnnh
IP = nnnnh SP = nnnnh SI = nnnnh DI = nnnnh
RF = nnnnh BP = nnnnh
```

   where

nnnn	is replaced with the current hex value of the register in question.

2. * RAX ‹cr›

   Contents of the A register are displayed.

3. * AX ‹cr›

   Contents of the A register are displayed.

4. * AX = 415h ‹cr›

   The A register is assigned the value 415 hexadecimal.

FLAG — Display Contents of 8086/8088 Flags

The Display Flags command displays the contents of all 8086/8088 flags or a set of flags.

Syntax

```
FLAG
flag
```

where

flag	is any of the one-bit status flags: CFL, PFL, AFL, ZFL, SFL, TFL, IFL, DFL, OFL.

Abbreviations

FLAG can be abbreviated to FLA.

Description

The value of one or more of the nine one-bit flags is displayed. Each flag can have a value of either 0 or 1.

FLAG — Set Contents of 8086/8088 Flags to a New Value

The Display Flags command sets the contents of all 8086/8088 flags or a set of flags.

Syntax

flag = variable

where

flag	is any of the one-bit status flags: CFL, PFL, AFL, ZFL, SFL, TFL, IDL, DFL, OFL.

Description

The flag is assigned the one-bit value to the left of the equal sign.

Examples

1. * FLAG <cr>

   The contents of all 9 flags are displayed.

   ```
   CFL=b, PFL=b, AFL=b, ZFL=b, SFL=b, TFL=b, IFL=b, DFL=b, OFL=b
   ```

2. * CFL = 0

   The carry flag is assigned a value of 0.

   Note that flags and registers can be used to function as numbers in expressions.

STACK — Display User Stack Contents

The Display Stack command displays the contents of the user's stack.

Syntax

```
STACK nn <cr>
```

where

nn is an integer that defines the number of words to be displayed.

Abbreviations

STACK can be abbreviated to STA.

Description

The stack is pointed at by SS:SP. The Display Stack command displays *nn* words, starting from the top of the stack. If *nn* is not explicitly stated, a default value of 1 is assumed.

Examples

1. * STACK 10 <cr>

Ten decimal words, starting from the top of the stack, are displayed.

DISPLAY MEMORY

The Display Memory command displays the contents of one or more 8086/8088 memory locations in machine code form.

Syntax

memory-type address `TO` *end-address* `<cr>`
`LENGTH` *n*

where

memory-type	is one of the following: BYTE, WORD, SINTEGER, INTEGER, POINTER, BOOLEAN.
address	is the address in memory to be displayed, or the starting address of the segment to be displayed.
end address	is the ending address of the display.
n	is an integer that represents the number of memory locations to be displayed. A decimal default radix is assumed.

Abbreviations

BYTE can be abbreviated to BYT, WORD to WOR, SINTEGER to SIN, INTEGER to INT, POINTER to POI, LENGTH to LEN.

Description

This command is used to display one or more memory locations or to set a memory location to a new value. The memory types that can be displayed are BYTE, SINTEGER, WORD, INTEGER, and POINTER.

Examples

1. * `BYTE CS:IP LENGTH 18 <cr>`

   18 bytes of memory starting at the instruction pointer are displayed.

2. * `BYTE CS:IP=0A3 <cr>`

   The memory location specified by the instruction pointer is set equal to the value A3 hex.

3. * `WORD CS:IP <cr>`

   Displays one word of memory located at the instruction pointer.

   It is also possible to display a series of memory locations.

4. * `WORD .VecStart TO .VecEnd <cr>`

   Displays the successive WORDs of memory starting at location .VecStart and ending at .VecEnd.

NOTE

Fractional parts of a memory location are not displayed. Thus, in example 4 above, if .VecStart to .VecEnd occupied an odd number of bytes, the byte at .VecEnd would not be displayed.

5. * WORD .Vec1 LEN 4<cr>

   Displays four successive WORDs of memory starting at location .Vec1.

6. Likewise, it is possible to assign new values to successive memory locations.

   * BYTE .A=5 <cr>

   Assigns the byte of memory at location .A the value 5.

7. * BYTE .A=BYTE .B <cr>

   Assigns the byte of memory at location .A the value of the byte of memory at location .B.

8. * WORD .A LEN 2=5,6 <cr>

   Assigns the two successive WORDs starting at memory location .A the values 5 and 6.

9. * WORD .A LEN 10=5,6,7 <cr>

   Implicitly recognizes from the right side of the equation that the three constants 5,6,7 are to be used in cyclical fashion to assign new values to the 10 successive words starting at location .A. Thus, WORD .A is set to 5 and the successive 9 words of memory are assigned the values 6,7,5,6,7,5,6,7,5.

10. * WORD .A= Word .B to Word .C <cr>

    Assigns each WORD of memory starting at location .A the value of each successive WORD of memory starting at location WORD .B until WORD .C is reached.

11. * BYTE .A= Word .B to .C <cr>

    Assigns each byte of memory starting at location .A the value of each successive word starting at word .B until the WORD at .C is reached. Each successive word, however, is truncated to a byte value during the assignment.

12. You may also intermix the use of expressions with LENGTH and TO to form complicated assignments such as

    * WORD .A=3,BYTE .B LEN 10,8, WORD .X TO .Y <cr>

    In this example, successive bytes of memory starting at location .A are assigned the value 3, the value of the 10 successive bytes starting at location .B the value 8, and truncated values of each WORD of memory from location .X to location .Y. The general rule governing assignments is that the left side of the equation defines the type of memory location (e.g., WORD or BYTE) to be assigned a value, and the right side of the equation determines the number of successive memory locations to be assigned new values. One assignment is made for each element on the right side of the equation. If the number of elements on the left side of the assignment exceeds the number of elements explicitly designated on the right side, the elements on the right side are assigned in cyclical fashion.

PORT/WPORT — Display/Set Contents of One or More I/O Ports

PORT and WPORT are used to display the contents of one or more I/O ports or to set the ports to new values.

Syntax

port-type address `TO` *end-address* `=` *expression...* ⟨cr⟩
`LENGTH` *n* `='`

where

port-type	is one of the following: PORT-an 8-bit port value located at *address*. WPORT-a 16-bit port value located at *address* and *address +1*.
address	is the port address or the starting port address. Its value must be 0 to 65535.
end address	is the address of the last port in the list.
n	is an integer that defines the number of ports.
expression	is the value(s) that replaces the port(s).

Abbreviations

PORT can be abbreviated to POR, WPORT to WPO, LENGTH to LEN.

Description

Entering PORT or WPORT and the address displays the port contents. To change the contents to a new value, enter that value to the right of an equal sign. If more than one port is to be displayed or changed, the TO or the LENGTH options should be used.

Examples

1. `* PORT D8 = 41 ⟨cr⟩`

   The contents of the one byte PORT are set to 41.

2. `* WPORT 1000 ⟨cr⟩`

   The value of the Word Port is displayed.

ASM — Display Contents of Memory as Disassembled Instructions

The ASM command displays the contents of one or more memory locations as disassembled instructions.

Syntax

```
ASM address    TO end-address    ‹cr›
               LENGTH nn
```

where

address	is the address in memory to be displayed, or the starting address of a segment of memory to be displayed.
end-address	is the last address of memory to be displayed.
nn	is an integer that represents the number of instructions to be displayed. A default decimal radix is assumed.

Abbreviations

ASM can be abbreviated to A, LENGTH to LEN.

Description

The ASM instruction allows you to display the contents of one or more user program locations as disassembled instructions, i.e., they are displayed as mnemonic code. If no starting address is specified, CS:IP is taken as a default address. If the LENGTH parameter is used, a default decimal radix is assumed. If neither the TO or the LENGTH options are specified (i.e., you enter ASM ‹cr›), ten (decimal) instructions starting at CS:IP are displayed.

Examples

1.
```
* ASM .LAB1 TO .LAB2 ‹cr›
```

   Instructions from .LAB1 to location .LAB2 are displayed as diassembled instructions.

2.
```
* ASM .LABL LENGTH 10 ‹cr›
```

   Ten instructions, starting at location .LABL, are displayed as disassembled instructions.

EVALUATE — Evaluate Expressions

The Evaluate command allows you to evaluate an integer in five bases or to evaluate a pointer as a symbol or as a line number.

Syntax

```
EVALUATE expression   {SYMBOL}
                      {LINE}
```

where

expression	is an integer or label treated as an integer.
SYMBOL	is a modifier that causes DEBUG-88 to treat the expression as a pointer to be compared against the addresses of symbol names in the symbol table.
LINE	is a modifier that causes DEBUG-88 to treat the expression as a pointer to be compared against the addresses of line numbers in the symbol table.

Abbreviations

EVALUATE can be abbreviated to EVA, SYMBOL to SYM and LINE to LIN. Y, Q, T and H represent the binary, octal, decimal and hexadecimal bases, respectively.

Description

The Evaluate command is to be used in three ways.

First (when no modifier is appended), it treats the variable expressions as an integer and displays its value in five different bases: binary, octal, decimal, hexadecimal and ASCII.

Second (when the modifier SYMBOL is appended), DEBUG-88 treats the variable expression as a pointer. The symbol table is searched for the address of a symbol that is equal to the pointer. If no equivalent is found, the closest symbol whose address is less than the variable expression is taken. The address of the pointer, or the closest address and its displacement from the pointer, is returned.

Third (when the modifier LINE is appended), DEBUG-88 treats the variable expression as a pointer. The symbol table is searched for the address of a line number that is equal to the pointer. If no equivalent is found, the closest line number whose address is less than the variable expression, is taken. The address of the pointer, or the closest address and its displacement from the pointer, is returned.

Examples

1.
```
* EVALUATE 22 <cr>
```

A default hexadecimal radix is assumed unless another radix is explicitly stated.

DEBUG-88 returns

```
>0000000000100010y = 00042q = 00034t = 0022H = "
```

This gives 22 in the binary, octal, decimal and hexadecimal radixes, and in the equivalent ASCII character.

2. * RADIX 10T <cr>

This sets the radix to decimal.

3. * EVALUATE 10 <cr>

DEBUG-88 returns

```
*00000000000001010y = 000012q = 00010t = 000Ah = .
```

ASCII characters not printable within a single space are shown as a period (.).

For examples 4 and 5 of EVALUATE SYMBOL and EVALUATE LINE, refer to the sample symbol table shown below.

Sample Symbol/Line Number Table

Symbol/Line Number	Address
..X	100
.SYM	672
.VAR	211
#10	110
..Y	200
#10	210
#20	220

4. * EVA 212 + 3 SYM <cr>

This command searches the symbol table for a symbol whose address = 212 + 3 =215 (hexadecimal default radix is assumed). Since no symbol has an address of 215, the next lowest address of a symbol is taken. In the sample symbol table above, the next lowest address of a symbol is 211.

DEBUG-88 returns

```
*0000:0215 = ..X.VAR + 4.
```

This indicates that address 215 is 4 locations beyond the symbol VAR in module X.

5. * EVA 212 + 3 LINE <cr>

This command searches the symbol table for a line number whose address = 212 + 3 = 215 (current radix is assumed). Since no line number has an address of 215, the next lowest address of a line number is taken. In the sample line number table above, the next lowest address of a line number is 210.

DEBUG-88 returns

```
*0000:0215 = ..Y#10 + 5.
```

This indicates that address 215 is 5 locations beyond the line number 10 in module Y.

RADIX — Establish a New Radix

The Radix command is used to establish a new radix.

Syntax

```
RADIX [rb] <cr>
```

where

r	is an integer used to designate the new radix. *r* must designate binary, octal, decimal or hexadecimal.
b	is the radix of *r*. *b* must equal y, q, t or h (or Y, Q, T, or H). If no radix is given, the existing radix is used.

Abbreviations

RADIX can be abbreviated to RAD.

Description

The Radix command is used to establish a new radix. A decimal default radix is assumed. After a new radix is set, it remains in effect until the radix command is used again to establish another radix. Be sure to keep in mind what radix you have established.

Examples

1.
```
* RADIX 10t <cr>
```

   The radix is set to decimal.

2.
```
* RAD 1010y <cr>
```

   This also sets the radix to decimal as 1010 binary = 10 decimal.

3.
```
* RAD 16T <cr>
```

   This re-establishes the radix to hexadecimal.

4.
```
* RAD 0A <cr>
```

   Since no radix is stated in the command, the existing radix (which could be the default radix of 16) is used. Assuming for the example that the default radix is still in effect, the radix is set by this command to decimal.

   Entering simply RADIX or RAD will display the current radix. The radix always is displayed as a decimal number.

5.
```
* RADIX <cr>
*10
```

   This indicates the current radix is decimal.

Symbol Manipulation Commands

SYMBOL — Display All Symbols

The Symbol command produces a listing of all module names and the names of all symbols that are not line numbers.

Syntax

```
SYMBOL ‹cr›
```

Abbreviations

SYMBOL can be abbreviated to SYM.

Description

The Symbol command produces a listing of all module names and all symbols that are not line numbers.If NOSYMBOL (NOS) was specified at invocation or in the Load command, only symbols in the unnamed module will be displayed.

Examples

1. * SYMBOL ‹cr›

2. * SYM ‹cr›

LINE — Display All Line Numbers

The Line command produces a listing of all module names and all line numbers.

Syntax

```
LINE <cr>
```

Abbreviations

Line can be abbreviated to LIN.

Description

The Line command produces a listing of all module names and all line numbers. The Line command will not work if NOLINE (NOL) was specified in the invocation or in the Load command.

Examples

1. * LINE <cr>

2. * LIN <cr>

MODULE — Display All Module Names

The Module command produces a listing of the names of all loaded modules.

Syntax

```
MODULE <cr>
```

Abbreviations

MODULE can be abbreviated to MOD.

Description

The Module command produces a listing of the names of all loaded modules. If the NOPUBLIC (NOP) option was specified at invocation, the names of public modules will not be listed by the Module command.

Examples

1.
```
* MODULE <cr>
```

2.
```
* MOD <cr>
```

DEFINE — Create a Symbol Table Entry

The Define command creates a new symbol table entry and places it in the unnamed module.

Syntax

```
DEFINE .symbol = expression ‹cr›
```

where

.symbol	is the name being given to the new symbol.
expression	is the value to which the symbol is being set equal.

Abbreviations

DEFINE can be abbreviated to DEF.

Description

The Define command creates a new user symbol and places it in the unnamed module. DEFINEd symbols always point to a WORD value. The pointer or symbol itself can be used as a WORD, a BYTE, or a POINTER. Symbols are placed in the unnamed module in the reverse order from which they were defined. This ensures that the Remove command, (see next command), which is the complement of the Define command, accesses the most recently defined values of a given symbol.

Examples

1.
```
* DEF .A = 0 ‹cr›
* DEF .VAR = 22:1 ‹cr›
* DEF .X = .A ‹cr›
* DEF .VAR = .A ‹cr›
* DEF .A = .VAR ‹cr›
```

The symbols, sequentially, take on the values

.A = 0:0
.VAR = 22:1
.X = .A = 0:0
.VAR = .A = 0:0
.A = .VAR = 0:0

REMOVE — Delete an Entry From the Symbol Table

The Remove command deletes one or more previously defined symbol names from the unnamed module.

Syntax

```
REMOVE .symbol name ...
```

where

.symbol name is a previously defined symbol in the unnamed module.

Abbreviations

REMOVE can be abbreviated to REM.

Description

The Remove command deletes one or more previously defined symbols from the unnamed module. Commas are used to separate multiple entries in the command. The entries are removed in the reverse order from which they were entered. This ensures that the most recently defined value of a symbol is accessed first.

Examples

1.
```
* REMOVE .STA <cr>
```

2.
```
* REM .STA, .VAR, .VAR3 <cr>
```

CHAPTER 7
COMPOUND COMMANDS

DEBUG-88 has three compound commands—REPEAT, COUNT, and IF. These commands allow you to set up program loops. By embedding the simple commands within the loops, you can significantly increase the power of the debugger. Program loops may be nested within other program loops. DEBUG-88 helps you to keep track of the nesting level by indenting one position for each command entered.

IF — Conditional Execution

The IF command provides for conditional execution of subsequent commands based upon the result of evaluating an expression.

Syntax

```
IF  expression evaluation THEN  <cr>
    command <cr>
[ORIF  expression evaluation THEN  <cr>]
[      command <cr>                    ]
[ELSE  <cr>]
[command]
END  <cr>
```

Abbreviations

THEN can be abbreviated to THE, ORIF to ORI, ELSE to ELS.

where

expression evaluation is an expression that can be evaluated using Boolean operations to produce a result that is either TRUE or FALSE.

Description

The IF command allows you to set up program loops in which the DEBUG-88 simple commands can be embedded. Execution of the simple commands is made conditional upon the evaluation of an expression that is contained within the compound command loop. The IF command consists of four parts: IF, ORIF, ELSE and END. The IF and the END lines are mandatory; the ORIF and ELSE lines are optional. If the IF line evaluates TRUE, the command following THEN is executed. If it evaluates FALSE, and no ORIF and ELSE lines have been entered, the command following END is executed. If you have entered an ORIF line, it is evaluated upon the IF line evaluating FALSE. If the ORIF line evaluates TRUE, the command on the line immediately below it is executed. If it evaluates FALSE, the command following ELSE is executed. All IF commands must have an END line as their final line.

Examples

1.
```
* IF .A = 0 THEN <cr>
.* GO TIL BR0 <cr>
.* ORIF .A = 1 THEN <cr>
.* GO TIL BR1 <cr>
.* ELSE <cr>
.* STEP <cr>
.* END <cr>
```

Symbol .A is evaluated to see if it equals 0. If so, the user program is executed until the instruction pointer reaches the first break register, BR0. .A is then tested to see if it is equal to 1. If so, the user program is executed until BR1, the other break register, is reached. If .A is equal to neither 0 nor 1, single step execution takes place. Additional commands must be entered to terminate single step execution.

COUNT — Specified Iteration

The Count command sets up a loop to be executed either the specified number of times, or until an expression evaluation causes the loop to be exited early.

Syntax

```
COUNT  expression <cr>
       command <cr>
  [WHILE  expression evaluation <cr>]    ...
  [UNTIL  expression evaluation <cr>]
END<cr>
```

where

expression	is an expression that can be evaluated to produce an integer.
expression evaluation	is an expression that can be evaluated using Boolean operations to produce a result that is either TRUE or FALSE.

Abbreviations

COUNT can be abbreviated to C or COU, WHILE to WHI, UNTIL to UNT.

Description

The Count command sets up a specified iteration loop. Simple commands embedded in the loop are executed the number of times specified by the expression that follows COUNT or until an expression evaluation causes the loop to be exited early. Escapes from the loop are provided by the use of optional WHILE and UNTIL lines.

Examples

1.
```
* COUNT WORD .ICount <cr>
.* UNTIL CS:IP = DELAY <cr>
.* STEP <cr>
.* ASM CS:IP <cr>
.* END <cr>
```

The loop is iterated either .ICount times (.ICount representing some positive integer value) or until the instruction pointer (CS:IP) points to location DELAY, whichever comes first.

2.
```
* COUNT WORD .ICount <cr>
.* WHILE CS:IP < NEWCODE <cr>
.* STEP <cr>
.* END <cr>
```

In this example, the loop is iterated either .ICount times or as long as the value of the instruction remains less than the value of NEWCODE. Should CS:IP reach a value not less than the value of NEWCODE (i.e., CS:IP >= NEWCODE), the loop will be exited without regard to the number of iterations completed. If

the WHILE option is used, execution of the loop continues so long as the expression on the WHILE line evaluates TRUE. Execution terminates before complete iteration of the loop if, during any iteration, the expression evaluates FALSE. The UNTIL option causes an early exit from the loop whenever the expression that follows UNTIL evaluates TRUE.

3.
```
* COUNT WORD .ICount <cr>
.* STEP <cr>
.* ASM CS:IP <cr>
.* END <cr>
```

A loop is set up to be iterated .ICount times. Within the loop, the instruction located at the current value of CS:IP is displayed as a disassembled instruction. Instructions are executed in single step mode.

REPEAT — Unspecified Iteration

The Repeat command sets up an infinite loop. Escape from the loop must be provided through the use of either the WHILE or the UNTIL operations.

Syntax

```
REPEAT <cr>
command <cr>
[{WHILE expression evaluation <cr>}]
[{UNTIL expression evaluation <cr>}]
END <cr>
```

where

expression evaluation is an expression that can be evaluated using Boolean operations to produce a result that is either TRUE or FALSE.

Abbreviations

REPEAT can be abbreviated to REP, WHILE to WHI, UNTIL to UNT.

Description

The Repeat command sets up an infinite loop. Exit from the loop is provided through evaluation of the WHILE or UNTIL lines. A FALSE result for the WHILE line or a TRUE result for the UNTIL line will cause the loop to be exited.

Examples

1.
```
* REPEAT <cr>
.* ASM CS:IP <cr>
.* UNTIL CS:IP = .LAB5 <cr>
.* STEP <cr>
.* END <cr>
```

The instruction pointed to by CS:IP is displayed in disassembled form until the instruction pointer reaches .LAB5. The loop is then exited and control is returned to DEBUG-88.

2.
```
* REP <cr>
.* ASM CS:IP <cr>
.* WHILE CS:IP<.BNDRY <cr>
.* END <cr>
```

The instruction pointed at by CS:IP is disassembled so long as CS:IP is less than .BNDRY. When the value of CS:IP equals the address of .BNDRY, the loop is exited and control is returned to DEBUG-88.

Nesting Compound Commands

The REPEAT, COUNT, and IF commands can be nested to provide a variety of control structure with increased flexibility. Each nested REPEAT or COUNT command can contain its own exit clauses (WHILE or UNTIL). Each exit clause can terminate the loop that contains it, but has no effect on any outer loops or commands.

As an example of nesting, assume you want to STEP through a program and display each instruction in disassembled form, but skip the repetitive timeout routine (.DELAY) that is called several times during program execution. One way of doing this is shown below:

Examples

1.
```
*   REPEAT <cr>
.*  IF CS:IP = .DELAY <cr>
..* IP = WORD SS:SP <cr>
..* SP = SP + 2 <cr>
..* END <cr>
.*  STEP <cr>
.*  ASM CS:IP <cr>
.*  END <cr>
```

At each call to .DELAY in the program, the displacement of the return address for the call is pushed onto the stack. The keyword SP refers to the Stack Pointer, and SS is the stack segment register. Therefore, SS:SP gives the address of the top of the stack where the return address is stored. The commands IP = WORD SS:SP and SP = SP + 2 load the return address back into IP and reset the stack pointer just as if the return instruction at the end of .DELAY had been executed.

APPENDIX A
ERROR MESSAGES

This appendix contains a list of all the error messages that can be generated by DEBUG-88. In the messages, italics indicate variables, i.e., the value is generated by each specific appearance of the error message.

#WARNING 1: Bad character; code = *n*.

A character not within DEBUG-88's character set was scanned. The character's hex value in the ASCII collating sequence is *n*. This message is a warning only. The offending character is ignored and processing resumes.

#WARNING 2: Token too long: *sss*.

The token named *sss* contains more than 128 characters. Since the longest allowable non-string token length in the Series IV DEBUG-88 system is 40 characters in length (in the case of user defined symbols), this error should occur only if a string of more than 128 characters is entered.

#WARNING 3: Invalid integer digit(s).

An integer has been entered, without explicit radix overriding, and the integer contains invalid digits for the current default input radix. For example, entering 1234 with the current default radix set to 2 would generate this error.

#ERROR 1: Invalid RADIX argument *n*.

The expression argument to the Radix command must evaluate to 2, 4, 8, or 16. This error indicates that *n* was not a valid entry.

#ERROR 2: Command is too complex.

The command entered is too complex (i.e., it is nested too deeply), thus causing an error control stack overflow. To correct this error you must simplify the command.

#ERROR 3: Command is too complex.

The command entered is too complex (i.e., it is nested too deeply), thus causing code generator stack overflow. To correct this error you must simplify the command.

#ERROR 4: Command is too complex.

The command entered is too complex (i.e., it is nested too deeply), thus causing parse stack overflow. To correct this error you must simplify the command.

#ERROR 5: DEBUG-88 heap space exhausted. *n* bytes remain.

The space used by DEBUG-88 to manage its internal data structures has been exhausted. The size of this space is controlled by the WORKSPACE parameter specified in the invocation line for DEBUG-88.

#ERROR 6: Division by zero attempted.

The evaluation of some arithmetic expression resulted in a division by zero.

#ERROR 7: Load error *n*h.

This message indicates that Series IV system error *n* hexadecimal occurred during an attempted program load. It is generated by the Load command or by the implicit load command that is issued if a filename is specified in the invocation line.

#ERROR 8: No program loaded. Press <RETURN> to execute anyway, <Ctl-D> otherwise.

This error is generated by the Step, Pstep and Go commands. It indicates that an attempt was made to transfer control to a nonexistent user program. You are prompted by the second sentence of the error message. If you wish to execute in spite of error, depress RETURN; if not, depress Control-D.

#ERROR 9: Memory access error at *ssss:nnnn*.

A memory read or write error was detected when accessing memory location *ssss:nnnn*.

#ERROR 10: Range specified with non-memory operand.

A range of "memory" was specified and one of the arguments was not a valid memory address. For example, the command AX to DX=0 is in error since AX is not the name of a memory address.

#ERROR 11: ^syntax error.

Indicates a syntax error. The circumflex (^) points to the end of the offending token.

#ERROR 12: Unaddressable object found in invalid context.

An object unaddressable by DEBUG-88 was encountered in a context that required object addressability. For example, the command 0481:1234 = 55 would generate this error since 0481:1234 is not addressable. The expression BYTE (0481:1234) is addressable however.

#ERROR 13: Unequal range limit pointer bases *aaaa:bbbb* and *cccc:dddd*.

The starting and ending limits of any range specified in DEBUG-88 have the same segment base. An example of an offending range is:

```
100:5 to 600:23
```

#ERROR 14: Attempt to copy more data into a partition than will fit.

An attempt has been made to copy more data into a partition than will fit. This error signifies that the number of elements in the right side of an assignment statement exceeds the number of elements specified on its left side. For example, Var5 length 4 = byte 0 to byte 2, 45, 33 is in error since the left side specifies four locations whereas the right side lists five values.

#ERROR 15: Symbol not found: *ssss*.

The symbol ***ssss*** was referenced by the user but could not be found in the symbol table.

```
#ERROR 16: Module not found: ssss.
```

The module *ssss* was referenced by the user but was not found in the module table for the currently loaded user program.

```
#ERROR 17: Line number not found: n.
```

Line number *n* was referenced by the user but was not found in the line number table for the currently loaded user program.

```
#ERROR 18: Symbol table overflow.
```

The symbol table has overflowed, i.e., the rule that the symbols from any given module must fit into memory has been violated. More memory can be allocated to DEBUG-88 by increasing the argument to the WORKSPACE parameter specified in the invocation file.

```
#FATAL ERROR 1: Requested WORKSPACE size n exceeds
available space.
```

The amount of memory requested during invocation by the WORKSPACE parameter exceeds the amount of free memory available to the system.

```
#FATAL ERROR 2: Error in invocation line.
```

An error was detected in the command used to invoke DEBUG-88. To prevent spurious results, this error suppresses the loading of DEBUG-88.

```
#FATAL ERROR 3: UDI error nh.
```

System error *n* hexadecimal was detected while calling some operating system service. The error was sufficiently serious to cause DEBUG-88 to exit to the operating system unconditionally.

```
#INTERNAL ERROR n[x,y]: Fatal DEBUG 88 error. Contact
Intel field service.
```

This error requires the attention of Intel field service. If you receive this error message, record the values of *n*, *x*, and *y* and report them to the field service office nearest you.

APPENDIX B
EXAMPLE OF A DEBUG SESSION

This appendix contains a log of a debug session for two user programs, the Towers of Hanoi and the Hanoi Driver. You should examine the source code for these programs first, then go to the log of the debug session. Doing this will show you how the DEBUG-88 commands are used, and the results they produce.

```
8086/7/8/186 MACRO ASSEMBLER    Hanoi: The towers of Hanoi                    02/26/81  PAGE    1


SERIES-IV  8086/8087/8088 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE HANOI
OBJECT MODULE PLACED IN :F1:HANOI.OBJ
INVOCATION LINE CONTROLS:  DEBUG


LOC  OBJ                  LINE     SOURCE

                             1 +1  $print(:f1:hanoi.lst) object(:f1:hanoi.obj)
                             2 +1  $title('Hanoi: The towers of Hanoi')
                             3
                             4             name    Hanoi
                             5
                             6 +1  $include(:f1:group.inc)
                       =1    7
                       =1    8     ;THIS IS THE GROUP DECLARATION FOR THE small MODEL IN PLM86
                       =1    9
                       =1    10    CGROUP  GROUP  CODE
                       =1    11
                       =1    12    DGROUP  GROUP  DATA, CONST, STACK
                       =1    13
                       =1    14    ASSUME CS:CGROUP, DS:DGROUP
                       =1    15
                       =1    16 +1 $NOLIST
                             30
                             31    ; let Hanoi(N,S,B,D) be
                             32    ;   if N>0 then
                             33    ;   $( Hanoi(N-1, S,D,B);
                             34    ;      WriteStr("> Move disc |n from '|c' to '|c'.*N", N, S, D);
                             35    ;      Hanoi(N-1, B,S,D)
                             36    ;   $)
                             37    ;
                             38    ; and Main() be
                             39    ;   Hanoi(5,'S','B','D')
                             40
                             41
----                         42    Data    segment public 'data'
                             43
                             44            extrn   WrCh:NEAR, WriteStr:NEAR, WriteUnSignedInt:NEAR
                             45
                             46    ; Data definitions
0000 3E204D6F766520          47    Str1    db      '> Move disc ', 0
     6469736320
000C 00
000D 2066726F6D20            48    Str2    db      ' from ', 0
0013 00
0014 20746F20                49    Str3    db      ' to ', 0
0018 00
0019 0D                      50    Str4    db      13,10,0
001A 0A
001B 00
                             51
----                         52    Data    ends
                             53
                             54
----                         55    Code    segment public 'code'
                             56
  0004[]                     57    D       equ     word ptr [BP+4]
  0006[]                     58    B       equ     word ptr [BP+6]
  0008[]                     59    S       equ     word ptr [BP+8]
  000A[]                     60    N       equ     word ptr [BP+10]
                             61
                             62
                             63            public  Hanoi
0000                         64    Hanoi   proc    near
                             65
0000 55                      66            push    BP      ; create
0001 8BEC                    67            mov     BP,SP   ;   new context
                             68
0003 8B460A                  69            mov     ax,N            ; return
0006 0BC0                    70            or      ax,ax           ;   if N<=0
0008 7652                    71            jbe     Exit_Hanoi      ;
                             72
000A 50                      73            push    ax              ; save N on stack
                             74
000B 48                      75            dec     ax              ;
000C 50                      76            push    ax              ;
000D FF7608                  77            push    S               ; call Hanoi(
```

```
0010 FF7606                78      BUG1:   push    B               ;    N-1, S,D,B)
0013 FF7604                79      BUG2:   push    D               ;
0016 E8E7FF                80      BUG3:   call    Hanoi           ;
                           81
0019 8D1E0000        R     82              lea     bx,str1         ;
001D 53                    83              push    bx              ; call WriteStr(
001E E80000          E     84      Lab1:   call    WriteStr        ;      ref ('> Move disc ', STC))
                           85
0021 8BF4                  86              mov     SI,SP           ;
0023 8B04                  87              mov     ax,[SI]         ; retrieve N
0025 50                    88              push    ax              ; call WriteUnSignedInt(N)
0026 E80000          E     89      Lab2:   call    WriteUnSignedInt
                           90
0029 8D1E0D00        R     91              lea     bx,str2         ;
002D 53                    92              push    bx              ; call WriteStr(
002E E80000          E     93      Lab3:   call    WriteStr        ;      ref (' from ', STC))
                           94
0031 FF7608                95              push    S               ;
0034 E80000          E     96              call    WrCh            ; call WrCh(S)
                           97
0037 8D1E1400        R     98              lea     bx,str3         ;
003B 53                    99              push    bx              ; call WriteStr(
003C E80000          E    100              call    WriteStr        ;      ref (' to ', STC))
                          101
003F FF7604               102              push    D               ;
0042 E80000          E    103              call    WrCh            ; call WrCh
                          104
0045 8D1E1900        R    105              lea     bx,str4         ;
0049 53                   106              push    bx              ; call WriteStr(
004A E80000          E    107              call    WriteStr        ;      ref (CR,LF, STC))
                          108
004D 58                   109              pop     ax              ; destructively retrieve N
004E 48                   110              dec     ax              ;
004F 50                   111      Lab4:   push    ax              ; push N-1
0050 FF7606               112              push    B               ;
0053 FF7608               113              push    S               ;
0056 FF7604               114      Lab5:   push    D               ;
0059 E8A4FF               115              call    Hanoi           ; call Hanoi(N-1, B,S,D)
                          116
005C                      117      Exit_Hanoi:
005C 5D                   118              pop     BP
005D C20800               119              ret     8
                          120
                          121      Hanoi   endp
                          122
----                      123      Code    ends
                          124
                          125              end

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

```
P /M-86 COMPILER     The Towers of Hanoi driver                    02/26/81          PAGE   1


SERIES-IV  PL/M-86 V1.0 COMPILATION OF MODULE HANOIDRIVER
OBJECT MODULE PLACED IN :F1:HANDRV.OBJ
COMPILER INVOKED BY:  PLM86.86 :F1:HANDRV.P86



                $optimize(3) debug
                $title('The Towers of Hanoi driver')
   1            HanoiDriver: do;

                /* PEX */
   2    1       DQEXIT: procedure(ZZ1) external; declare (ZZ1) word; end;
                /* ENDPEX */

   5    1       Hanoi: procedure(N,S,B,D) external;
   6    2         declare N WORD,
                    (S,B,D) BYTE;
   7    2       end Hanoi;
```

```
   8   1       call Hanoi(5,'S','B','D');
   9   1       call DqExit(0);

  10   1       end HanoiDriver;


MODULE INFORMATION:

     CODE AREA SIZE     = 001CH     28D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 000AH     10D
     17 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

```
CONSOL , :F1:DEMON.LOG
D88                                     ;invoke debugger

LOAD ":F1:HANDRV.86" NOPUBLIC           ;load user program, with nopublic modifier

LIST ":F1:LIST.LOG"                     ;create log file

REG                                     ;display all registers

FLAG                                    ;display all flags

STACK 10                                ;display 10 words from the top of the stack

#5                                      ;display memory address of line #5

ASM .LAB1 TO .LAB2                      ;disassemble from LAB1 to LAB2

ASM .LAB3 LEN 5                         ;disassemble 5t lines from LAB3

DS,ES                                   ;display DS, ES
(DS = ES)                               ;boolean expression

RAX, RBX, RCX                           ;display reg. AX,BX,CX
AX <> BX                                ;boolean expression

EVA (AX + BX) * (CX / 3 - 2)            ;evaluate the expression in 4 different base

RADIX 10T                               ;set default radix to decimal

EVA 10                                  ;evaluate 10t

RAD 8T                                  ;set default radix to octal

EVA 13                                  ;evaluate 13q

EVA CS:IP LINE                          ;evaluate CS:IP for matching line number

BR0 = .BUG1                             ;set break point
GR = TILL BR0                           ;set GO-REG to activate break point
GO                                      ;execute till break
ASM CS:IP                               ;disassemble current cs:ip

REPEAT                                  ;
ASM CS:IP                               ;
UNTIL CS:IP = .BUG3                     ;this group of command will change the
STEP FROM .BUG2                         ;flow of program to get around the
END                                     ;bug on label BUG1 and BUG2
                                        ;
GO FROM .BUG1 TILL .BUG2                ;

GO FROM .BUG3 &
TILL .LAB2      OR &                    ;execute program until it encounter
&                                       ;first break point
.LAB1                                   ;note: continuation inserted
```

```
EVA .LAB2 SYMB                          ;evaluate the address symbolically

DEFINE .TEMP = 3                        ;define new symbol into unnamed module
COUNT .TEMP                             ;bounded iteration - repeat 3 times
IF CS:IP = .LAB2 THEN                   ;
'go command did not work right'         ;
ORIF CS:IP = .LAB1 THEN                 ;result of this compound command
'everything is fine'                    ;should be 'everything is fine'
END                                     ;
END                                     ;

REMOVE .TEMP                            ;remove symbol

GO TILL .LAB4                           ;execute until LAB4 is reached

REP                                     ;repeat until the condition is met
ASM CS:IP                               ;
UNTIL CS:IP = .LAB5                     ;
STEP                                    ;execute by single step
END                                     ;end of repeat compound command
.LAB5                                   ;.LAB5 = CS:IP
!!HANOI !LAB5                           ;dereference the symbol

PSTEP FROM .LAB5                        ;execute by single procedure step
                                      ;4 esc and 1 cr is inserted

DOMAIN ..HANOI                          ;set default domain module
EVA .HANOI SYMB                         ;module HANOI is the first module to be searched

BYTE CS:IP LEN 18                       ;display range of byte quantites

WORD CS:IP LEN 10                       ;display range of word quantites

POINTER 0 LEN 5 = 0                     ;assign value to pointer address

POI 0 LEN 5                             ;display range of pointer quantites


EXIT                                    ;exit from debugger
CONSOL , :VO:
ENDJOB
```

```
D88                                     ;invoke debugger
Series IV DEBUG 8088, X008
*
*LOAD "/WORKDISK/HANDRV.86" NOPUBLIC     ;load user program, with nopublic modifier
*
*LIST "/LOGDISK/LIST.LOG"                ;create log file
*
*REG                                     ;display all registers
>AX  = 4154h   BX  = 4B43h   CX  = 4305h   DX  = 4E4Fh
>CS  = 11B3h   SS  = 1219h   DS  = 1219h   ES  = 5453h
>IP  = 0000n   SP  = 08E8h   SI  = 5305n   DI  = 0000h
>RF  = 0200h   BP  = 2C96h
*
*FLAG                                    ;display all flags
>CFL=0 PFL=0 AFL=0 ZFL=0 SFL=0 TFL=0 IFL=1 DFL=0 OFL=0
*
*STACK 10                                ;display 10 words from the top of the stack
>[1219:08E8] FFFFh 0000n FFFFh 0000n FFFFn 0000n FFFFn 8000n 1218h 0000n
>[1219:08FC] 0800h 0D56h 0000n 0000n FFFFh 0000h
*
*#5                                      ;display memory address of line #5
>11B3:0000
*
*ASM .LAB1 TO .LAB2                      ;disassemble from LAB1 to LAB2
>[11B3:003E] E8F902          CALL  A=033A
>[11B3:0041] 8BF4            MOV   SI,SP
>[11B3:0043] 8B04            MOV   AX,[SI]
>[11B3:0045] 50              PUSH  AX
*
*ASM .LAB3 LEN 5                         ;disassemble 5t lines from LAB3
```

```
>[11B3:004E] E8E902              CALL  A=033A
>[11B3:0051] FF7608              PUSH  [BP+0008]
>[11B3:0054] E8B100              CALL  A=0108
>[11B3:0057] 8D1E3400            LEA   BX,[0034]
>[11B3:005B] 53                  PUSH  BX
*
*DS,ES                                  ;display DS, ES
>[DS] 1219h 5453h
*(DS = ES)                              ;boolean expression
>FALSE
*
*RAX, RBX, RCX                          ;display reg. AX,BX,CX
>[AX] 4154h 4B43h 4305h
*AX *> BX                               ;boolean expression
>TRUE
*
*EVA (AX + BX) * (CX / 3 - 2)           ;evaluate the expression in 4 different base
>1010100000100011y = 124043q = 43043t = A823h = #
*
*RADIX 10T                              ;set default radix to decimal
*
*EVA 10                                 ;evaluate 10t
>0000000000001010y = 000012q = 10t = 000Ah = ?
*
*RAD 8T                                 ;set default radix to octal
*
*EVA 13                                 ;evaluate 13q
>0000000000001011y = 000013q = 11t = 000Bh = ?
*
*EVA CS:IP LINE                         ;evaluate CS:IP for matching line number
>11B3:0000 = ..HANOIDRIVER#1
*
*BR0 = .BUG1                            ;set break point
*GR* = TILL BR0                         ;set GO-REG to activate break point
*GO                                     ;execute till break
>[11B3:0030] FF7606              PUSH  [BP+0006]
*ASM CS:IP                              ;disassemble current cs:ip
>[11B3:0030] FF7606              PUSH  [BP+0006]
*
*REPEAT                                 ;
.*ASM CS:IP                             ;
.*UNTIL CS:IP = .BUG3                   ;this group of command will change the
.*STEP FROM .BUG2                       ;flow of program to get around the
.*END                                   ;bug on label BUG1 and BUG2
>[11B3:0030] FF7606              PUSH  [BP+0006]
>[11B3:0036] E8E7FF              CALL  A=0020
*                                       ;
*GO FROM .BUG1 TILL .BUG2               ;
>[11B3:0033] FF7604              PUSH  [BP+0004]
*
*GO FROM .BUG3 &
**TILL .LAB2      OR &                  ;execute program until it encounter
**&                                     ;first break point
**.LAB1                                 ;note: continuation inserted
>[11B3:003E] E8F902              CALL  A=033A
*
*EVA .LAB2 SYMB                         ;evaluate the address symbolically
>11B3:0046 = ..HANOI.LAB2
*
*DEFINE .TEMP = 3                       ;define new symbol into unnamed module
*COUNT .TEMP                            ;bounded iteration - repeat 3 times
.*IF CS:IP = .LAB2 THEN                 ;
..*'go command did not work right'      ;
..*ORIF CS:IP = .LAB1 THEN              ;result of this compound command
..*'everything is fine'                 ;should be 'everything is fine'
..*END                                  ;
.*END                                   ;
>everything is fine
>everything is fine
>everything is fine
*
*REMOVE .TEMP                           ;remove symbol
*
*GO TILL .LAB4                          ;execute until LAB4 is reached
> Move disc 1 from S to B
>[11B3:006F] 50                  PUSH  AX
*
*REP                                    ;repeat until the condition is met
```

```
.*ASM CS:IP                              ;
.*UNTIL CS:IP = .LAB5                    ;
.*STEP                                   ;execute by single step
.*END                                    ;end of repeat compound command
>[11B3:006F] 50                 PUSH  AX
>[11B3:0070] FF7606             PUSH  [BP+0006]
>[11B3:0073] FF7608             PUSH  [BP+0008]
>[11B3:0076] FF7604             PUSH  [BP+0004]
*.LAB5                                   ;.LAB5 = CS:IP
>11B3:0076
*!!HANOI !LAB5                           ;dereference the symbol
>[11B3:0076] 073377q
*
*PSTEP FROM .LAB5                        ;execute by single procedure step
>[11B3:0076] FF7604             PUSH  [BP+0004] ? $
>[11B3:0079] E8A4FF             CALL  A=0020 ? $
>[11B3:007C] 5D                 POP   BP ? $
>[11B3:007D] C20800             RET   0008 ? $
>[11B3:007C] 5D                 POP   BP ?
*                                        ;4 esc and 1 cr is inserted
*
*DOMAIN ..HANOI                          ;set default domain module
*EVA .HANOI SYMB                         ;module HANOI is the first module to be searched
>11B3:0020 = ..HANOI.HANOI
*
*BYTE CS:IP LEN 18                       ;display range of byte quantites
>[11B3:007C] 5D C2 08 00 55 8B EC 8A 06 56 00 F6 D0 D0 D8 72  ]B??U?l??V?vPPXr
>[11B3:008C] 03 E9                                           ?i
*
*WORD CS:IP LEN 10                       ;display range of word quantites
>[11B3:007C] 141135q 000010q 105525q 105354q 053006q 173000q 150320q 071330q
>[11B3:008C] 164403q 000111q
*
*POINTER 0 LEN 5 = 0                     ;assign value to pointer address
*
*POI 0 LEN 5                             ;display range of pointer quantites
>[0000:0000] 0000:0000 0000:0000 0000:0000 0000:0000 0000:0000
*
*
*EXIT                                    ;exit from debugger
>DEB88 exit.
```

INDEX

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating).

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

(COUNTRY)

Please check here if you require a written reply ☐

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 U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.