

# **VAX BASIC Reference Manual**

Order Number: AA-HY16B-TE

**February 1990**

This manual provides reference material and syntax for VAX BASIC language elements.

**Revision/Update Information:** This revised manual supersedes the VAX BASIC Reference Manual, Version 3.3

**Operating System and Version:** VMS Version 5.0 or higher

**Software Version:** VAX BASIC Version 3.4

**digital equipment corporation  
maynard, massachusetts**

---

**First Printing, August 1986**  
**Revised, July 1988**  
**Revised, February 1990**

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.


Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

© Digital Equipment Corporation 1986, 1988, 1990.

All Rights Reserved.  
Printed in U.S.A.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA	MASSBUS	VAX RMS
DDIF	PrintServer 40	VAXstation
DEC	Q-bus	VMS
DECnet	ReGIS	VT
DECUS	ULTRIX	XUI
DECwindows	UNIBUS	
DIGITAL	VAX	
LN03	VAXcluster	

The following is a third-party trademark:

PostScript is a registered trademark of Adobe Systems, Inc.

ZK5433

# Contents

---

<b>Preface</b> .....	xv
<b>Summary of Technical Changes</b> .....	xix

---

## **Chapter 1 Program Elements and Structure**

<b>1.1 Components of Program Lines</b> .....	1-1
1.1.1 Line Numbers .....	1-2
1.1.1.1 Programs With Line Numbers .....	1-2
1.1.1.2 Programs Without Line Numbers .....	1-2
1.1.2 Labels .....	1-3
1.1.3 Statements .....	1-4
1.1.3.1 Keywords .....	1-5
1.1.3.2 Identifying Program Units .....	1-6
1.1.3.3 Single-Statement Lines and Continued Statements .....	1-7
1.1.3.4 Multi-Statement Lines .....	1-8
1.1.4 Compiler Directives .....	1-9
1.1.5 Line Terminators .....	1-10
1.1.6 Lexical Order .....	1-10
<b>1.2 VAX BASIC Character Set</b> .....	1-10
<b>1.3 VAX BASIC Data Types</b> .....	1-11
1.3.1 Implicit Data Typing .....	1-14
1.3.2 Explicit Data Typing .....	1-15
<b>1.4 Variables</b> .....	1-16
1.4.1 Variable Names .....	1-17
1.4.2 Implicitly Declared Variables .....	1-17

1.4.3	Explicitly Declared Variables	1-19
1.4.4	Subscripted Variables and Arrays	1-19
1.4.5	Initialization of Variables	1-22
<b>1.5</b>	<b>Constants</b>	1-23
1.5.1	Numeric Constants	1-24
1.5.1.1	Floating-Point Constants	1-24
1.5.1.2	Integer Constants	1-26
1.5.1.3	Packed Decimal Constants	1-27
1.5.2	String Constants	1-27
1.5.3	Named Constants	1-29
1.5.3.1	Naming Constants Within a Program Unit	1-29
1.5.3.2	Naming Constants External to a Program Unit	1-31
1.5.4	Explicit Literal Notation	1-31
1.5.5	Predefined Constants	1-34
<b>1.6</b>	<b>Expressions</b>	1-36
1.6.1	Numeric Expressions	1-36
1.6.1.1	Floating-Point and Integer Promotion Rules	1-38
1.6.1.2	DECIMAL Promotion Rules	1-39
1.6.2	String Expressions	1-41
1.6.3	Conditional Expressions	1-42
1.6.3.1	Numeric Relational Expressions	1-42
1.6.3.2	String Relational Expressions	1-43
1.6.3.3	Logical Expressions	1-45
1.6.4	Evaluating Expressions	1-50
<b>1.7</b>	<b>Program Documentation</b>	1-52
1.7.1	Comment Fields	1-52
1.7.2	REM Statements	1-54

---

## Chapter 2

### Environment Commands

! YOUR-COMMENT	2-2
\$ SYSTEM-COMMAND	2-4
APPEND	2-6
ASSIGN	2-8
COMPILE	2-10
CONTINUE	2-21
DELETE	2-22
EDIT	2-24
EXIT	2-27
HELP	2-29
IDENTIFY	2-31
INQUIRE	2-32

LIST AND LISTNH . . . . .	2-33
LOAD . . . . .	2-35
LOCK . . . . .	2-37
NEW . . . . .	2-38
OLD . . . . .	2-40
RENAME . . . . .	2-41
REPLACE . . . . .	2-43
RESEQUENCE . . . . .	2-45
RUN . . . . .	2-48
SAVE . . . . .	2-51
SCALE . . . . .	2-53
SCRATCH . . . . .	2-55
SEQUENCE . . . . .	2-56
SET . . . . .	2-58
SHOW . . . . .	2-60
UNSAVE . . . . .	2-62

---

**Chapter 3      Compiler Directives**

%ABORT . . . . .	3-2
%CROSS . . . . .	3-3
%DECLARED . . . . .	3-4
%IDENT . . . . .	3-6
%IF-%THEN-%ELSE-%END %IF . . . . .	3-8
%INCLUDE . . . . .	3-10
%LET . . . . .	3-15
%LIST . . . . .	3-17
%NOCROSS . . . . .	3-18
%NOLIST . . . . .	3-20
%PAGE . . . . .	3-21
%PRINT . . . . .	3-22
%REPORT . . . . .	3-23
%SBTTL . . . . .	3-25
%TITLE . . . . .	3-27
%VARIANT . . . . .	3-29

---

**Chapter 4     Statements and Functions**

ABS	4-2
ABS%	4-4
ASCII	4-6
ATN	4-7
BUFSIZ	4-9
CALL	4-10
CAUSE ERROR	4-16
CCPOS	4-18
CHAIN	4-20
CHANGE	4-22
CHR\$	4-25
CLOSE	4-27
COMMON	4-29
COMP%	4-35
CONTINUE	4-37
COS	4-39
CTRLC	4-41
CVT\$\$	4-43
CVTxx	4-44
DATA	4-48
DATE\$	4-50
DECIMAL	4-52
DECLARE	4-54
DEF	4-59
DEF*	4-65
DELETE	4-70
DET	4-72
DIF\$	4-74
DIMENSION	4-76
ECHO	4-81
EDIT\$	4-83
END	4-85
ERL	4-89
ERN\$	4-91
ERR	4-93
ERT\$	4-95
EXIT	4-97
EXP	4-100
EXTERNAL	4-102
FIELD	4-108
FIND	4-111
FIX	4-119

FNEND	4-121
FNEXIT	4-122
FOR	4-123
FORMAT\$	4-127
FREE	4-129
FSP\$	4-131
FUNCTION	4-133
FUNCTIONEND	4-137
FUNCTIONEXIT	4-138
GET	4-139
GETRFA	4-147
GOSUB	4-149
GOTO	4-151
HANDLER	4-153
IF	4-155
INKEY\$	4-158
INPUT	4-162
INPUT LINE	4-166
INSTR	4-169
INT	4-171
INTEGER	4-173
ITERATE	4-175
KILL	4-177
LBOUND	4-178
LEFT\$	4-180
LEN	4-182
LET	4-184
LINPUT	4-186
LOC	4-189
LOG	4-191
LOG10	4-193
LSET	4-195
MAG	4-197
MAGTAPE	4-199
MAP	4-201
MAP DYNAMIC	4-205
MAR	4-209
MARGIN	4-210
MAT	4-212
MAT INPUT	4-217
MAT LINPUT	4-220
MAT PRINT	4-223
MAT READ	4-226
MAX	4-229

MID\$	4-231
MIN	4-234
MOD	4-236
MOVE	4-238
NAME...AS	4-242
NEXT	4-244
NOECHO	4-246
NOMARGIN	4-248
NUM	4-250
NUM2	4-252
NUM\$	4-254
NUM1\$	4-256
ON ERROR GO BACK	4-258
ON ERROR GOTO	4-260
ON ERROR GOTO 0	4-263
ON...GOSUB	4-265
ON...GOTO	4-267
OPEN	4-269
OPTION	4-283
PLACE\$	4-288
POS	4-292
PRINT	4-294
PRINT USING	4-298
PROD\$	4-305
PROGRAM	4-308
PUT	4-310
QUO\$	4-314
RAD\$	4-317
RANDOMIZE	4-319
RCTRLC	4-321
RCTRLO	4-322
READ	4-324
REAL	4-326
RECORD	4-328
RECOUNT	4-333
REM	4-335
REMAP	4-337
RESET	4-342
RESTORE	4-343
RESUME	4-345
RETRY	4-348
RETURN	4-350
RIGHT\$	4-352
RMSSTATUS	4-354



RND .....	4-357
RSET .....	4-359
SCRATCH .....	4-361
SEG\$ .....	4-363
SELECT .....	4-365
SET PROMPT .....	4-368
SGN .....	4-370
SIN .....	4-371
SLEEP .....	4-373
SPACE\$ .....	4-375
SQR .....	4-376
STATUS .....	4-378
STOP .....	4-380
STR\$ .....	4-382
STRING\$ .....	4-384
SUB .....	4-386
SUBEND .....	4-390
SUBEXIT .....	4-391
SUM\$ .....	4-392
SWAP% .....	4-394
TAB .....	4-396
TAN .....	4-398
TIME .....	4-399
TIME\$ .....	4-401
TRM\$ .....	4-403
UBOUND .....	4-404
UNLESS .....	4-406
UNLOCK .....	4-407
UNTIL .....	4-409
UPDATE .....	4-411
VAL .....	4-413
VAL% .....	4-415
VMSSTATUS .....	4-417
WAIT .....	4-419
WHEN ERROR .....	4-421
WHILE .....	4-426
XLATE\$ .....	4-428

---

**Appendix A Transporting Programs Between VAX BASIC and BASIC-PLUS-2**

<b>A.1</b>	<b>Overview</b> .....	<b>A-1</b>
<b>A.2</b>	<b>Language-Specific Functionality</b> .....	<b>A-2</b>
<b>A.3</b>	<b>I/O Differences</b> .....	<b>A-3</b>
A.3.1	The MAGTAPE Function .....	A-3
A.3.2	The OPEN Statement .....	A-4
A.3.3	The PUT Statement .....	A-6
<b>A.4</b>	<b>Procedure Calling</b> .....	<b>A-6</b>
A.4.1	The CALL Statement .....	A-6
A.4.2	The CHAIN Statement .....	A-7
A.4.3	SYS and FIP SYS Calls .....	A-7
<b>A.5</b>	<b>Generated Errors</b> .....	<b>A-8</b>
<b>A.6</b>	<b>Miscellaneous Differences</b> .....	<b>A-9</b>
A.6.1	Data Types .....	A-10
A.6.2	The DEF and DEF* Statements .....	A-10
A.6.3	Default Integer Size .....	A-10
A.6.4	Integer Overflow .....	A-10
A.6.5	Line Numbers and Labels .....	A-10
A.6.6	The MAP and COMMON Statements .....	A-11
A.6.7	The MAP DYNAMIC Statement .....	A-11
A.6.8	The PRINT Statement .....	A-12
A.6.9	The PRINT USING Statement .....	A-12
A.6.10	The REPLACE Command .....	A-12
A.6.11	The SPEC% and PEEK Functions .....	A-12
A.6.12	String Comparisons .....	A-12
A.6.13	Assigning Symbols .....	A-13
A.6.14	The TIME Function .....	A-13
A.6.15	The TIME\$ Function .....	A-13

---

**Appendix B ANSI Minimal BASIC**

<b>B.1</b>	<b>Introduction</b> .....	<b>B-1</b>
<b>B.2</b>	<b>The /ANSI_STANDARD Qualifier</b> .....	<b>B-2</b>

<b>B.3</b>	<b>Extensions To ANSI Minimal BASIC Standard X3.60-1978</b> . . . . .	<b>B-2</b>
B.3.1	Program Format . . . . .	B-2
B.3.2	Statements . . . . .	B-3
B.3.3	Delimiters . . . . .	B-3
B.3.4	Variables . . . . .	B-3
B.3.5	Numeric Constants . . . . .	B-4
B.3.6	Data Input . . . . .	B-4
	B.3.6.1 Unquoted String Data . . . . .	B-5
	B.3.6.2 Null Input . . . . .	B-5
B.3.7	User-Defined Functions (the DEF Statement) . . . . .	B-6
B.3.8	Built-In Functions . . . . .	B-6
B.3.9	Arrays . . . . .	B-7
<b>B.4</b>	<b>Implementation-Defined Features</b> . . . . .	<b>B-7</b>
B.4.1	Initial Values for Variables . . . . .	B-8
B.4.2	Retention of Long Strings . . . . .	B-8
B.4.3	Accuracy of Evaluation of Numeric Expressions . . . . .	B-8
B.4.4	Machine Infinitesimal . . . . .	B-8
B.4.5	Machine Infinity . . . . .	B-8
B.4.6	Precision For Numeric Values . . . . .	B-9
B.4.7	Exrad-Width For Printing Numeric Representations . . . . .	B-9
B.4.8	Significance-Width For Printing Numeric Representations . . . . .	B-9
B.4.9	Print Zone Length . . . . .	B-9
B.4.10	Margin for Output Line . . . . .	B-9
B.4.11	Pseudorandom Number Sequence . . . . .	B-10
B.4.12	Unique Line Numbers . . . . .	B-10
B.4.13	Input Prompt . . . . .	B-10
B.4.14	End of Input Reply . . . . .	B-10
B.4.15	End of Print Line . . . . .	B-10
B.4.16	Exponentiation Operator . . . . .	B-10

---

**Appendix C ASCII Character Codes**

---

**Appendix D VAX BASIC Keywords**

---

## Index

---

### Examples

1-1	Referencing Label Names in VAX BASIC Programs . . . . .	1-4
1-2	Using the DECLARE Statement to Set Array Boundaries . . . . .	1-20
1-3	Naming Constants Within a Program Unit . . . . .	1-29
1-4	Associating Values with Named Constants . . . . .	1-30
1-5	Naming Constants as Expressions . . . . .	1-30
1-6	Declaring Constants Outside the Program Unit . . . . .	1-31
1-7	Specifying a Comment Field . . . . .	1-53
1-8	Using Comments Fields to Format a Program . . . . .	1-53
1-9	Using Empty Statements in a VAX BASIC Program . . . . .	1-54
1-10	Using REM Statements in VAX BASIC Programs . . . . .	1-55

---

### Figures

1-1	Representation of the Subscript Variable A%(4%,6%) . . . . .	1-22
1-2	Truth Tables . . . . .	1-48

---

### Tables

1-1	Keyword Space Requirements . . . . .	1-5
1-2	VAX BASIC Data Types . . . . .	1-13
1-3	Valid Floating-Point Variable Names . . . . .	1-18
1-4	Specifying Floating-Point Constants . . . . .	1-25
1-5	Numbers in E Notation . . . . .	1-25
1-6	Specifying Integer Constants . . . . .	1-26
1-7	Predefined Constants . . . . .	1-34
1-8	Arithmetic Operators . . . . .	1-37
1-9	Result Data Types in VAX BASIC Expressions . . . . .	1-39
1-10	VAX BASIC Result Data Types . . . . .	1-39
1-11	Result Data Types for DECIMAL Data . . . . .	1-40
1-12	Numeric Relational Operators . . . . .	1-43
1-13	String Relational Operators . . . . .	1-45
1-14	Logical Operators . . . . .	1-46
1-15	Numeric Operator Precedence . . . . .	1-51
2-1	Multiplying a Numeric Value with the SCALE Command . . . . .	2-54

4-1	VAX BASIC Parameter-Passing Mechanisms . . . . .	4-11
4-2	FILL Item Formats and Storage Allocations . . . . .	4-31
4-3	EDIT\$ Values . . . . .	4-84
4-4	MAGTAPE Functionality in VAX BASIC . . . . .	4-200
4-5	Rounding and Truncation of 123456.654321 . . . . .	4-290
4-6	VAX BASIC STATUS Bits . . . . .	4-379
4-7	TIME Function Values . . . . .	4-400
A-1	MAGTAPE Functionality in VAX BASIC . . . . .	A-4
A-2	RSTS/E Disk MODE Values and Corresponding BASIC Statements . . . . .	A-5
A-3	VAX BASIC Subset of RSTS/E SYS Calls . . . . .	A-7
A-4	VAX BASIC Subset of RSTS/E FIP SYS Calls . . . . .	A-8
A-5	Fatal Errors in VAX BASIC that are Warnings in BASIC-PLUS-2 . . . . .	A-9
A-6	VAX BASIC and BASIC-PLUS-2 TIME Function Differences . . . . .	A-13
C-1	ASCII Characters Reserved for National Use . . . . .	C-1
C-2	ASCII Codes . . . . .	C-2



# Preface

---

## Intended Audience

This manual describes the language elements and syntax of VAX BASIC. Readers are presumed to be familiar with VAX BASIC programming techniques. This manual provides reference material to be used in conjunction with the other two manuals in the documentation set.

## Associated Documents

This manual is one of three manuals that form the VAX BASIC document set. The other two manuals are as follows:

<i>VAX BASIC User Manual</i>	Provides tutorial material for VAX BASIC language constructs and information pertaining to programming with VAX BASIC on VMS systems
<i>Programming with VAX BASIC Graphics</i>	Provides tutorial and reference material on VAX BASIC graphics capabilities

You may also be interested in the following supplementary manuals:

- *VAX BASIC Syntax Summary*
- *Introduction to BASIC*
- *BASIC for Beginners*
- *More BASIC for Beginners*

---

## Document Structure

This manual consists of four chapters and four appendixes.

Chapter 1	Summarizes VAX BASIC program elements and structure
Chapter 2	Describes VAX BASIC environment commands
Chapter 3	Describes VAX BASIC compiler directives
Chapter 4	Describes VAX BASIC statements and functions
Appendix A	Summarizes transportability issues between BASIC-PLUS-2 and VAX BASIC
Appendix B	Explains how VAX BASIC conforms to the ANSI Minimal Standard for BASIC
Appendix C	Lists the ASCII codes
Appendix D	Lists VAX BASIC keywords

In Chapters 2, 3, and 4, the VAX BASIC language elements are arranged in alphabetical order within each part; each language element begins on a separate page. These chapters provide reference material on each VAX BASIC language element. The descriptions are arranged in alphabetical order and include the following sections:

<b>Overview</b>	An overview of what the statement or command does.
<b>Format</b>	The required syntax for the language element.
<b>Syntax Rules</b>	Any rules governing the use of parameters, separators, or other syntax items, effect of the statement or command on program execution, and any restrictions governing its use.
<b>Example</b>	One or more examples of the statement in a partial program. Where appropriate, explanatory text and program output are included.

---

## Conventions

This manual uses case of text, symbols, and mnemonics in syntactical diagrams. This symbology aids in providing more concise and exact descriptions of syntatic variables, rules, and format.

---

<b>Convention</b>	<b>Meaning</b>
\$ BASIC	In command-line examples, the user's response to a system prompt is printed in red; system prompts are printed in black.



<b>Convention</b>	<b>Meaning</b>
UPPERCASE letters	Uppercase letters are used for VAX BASIC keywords and must be coded exactly as shown.
lowercase letters	Lowercase letters are used to indicate user-supplied names or characters.
[ ]	Brackets enclose an optional portion of a format. Brackets around vertically stacked items indicate that you can select one of the enclosed items. You must include all punctuation as it appears in the brackets.
{ }	Braces enclose a mandatory portion of a format. Braces around vertically stacked items indicate that you must choose one of the enclosed items. You must include all punctuation as it appears in the braces.
.	A vertical ellipsis indicates that code which would normally be present is not shown.
...	An ellipsis indicates that the immediately preceding item can be repeated. An ellipsis following a format unit enclosed in brackets or braces means that you can repeat the entire unit. If repeated items or format units must be separated by commas, the ellipsis is preceded by a comma ( , ... ).

The following mnemonics are used in the syntax diagrams:

<b>Mnemonic</b>	<b>Meaning</b>
<i>angle</i>	An angle in radians or degrees
<i>array</i>	An array; syntax rules specify whether the bounds or dimensions can be specified
<i>chnl</i>	An I/O channel associated with a file
<i>com</i>	Specific to a COMMON block
<i>cond</i>	Conditional expression; indicates that an expression can be either logical or relational
<i>const</i>	A constant value
<i>data-type</i>	A data type keyword
<i>def</i>	Specific to a DEF function
<i>exp</i>	An expression
<i>file-spec</i>	A file specification
<i>func</i>	Specific to a FUNCTION subprogram

<b>Mnemonic</b>	<b>Meaning</b>
<i>int</i>	An integer value
<i>int-exp</i>	An expression that represents an integer value
<i>int-var</i>	A variable that contains an integer value
<i>label</i>	An alphanumeric statement label
<i>lex</i>	Lexical; used to indicate a component of a compiler directive
<i>line</i>	A statement line; may or may not be numbered
<i>line-num</i>	A statement line number
<i>lit</i>	A literal value, in quotation marks
<i>log-exp</i>	Logical expression
<i>map</i>	Specific to a MAP statement
<i>matrix</i>	A two-dimensional array
<i>name</i>	A name or identifier; indicates the declaration of a name or the name of a VAX BASIC structure, such as a SUB subprogram
<i>num</i>	A numeric value
<i>param-list</i>	A parameter list, such as for a SUB subprogram
<i>pass-mech</i>	A valid VAX BASIC passing mechanism
<i>real</i>	A floating-point value
<i>rel-exp</i>	Relational expression
<i>str</i>	A character string
<i>str-exp</i>	An expression that represents a character string
<i>str-var</i>	A variable that contains a character string
<i>sub</i>	Specific to a SUB subprogram
<i>target</i>	The target point of a branch statement; either a line number or a label
<i>unsubs-var</i>	Unsubscripted variable, as opposed to an array element
<i>var</i>	A variable

# Summary of Technical Changes

---

## Summary of New and Changed Features for Version 3.4

Version 3.4 of VAX BASIC includes support for the Program Design Facility (PDF). This new functionality can be implemented by appending the /DESIGN qualifier to the DCL command BASIC. See the *VAX BASIC User Manual* for more information on the /DESIGN qualifier.

This documentation update also contains numerous corrections and clarifications to Version 3.3 documentation. These documentation changes do not reflect new features.



# Program Elements and Structure

---

The building blocks of a VAX BASIC program are as follows:

- Program lines and their components
- The VAX BASIC character set
- VAX BASIC data types
- Variables and constants
- Expressions
- Program documentation

These building blocks are described in the following sections.

---

## 1.1 Components of Program Lines

A VAX BASIC program is a series of program lines that contain instructions for the VAX BASIC compiler. These instructions are in the form of *statements* that contain keywords, operators, and operands.

All VAX BASIC program lines can contain the following:

- Statements
- Line numbers or labels
- Compiler directives
- Comment fields
- A line terminator (carriage return)

Only a line terminator is required in a program line. The other elements are optional.

---

## 1.1.1 Line Numbers

Line numbers are no longer required in VAX BASIC programs; you can compile, link, and execute a program with or without line numbers. There are, however, different rules for writing programs with line numbers and for writing programs without line numbers. These differences are described in the following sections.

---

### 1.1.1.1 Programs With Line Numbers

If you are entering program lines directly into the BASIC environment in line mode, then only those statements with line numbers are allowed to start in the first column. Also, any programs entered in line mode must have an initial line number associated with the first program line.

A VAX BASIC line number must be a unique integer from 1 through 32767, and must be terminated by a space or tab. VAX BASIC ignores leading spaces, tabs, and zeros in line numbers. Embedded spaces, tabs, and commas cause VAX BASIC to signal an error.

In line mode, a line number followed by a carriage return does not constitute a VAX BASIC program line. A program line entered in line mode must contain a statement or a comment field. (Comment fields are discussed in Section 1.7.1). A new line number or a carriage return terminates a VAX BASIC program line.

A program line can contain any number of text lines; however, a text line cannot exceed 255 characters.

---

### 1.1.1.2 Programs Without Line Numbers

VAX BASIC searches for a line number on the first line of program text when you

- Load a program into the BASIC environment with the OLD command.
- Edit a program in the BASIC environment.

If no line number is found, then the following rules apply:

- No line numbers are allowed in that program module.
- References to the ERL function and a RESUME statement to a line number are not allowed.

- A subroutine will signal the same errors as it would if it were compiled with the /NOLINES qualifier. If an error is resignaled back to the caller, ERL gives the line number of the calling site, rather than the line number of the actual error in the subprogram.
- The REM statement is not allowed.

If your program contains multiple units, the point at which VAX BASIC breaks each program unit is determined by the placement of the statement that terminates each program unit. Any text that follows the program terminator becomes associated with the the following program unit. A program terminator can be any END statement, such as an END PROGRAM statement followed by any valid expression.

You cannot use the APPEND command in the BASIC environment, or a plus sign (+) at DCL level, to concatenate programs without line numbers.

Note that when you compile a program from DCL, or when you copy a program into the BASIC environment with the OLD command, program statements can begin in the first column.

Instead of line numbers, you can use labels to identify and reference program lines.

---

## 1.1.2 Labels

A label is a 1- to 31-character name that identifies a statement or block of statements. The label name must begin with a letter; the remaining characters, if any, can be any combination of letters, digits, dollar signs (\$), underscores (\_), or periods (.). If the program is being entered in line mode, and therefore contains line numbers, then only line numbers and immediate mode statements can begin in the first character position.

A label name must be separated from the statement it identifies with a colon (:). For example:

```
Yes_routine: PRINT "Your answer is YES."
```

The colon is not part of the label name. It informs VAX BASIC that the label is being defined rather than referenced. Consequently, the colon is not allowed when you use a label to reference a statement. For example:

```
200      GOTO Yes_routine
```

You can reference a label almost anywhere you can reference a line number. However, there are the following exceptions:

- You cannot compare a label with the value returned by the ERL function.
- You cannot reference a label in an IF...THEN...ELSE statement without using the keyword GOTO or GO TO. You can use the implied GOTO form only to reference a line number. In Example 1-1, the GOTO keyword is not required in statement 100 because the reference is to a line number. However, the GOTO keyword is required in statement 200 because the references are to labels.

---

### Example 1-1: Referencing Label Names in VAX BASIC Programs

---

```
100 IF A% = B%  
    THEN 1000  
    ELSE 1050  
  
200 IF A$ = "YES"  
    THEN GOTO Yes  
    ELSE GOTO No
```

---

---

## 1.1.3 Statements

A VAX BASIC statement consists of a statement keyword and optional operators and operands. For example, both of the following statements are valid:

```
LET A% = 534% + (SUM% - DIF%)  
PRINT A%
```

VAX BASIC statements can be either executable or nonexecutable:

- Executable statements perform operations (for example, PRINT, GOTO, and READ).
- Nonexecutable statements describe the characteristics and arrangement of data, specify usage information, and serve as comments in the source program (for example, DATA, DECLARE, and REM).

VAX BASIC can accept and process one statement on a line of text, several statements on a line of text, multiple statements on multiple lines of text, and single statements continued over several lines of text. Each line of program text is associated with the last specified line number, and each must contain a keyword.



---

### 1.1.3.1 Keywords

Every VAX BASIC statement except LET and empty statements must begin with a keyword. A keyword is a reserved element of the VAX BASIC language. Keywords are used to

- Define data and user identifiers.
- Perform operations.
- Invoke built-in functions.

#### NOTE

Keywords are reserved words and cannot be used as user identifiers, such as variable names, labels, or names for MAP or COMMON areas.

Keywords cannot be used in any context other than as VAX BASIC keywords. The assignment `STRING$ = "YES"`, for example, is invalid because `STRING$` is a reserved VAX BASIC keyword and therefore cannot be used as a variable. Appendix D in this manual contains a list of the VAX BASIC keywords.

A VAX BASIC keyword cannot have embedded spaces and cannot be split across lines of text. There must be a space, tab, or special character such as a comma between the keyword and any other variable or operator.

Some keywords use two words. In this case, their spacing requirements vary, as shown in Table 1-1.

**Table 1-1: Keyword Space Requirements**

<b>Optional Space</b>	<b>Required Space</b>	<b>No Space</b>
GO TO	BY DESC	FNEND
GO SUB	BY REF	FNEXIT
ON ERROR	BY VALUE	FUNCTIONEND
	END DEF	FUNCTIONEXIT
	END FUNCTION	NOECHO
	END GROUP	NOMARGIN

(continued on next page)

**Table 1–1 (Cont.): Keyword Space Requirements**

<b>Optional Space</b>	<b>Required Space</b>	<b>No Space</b>
	END IF	SUBEND
	END PROGRAM	SUBEXIT
	END RECORD	
	END SELECT	
	END SUB	
	EXIT DEF	
	EXIT FUNCTION	
	EXIT SUB	
	INPUT LINE	
	MAP DYNAMIC	
	MAT INPUT	
	MAT LINPUT	
	MAT PRINT	
	MAT READ	

### 1.1.3.2 Identifying Program Units

You can delimit a main program compilation unit with the **PROGRAM** and **END PROGRAM** statements, as shown in the following example:

```
PROGRAM Sort_out
  .
  .
  .
END PROGRAM
```

This allows you to identify a program with a name other than the file name. The **PROGRAM** name must not be the same as that of any **SUB**, **FUNCTION**, or **PICTURE** subprogram.

If you include the **PROGRAM** statement in your program, the name you specify becomes the module name of the compiled source. This feature is useful when you use object libraries, because the librarian stores modules by their module names rather than by their file names. Similarly, module names are used by the VMS Debugger and the VMS Linker.

For more information about program units, see the *VAX BASIC User Manual*.

### 1.1.3.3 Single-Statement Lines and Continued Statements

A single-statement line consists of one statement on one numbered line, or one statement continued over two or more text lines. For example:

```
30 PRINT B * C / 12
```

This single-statement line has a line number, the keyword (PRINT), the operators (\*, /), and the operands (B, C, 12).

You can have a single statement span several text lines by typing an ampersand (&) and the RETURN key. Note that only spaces or tabs are valid between the ampersand and the carriage return. For example:

```
OPEN "SAMPLE.DAT" AS FILE 2%, & RET  
    SEQUENTIAL VARIABLE, & RET  
    MAP ABC
```

The ampersand continuation character may be used but is not required for continued REM statements. The following example is valid:

```
REM This is a remark  
    And this is also a remark
```

You can continue any VAX BASIC statement, but you cannot continue a string literal or VAX BASIC keyword. The following example generates the error message "Unterminated string literal".

```
PRINT "IF-THEN-ELSE- &  
    END-IF"
```

This example is valid:

```
PRINT "IF-";           &  
    "THEN-";           &  
    "ELSE-";           &  
    "END-";            &  
    "IF"
```

A more efficient way to continue string literals is to use the string concatenation operator (+) as follows:

```
PRINT "IF-"           &  
    + "THEN-"         &  
    + "ELSE-"         &  
    + "END-"          &  
    + "IF"
```

VAX BASIC concatenates the four string literals at compilation and stores them as one string. When the PRINT statement executes, VAX BASIC displays the one concatenated string literal rather than four separate string literals, thereby causing your program to execute faster and more efficiently.

### 1.1.3.4 Multi-Statement Lines

Multi-statement lines contain several statements on one line of text or multiple statements on separate lines of text. All the statements on a multi-statement line are associated with a single line of code.

Multiple statements on one line of text must be separated by backslashes (\). For example:

```
40 PRINT A \ PRINT V \ PRINT G
```

Because all statements are on the same program line, any reference to line number 40 refers to all three statements and execution begins with the first statement on the line. For example, VAX BASIC cannot execute the second statement without executing the first statement.

You can also write a multi-statement program line that associates all statements with a single line number by placing each statement on a separate line. VAX BASIC assumes that such an unnumbered line of text is either a new statement or an IF statement clause.

In the following example, each line of text begins with a VAX BASIC statement and each statement is associated with line number 400.

```
400 PRINT A
      PRINT B
      PRINT "FINISHED"
```

VAX BASIC also recognizes IF statement keywords on a new line of text and associates such keywords with the preceding IF statement. For example:

```
100 REM      Determine if the user's response
              was YES or NO.
200 IF (A$ = "YES") OR (A$ = "Y")
      THEN PRINT "You typed YES"
      ELSE PRINT "You typed NO"
      STOP
      END IF
```

The VAX BASIC compiler assigns listing line numbers to the statements as they occur physically in the program. For example:

```
1 100 REM      Determine if the user's response
2              was YES or NO.
3 200 IF (A$ = "YES") OR (A$ = "Y")
4      THEN PRINT "You typed YES"
5      ELSE PRINT "You typed NO"
6      STOP
7      END IF
```

You cannot use listing line numbers as targets of branch statements. The target of a branch statement such as GOTO must be a line number or a label. See the *VAX BASIC User Manual* for more information on listing file formats.

You can use any VAX BASIC statement in a multi-statement line. Since the VAX BASIC compiler ignores all text following a REM keyword until it reaches a new line number, a REM statement must be the last statement on a multi-statement line. REM statements are disallowed in programs without line numbers.

In the environment, a leading space or tab not followed by a line number implies a new statement in a multi-statement line, compiler commands and immediate mode statements cannot be preceded by a space, tab, or line number. If you enter a compiler command or immediate mode statement, you cannot add more continuation lines to the last program line. If you attempt to do so, VAX BASIC signals the error "Unknown command input."

---

## 1.1.4 Compiler Directives

Compiler directives are instructions for the VAX BASIC compiler. These instructions cause the VAX BASIC compiler to perform certain operations as it compiles the program.

By including compiler directives in a program, you can:

- Place program titles and subtitles in the header that appears on each page of the listing file
- Place a program version identification string in both the listing file and object module
- Start or stop the inclusion of listing information for selected parts of a program
- Start or stop the inclusion of cross reference information for selected parts of a program
- Include VAX BASIC code from another source file or a text library
- Conditionally compile parts of a program
- Terminate compilation
- Include CDD record definitions in a VAX BASIC program
- Display messages during the compilation

Follow these rules when using compiler directives:

- Compiler directives must begin with a percent sign

- Compiler directives must be the only text on the line (except for %IF-%THEN-%ELSE-%END-%IF)
- Compiler directives cannot appear within a quoted string
- Compiler directives can be preceded by an optional line number

See the *VAX BASIC User Manual* and Chapter 2 in this manual for more information on compiler directives.

---

### 1.1.5 Line Terminators

In the BASIC environment, a program line ends with a carriage return/line feed combination (the RETURN key) followed by an optional space or tab or a new line number. An ampersand followed by a carriage return ends a line of text, but not the program line. Note that spaces and tabs are valid between the ampersand and the carriage return; no other characters are valid. When line numbers are present, all statements between the first line number and the next line number are associated with the first line number.

---

### 1.1.6 Lexical Order

*Lexical order* refers to the order in which the statements in a program are compiled. In general terms, VAX BASIC compiles program lines in sequential order: multiple statements on a line of text are processed from left to right, and lines of text are processed from top to bottom. Note that certain VAX BASIC statements can alter this flow of compilation, for example GOSUB and GOTO.

Some VAX BASIC statements, such as comments and MAP declarations, are nonexecutable. If program control passes to a nonexecutable statement, the VAX BASIC compiler executes the first statement that lexically follows the nonexecutable statement.

---

## 1.2 VAX BASIC Character Set

VAX BASIC uses the full ASCII character set. This includes

- The letters A through Z, both upper- and lowercase
- The digits 0 through 9
- Special characters

Appendix C in this manual lists the full ASCII character set and character values.

The VAX BASIC compiler does not distinguish between upper- and lowercase letters except in string literals or within a DATA statement. The VAX BASIC compiler does not process characters in REM statements or comment fields, nor does it process nonprinting characters unless they are part of a string literal.

In string literals, VAX BASIC processes:

- Lowercase letters as lowercase
- Nonprinting characters

The ASCII character NUL (ASCII code 0) and line terminators cannot appear in a string literal. Use the CHR\$ function or explicit literal notation to use these characters and terminators.

You can use nonprinting characters in your program, for example, in string constants, but to do so you must use one of the following:

- A predefined constant such as ESC or DEL
- The CHR\$ function to specify an ASCII value
- Explicit literal notation

See Section 1.5.4 for more information on explicit literal notation.

---

## 1.3 VAX BASIC Data Types

Each unit of data in a VAX BASIC program has a specific data type that determines how that unit of data is to be interpreted and manipulated by the VAX BASIC compiler. This data type also determines how many storage bits make up the unit of data.

VAX BASIC recognizes five primary data types:

- Integer
- Floating-point
- Character string
- Packed decimal
- Record's file address

Integer data is stored as binary values in a byte, word, or longword. These values correspond to the VAX BASIC data type keywords BYTE, WORD, and LONG; these are all subtypes of the type INTEGER.

Floating-point values are stored using a signed exponent and a binary fraction. VAX BASIC allows four floating-point formats: single, double, G\_floating, and H\_floating. These formats correspond to the VAX BASIC data type keywords SINGLE, DOUBLE, GFLOAT, AND HFLOAT; these are all subtypes of the type REAL.

VAX BASIC packed decimal data is stored in a string of bytes. Refer to the *VAX BASIC User Manual* for more information on the storage of packed decimal data.

Character data consists of strings of bytes containing ASCII code as binary data. The first character in the string is stored in the first byte, the second character is stored in the second byte, and so on. VAX BASIC allows up to 65535 characters for a STRING data element.

In addition to this data type, VAX BASIC also recognizes a special RFA data type to provide information about a record's file address. An RFA uniquely specifies a record in a file: you can access RMS files of any organization by record's file address. By specifying the disk address of a record, RMS retrieves the record at that address. Accessing records by RFA is more efficient and faster than other forms of random record access. The RFA data type can only be used for

- RFA operations (the GETRFA function and the GET and FIND statements)
- Assignments to other variables of the RFA data type
- Comparisons with other variables of the RFA data type with the equal to (=) and not equal to (<>) relational operators
- Formal and actual parameters
- DEF and function results

You cannot declare a constant of the RFA data type, nor can you use RFA variables for any arithmetic operations.

The RFA data type requires 6 bytes of information. See the *VAX BASIC User Manual* for more information on Record File Addresses and the RFA data type.

For the DECIMAL(d,s) data type, you can specify the total number of digits (d) in the data type and the number of digits to the right of the decimal point (s). For instance, DECIMAL(10,3) specifies decimal data with a total of 10 digits, 3 of which are to the right of the decimal point.



Table 1–2 lists VAX BASIC data type keywords and summarizes VAX BASIC data types.

**Table 1–2: VAX BASIC Data Types**

<b>Data Type Keyword</b>	<b>Size</b>	<b>Range</b>	<b>Precision (decimal) (digits)</b>
<b>Integer</b>			
BYTE	8 bits	–128 to +127	NA
WORD	16 bits	–32768 to +32767	NA
LONG	32 bits	–2147483648 to +2147483647	NA
<b>Real</b>			
SINGLE	32 bits	.29 * 10 <sup>–38</sup> to 1.7 * 10 <sup>38</sup>	6
DOUBLE	64 bits	.29 * 10 <sup>–38</sup> to 1.7 * 10 <sup>38</sup>	16
GFLOAT	64 bits	.56 * 10 <sup>–308</sup> to .90 * 10 <sup>308</sup>	15
HFLOAT	128 bits	.84 * 10 <sup>–4932</sup> to .59 * 10 <sup>4932</sup>	33
<b>Decimal</b>			
DECIMAL(d,s)	0 to 16 bytes	1 * 10 <sup>–31</sup> to 1 * 10 <sup>31</sup>	NA
<b>String</b>			
STRING	One character per byte	Max = 65535	NA
<b>RFA</b>			
RFA	6 bytes	NA	NA

In Table 1–2, REAL and INTEGER are generic data type keywords that specify floating-point and integer storage, respectively. If you use the REAL or INTEGER keywords to type data, the actual data type (SINGLE,

DOUBLE, GFLOAT, HFLOAT, BYTE, WORD, or LONG) depends on the current default. If you do not explicitly type one of the appropriate subtypes, VAX BASIC uses the current subtype defaults for REAL and INTEGER.

You can specify data type defaults in the BASIC environment with the SET and COMPILE commands, or from DCL level with the DCL command BASIC. You can also specify whether program values are to be typed implicitly or explicitly. The following sections discuss data type defaults and implicit and explicit data typing.

---

### 1.3.1 Implicit Data Typing

You can implicitly assign a data format to program values by adding a suffix to the variable name or constant value. If you do not specify any suffix, the variable or constant is assigned the current default data type. The following rules apply for implicit data typing:

- A dollar sign suffix (\$) specifies STRING storage.
- A percent sign suffix (%) specifies INTEGER storage.
- No special suffix character specifies storage of the default type, which can be INTEGER, REAL, or DECIMAL.

With implicit data typing, the range and precision for program values are determined by the corresponding default data sizes or subtypes:

- BYTE, WORD, or LONG for INTEGER values
- SINGLE, DOUBLE, GFLOAT, or HFLOAT for REAL values
- The default (d,s) values for DECIMAL values

The default data type is determined by one of the following:

- The system default (REAL)
- The data type set for the BASIC environment with the SET or COMPILE compiler command
- The data type set for the BASIC environment with the BASIC statement OPTION
- The data type set for VAX BASIC with a qualifier for the DCL command BASIC

The VAX BASIC qualifiers for the SET and COMPILE commands are described in Chapter 2 of this manual. The qualifiers for the DCL command BASIC are listed in the *VAX BASIC User Manual*.

Note that if you compile your program with the `/TYPE_DEFAULT=EXPLICIT` qualifier (on either the DCL command `BASIC` or `VAX BASIC` command `COMPILE`), you can still add the appropriate suffixes to your variable names or constant values. The suffixes are useful because they identify the data type of the variable or constant immediately; the reader does not have to refer to the declarations at the top of the program to see which data type applies to a particular program value. However, with the `/TYPE_DEFAULT=EXPLICIT` qualifier, you must still explicitly assign data types to all program values or `VAX BASIC` signals an error.

It is considered good programming practice to use explicit data typing because implicit data typing is dependent on compilation defaults. These defaults may change, thereby affecting the precision of the program values.

---

### 1.3.2 Explicit Data Typing

Explicit data typing means that you use a declarative statement to specify the type, range, and precision of your program values. Declarative statements associate attributes such as data type and value with user identifiers.

In the following example, the first `DECLARE` statement associates the constant value `03060` and the `STRING` data type with a constant named `zip_code`. The second `DECLARE` statement associates the `STRING` data type with `emp_name`, the `DOUBLE` data type with `with_tax`, and the `SINGLE` data type with `int_rate`. No constant values are associated with identifiers in the second `DECLARE` statement because they are variable names.

```
DECLARE STRING CONSTANT zip_code = "03060"  
DECLARE STRING emp_name, DOUBLE with_tax, SINGLE int_rate
```

With explicit data typing, each program variable within a program can have a different range and precision. You can explicitly assign data types to variables, constants, arrays, parameters, and functions; therefore, integer data does not have to take the compilation default types. Explicit data typing gives you more control over your program.

Using the `REAL` and `INTEGER` keywords to explicitly type program values allows you to write programs that are transportable across systems, because these data type keywords specify that all floating-point and integer data take the current defaults for `REAL` and `INTEGER`. The data type `INTEGER`, for example, specifies only that the constant or variable is an integer. The actual subtype (`BYTE`, `WORD`, or `LONG`) depends on the default set with the `COMPILE` or `SET` command, the DCL command `BASIC`, or with the `OPTION` statement.

You can also specify a particular data type size for values declared INTEGER or REAL with compilation qualifiers. The /DOUBLE qualifier, for instance, specifies that all data typed REAL is to be treated as double-precision data.

The /TYPE\_DEFAULT=EXPLICIT qualifier or OPTION TYPE=EXPLICIT statement allows you to specify that all program data must be explicitly typed. Compiling a program with /TYPE\_DEFAULT= EXPLICIT or specifying OPTION TYPE=EXPLICIT means that any program value not explicitly declared causes VAX BASIC to signal an error.

For new applications, DIGITAL recommends using VAX BASIC's explicit data typing features. See the *VAX BASIC User Manual* for more information.

---

## 1.4 Variables

A variable is a named quantity whose value can change during program execution. Each variable name refers to a location in the program's storage area. Each location can hold only one value at a time. Variables of all data types can have subscripts that indicate their position in an array. You can declare variables implicitly or explicitly.

Depending on the program operations specified, the value of a variable can change from statement to statement. VAX BASIC uses the most recently assigned value when performing calculations. This value remains in effect until a new value is assigned to the variable.

VAX BASIC accepts the following general types of variables:

- Floating-point
- Integer
- String
- RFA
- Packed decimal
- Record

See the *VAX BASIC User Manual* for more information on RFA variables and RECORD data structures.

---

## 1.4.1 Variable Names

The name given to a variable depends on whether the variable is internal or external to the program and whether the variable is implicitly or explicitly declared.

All variable names must conform to the following rules:

- The name can have from 1 to 31 characters.
- The name has no embedded spaces.
- The first character of the name must be an upper- or lowercase alphabetic character (A through Z).
- The last character of the name can be either a dollar sign (\$) to indicate a string variable or a percent sign (%) to indicate an integer variable. If the last character is neither a dollar sign nor a percent sign, the name indicates a variable of the default type.
- The remaining characters, if present, can be any combination of upper- or lowercase letters (A through Z), numbers (0 through 9), dollar signs (\$), underscores (\_), or periods (.). The use of underscores in variable names helps improve readability and is preferred to the use of periods.
- The name of an external, explicitly declared variable in VAX BASIC must follow the rules for naming any explicitly declared variable.

Note that a program cannot have external, implicitly declared variables since all implicitly declared names except SUB subprogram names are internal to the program.

---

## 1.4.2 Implicitly Declared Variables

VAX BASIC accepts three types of implicitly declared variables:

- Integer
- String
- Floating-point (or the default data type)

The name of an implicitly declared variable defines its data type. Integer variables end with a percent sign (%), string variables end with a dollar sign (\$), and variables of the default type (usually floating-point) end with any allowable character except a percent sign or dollar sign. All three types of variables must conform to the rules listed in Section 1.4.1 for naming variables. The current data type default (INTEGER, REAL, or DECIMAL)

determines the data type of implicitly declared variables that do not end in a percent sign or dollar sign.

A floating-point variable is a named location that stores a single floating-point value. The current default size for floating-point numbers (SINGLE, DOUBLE, GFLOAT or HFLOAT) determines the data type of the floating-point variable. Table 1-3 lists valid floating-point variable names:

**Table 1-3: Valid Floating-Point Variable Names**

C	L . . . 5	ID_NUMBER
M1	BIG47	STORAGE_LOCATION_FOR_XX
F67T_J	Z2.	STRESS_VALUE

If a numeric value of a different data type is assigned to a floating-point variable, VAX BASIC converts the value to a floating-point number.

An integer variable is a named location that stores a single integer value. The current default size for integers (BYTE, WORD, or LONG) determines the data type of an integer variable. The following are valid integer variable names:

ABCDEFG%	C_8%	RECORD_NUMBER%
B%	D6E7%	THE_VALUE_I_WANT%

If the default data type is INTEGER, the percent suffix (%) is not necessary.

If you assign a floating-point or decimal value to an integer variable, VAX BASIC truncates the fractional portion of the value. It does not round to the nearest integer. For example:

```
100      B% = -5.7
```

VAX BASIC assigns the value -5 to the integer variable, not -6.

A string variable is a named location that stores strings. The following are valid string variable names:

C1\$	M\$	EMPLOYEE_NAME\$
L_6\$	F34G\$	TARGET_RECORD\$
ABC1\$	T.\$	STORAGE_SHELF_IDENTIFIER\$

Strings have both value and length. VAX BASIC sets all string variables to a default length of zero before program execution begins, with the exception of those variables in a COMMON, MAP, virtual array, or record definition. See the COMMON statement and the MAP statement in Chapter 4 of this manual for information on string length in COMMON and MAP areas. See

the *VAX BASIC User Manual* for information on default string length in virtual arrays.

During execution, the length of a character string associated with a string variable can vary from zero (signifying a null or empty string) to 65535 characters.

---

### 1.4.3 Explicitly Declared Variables

VAX BASIC lets you explicitly assign a data type to a variable or an array. For example:

```
DECLARE DOUBLE Interest_rate
```

Data type keywords are described in Section 1.1.3.1. For more information on explicit declaration of variables, see the sections on the COMMON, DECLARE, DIMENSION, DEF, FUNCTION, EXTERNAL, MAP, and SUB statements in Chapter 4 of this manual. See also the *VAX BASIC User Manual*.

---

### 1.4.4 Subscripted Variables and Arrays

A subscripted variable references part of an array. Arrays can be of any valid data type. Subscripted variables and arrays follow the same naming conventions as unsubscripted variables. Subscripts follow the variable name in parentheses and define the variable's position in the array. When you create an array, you specify the maximum size of the array (the bounds) in parentheses following the array name.

In Example 1–2, the DECLARE statement sets the bounds of the array *emp\_name* to 1000. Therefore, the maximum value for an *emp\_name* subscript is 1000. The bounds of the array define the maximum value for a subscript of that array.

## Example 1–2: Using the DECLARE Statement to Set Array Boundaries

---

```
DECLARE STRING emp_name(1000)
FOR I% = 0% TO 1000%
    INPUT "Employee name";emp_name(I%)
NEXT I%
```

---

Subscripts can be any positive LONG integer value between 0 and 2147483647.

### NOTE

By default, VAX BASIC signals an error if a subscript is bigger than the allowable range. Note, however, that the amount of storage the system can allocate depends on available memory. Therefore, very large arrays may cause an internal allocation error even though the subscript is still within the specified range.

An array is a set of data ordered in any number of dimensions. A one-dimensional array, like *emp\_name*(1000), is called a *list* or *vector*. A two-dimensional array, like *payroll\_data*(5,5), is called a *matrix*. An array of more than two dimensions, like *big\_array*(15,9,2), is called a *tensor*.

As a default, VAX BASIC arrays are always zero-based. The number of elements in any dimension includes element number zero. For example, the array *emp\_name* contains 1001 elements, since VAX BASIC allocates element zero. *Payroll\_data*(5,5) contains 36 elements because VAX BASIC allocates row and column zero.

Often, however, applications call for arrays that are not zero-based. In VAX BASIC, you can define arrays that are not zero-based by specifying a lower bound, as well as an upper bound, for the subscripts. In this way, you can create an array with arbitrary starting and ending points. For example, you might want to create array *birth\_rate* that holds the annual birth rate statistics for the years 1950 through 1985:

```
DECLARE birth_rate(1950 TO 1985)
```

Lower bounds are not allowed with virtual arrays or arrays used in MAT statements. However, you can specify lower bounds for any or all dimensions of a compile-time dimensioned array. If a multi-dimensional array is declared with lower bounds specified for some dimensions and not others, zero will be used for those dimensions without lower bounds.

You can use the UBOUND and LBOUND functions to determine the upper and lower bounds of an array. For a description of these functions, see Chapter 4 of this manual.



For all arrays except virtual arrays, the total number of array elements cannot exceed 2147483647. Note, however, that this is a theoretical value; the actual maximum size of an array which you can declare depends on the configuration of your system.

VAX BASIC arrays can have up to 32 dimensions. You can specify the type of data the array contains with data type keywords. Table 1–2 lists VAX BASIC data types.

An element in a one-dimensional array has a variable name followed by one subscript in parentheses. There can be a space between the array name and the subscripts. For example:

```
A (6%)  
B (6%)  
C$ (6%)
```

A(6%) refers to the seventh item in this list:

```
A(0%)  A(1%)  A(2%)  A(3%)  A(4%)  A(5%)  A(6%)
```

An element in a two-dimensional array has two subscripts, in parentheses, following the variable name. The first subscript specifies the row number and the second subscript specifies the column. Use a comma to separate the subscripts. You can include a space between the array name and the subscripts if you like. For example:

```
A (7%,2%)  A%(4%,6%)  A$(10%,10%)
```

In figure Figure 1–1, the arrow points to the element specified by the subscripted variable A%(4%,6%):

**Figure 1-1: Representation of the Subscript Variable A%(4%,6%)**

---

C O L U M N S	
	0 1 2 3 4 5 6
R 0	0 0 0 0 0 0 0
O 1	0 0 0 0 0 0 0
W 2	0 0 0 0 0 0 0
S 3	0 0 0 0 0 0 0
4	0 0 0 0 0 0 0 ← A%(4%, 6%)

ZK-5549-GE

---

An element in an array has as many subscripts as there are dimensions.

Although a program can contain a variable and an array with the same name, this is poor programming practice. Variable *A* and the array *A(3%,3%)* are \*separate\* entities and are stored in completely separate locations, so it is a good idea to give them different names.

Note that a program cannot contain two arrays with the same name but a different number of subscripts. For example, the arrays *A(3%)* and *A(3%,3%)* are invalid in the same program.

VAX BASIC arrays can be redimensioned at run time. See the *VAX BASIC User Manual* for more information on arrays.

---

### 1.4.5 Initialization of Variables

VAX BASIC sets variables to zero or null values at the start of program execution. Variables initialized by VAX BASIC include:

- Numeric variables and in-storage array elements (except those in MAP or COMMON statements).
- String variables (except those in MAP or COMMON statements).
- Variables in subprograms. Subprogram variables are initialized to zero or the null string each time the subprogram is called.

VAX BASIC does not initialize the following:

- Virtual arrays
- Variables in MAP and COMMON areas

---

## 1.5 Constants

A constant is a numeric or character literal that does not change during program execution. A constant can also be named and associated with a data type. VAX BASIC allows the following types of constants:

- Numeric:
  - Floating-point
  - Integer
  - Packed decimal
- String (ASCII characters enclosed in quotation marks)

A constant of any of the above data types can be named with the DECLARE CONSTANT statement. You can then refer to the constant by name in your program. Refer to Section 1.5.3 for information on naming constants.

You can use the OPTION statement to declare a default data type for all constants in your program. This statement allows you to specify a data type for only the constants in your program; you can specify a different data type for variables. You can also use a special numeric literal notation to specify the value and data type of a numeric literal. Numeric literal notation is discussed in Section 1.5.4.

If you do not specify a data type for a numeric constant with the DECLARE CONSTANT statement or with numeric literal notation, the type and size of the constant is determined by the default REAL, INTEGER, or DECIMAL type set with the DCL command BASIC, the VAX BASIC SET or COMPILE commands, or the OPTION statement.

To simplify the representation of certain ASCII characters and mathematical values, VAX BASIC also supplies some predefined constants.

The following sections discuss numeric and string constants, named constants, numeric literal notation, and predefined constants.

---

## 1.5.1 Numeric Constants

A numeric constant is a literal or named constant whose value never changes. In VAX BASIC, a numeric constant can be a floating-point number, an integer, or a packed decimal number. The type and size of a numeric constant is determined by

- The system default values
- The defaults set by the qualifiers for the DCL command BASIC
- The data type qualifiers specified with the COMPILE command
- The defaults set by the SET command
- The data type specified in a DECLARE CONSTANT or OPTION statement
- Numeric literal notation

If you use a declarative statement to name and declare the data type of a numeric constant, the constant is of the type and size specified in the statement. For example:

```
DECLARE BYTE CONSTANT age = 12
```

This example associates the numeric literal 12 and the BYTE data type with the identifier *age*. To specify a data type for an unnamed numeric constant, you must use the numeric literal notation format described in Section 1.5.4.

---

### 1.5.1.1 Floating-Point Constants

A floating-point constant is a literal or named constant with one or more decimal digits, either positive or negative, with an optional decimal point and an optional exponent (E notation). If the default data type is integer, VAX BASIC will treat the literal as an INTEGER unless it contains a decimal point or the character E. If the default data type is DECIMAL, an E is required or VAX BASIC treats the literal as a packed decimal value.

Table 1–4 contains examples of floating-point literals with REAL, INTEGER, and DECIMAL default data types.

**Table 1–4: Specifying Floating-Point Constants**

<b>REAL</b>	<b>INTEGER</b>	<b>DECIMAL</b>
–8.738	–8.738	–8.738
239.21E-6	239.21E-6	239.21E-6
.79	.79	.79E
299	299E	299E

Very large and very small numbers can be represented in E (exponential) notation. If a positive number appears in E notation, it can be preceded by an optional plus sign (+). A negative number in E notation must be preceded by a minus sign (–). A number can be carried to a maximum of 6 decimal places for SINGLE precision, 16 decimal places for DOUBLE precision, 15 decimal places for GFLOAT precision, and 33 places for HFLOAT precision.

To indicate E notation, a number must be followed by the letter E. It also must be followed by an exponent sign and an exponent. The exponent sign indicates if the exponent is either positive or negative and is optional only if you are specifying a positive exponent. The exponent is an integer constant (the power of 10).

Table 1–5 compares numbers in standard and E notation.

**Table 1–5: Numbers in E Notation**

<b>Standard Notation</b>	<b>E Notation</b>
.0000001	.1E-06
1,000,000	.1E+07
–10,000,000	–.1E+08
100,000,000	.1E+09
1,000,000,000,000	.1E+13

The range and precision of floating-point constants are determined by the current default data types or the explicit data type used in the DECLARE CONSTANT statement. However, there are limits to the range allowed for numeric data types. Table 1–2 lists VAX BASIC data types and ranges. VAX BASIC signals the fatal error “Floating point error or overflow” (ERR=48) when your program attempts to specify a constant value outside of the allowable range for a floating-point data type.

---

### 1.5.1.2 Integer Constants

An integer constant is a literal or named constant, either positive or negative, with no fractional digits and an optional trailing percent sign (%). The percent sign is required for integer literals only if the default type is not `INTEGER`.

In Table 1–6, the values are all integer constants. The presence of the percent sign varies depending on the default data type.

**Table 1–6: Specifying Integer Constants**

<b>INTEGER</b>	<b>REAL or DECIMAL</b>
81257	81257%
-3477	-3477%
79	79%

The range of allowable values for integer constants is determined by either the current default data type or the explicit data type used in the `DECLARE CONSTANT` statement. Table 1–2 lists VAX BASIC data types and ranges. VAX BASIC signals an error for a number outside the applicable range.

If you want VAX BASIC to treat numeric literals as integer numbers, you must do one of the following:

- Set the default data type to `INTEGER`
- Make sure the literal has a percent sign suffix
- Use explicit literal notation

The VAX BASIC compiler must convert numeric literals when assigning them to integer variables. This means that your program runs somewhat slower than it would if integer values were explicitly declared. You can prevent this conversion step by using one of the following:

- Percent signs for integer constants
- Numeric literal notation
- Named integer constants

#### **NOTE**

You cannot use percent signs in integer constants that appear in `DATA` statements. An attempt to do so causes VAX BASIC to signal “Data format error” (`ERR=50`).

---

### 1.5.1.3 Packed Decimal Constants

A packed decimal constant is a number, either positive or negative, that has a specified number of digits and a specified decimal point position (scale). You specify the number of digits (*d*) and the position of the decimal point (*s*) when you declare the constant as a `DECIMAL(d,s)`. If the constant is not declared, the number of digits and the position of the decimal is determined by the representation of the constant.

For example, when the default data type is `DECIMAL`, `1.234` is a `DECIMAL(4,3)` constant, regardless of the default decimal size. Likewise, using numeric literal notation, `"1.234" P` is a `DECIMAL(4,3)` constant, regardless of the default data type and default `DECIMAL` size. Numeric literal notation is described in Section 1.5.4.

---

### 1.5.2 String Constants

String constants are either string literals or named constants. A string literal is a series of characters enclosed in string delimiters. Valid string delimiters are:

- Double quotation marks (`"text"`)
- Single quotation marks (`'text'`)

You can embed double quotation marks within single quotation marks (`'this is a "text" string'`) and vice versa (`"this is a 'text' string"`). Note, however, that VAX BASIC does not accept incorrectly paired quotation marks and that only the outer quotation marks must be paired. For example, the following character strings are valid:

```
"The record number does not exist."  
"I'm here!"  
"The terminating 'condition' is equal to A$."  
"REPORT 543"
```

However, the following strings are not valid:

```
"Quotation marks that do not match'  
"No closing quotation mark
```

Characters in string constants can be letters, numbers, spaces, tabs, 8-bit data characters, or the NUL character (ASCII code 0). If you need a string constant that contains a NUL, you should use the NUL predefined constant. See Section 1.5.4 in this manual for information on explicit literal notation.

The VAX BASIC compiler determines the value of the string constant by scanning all its characters. For example, because of the number of spaces between the delimiters and the characters, these two string constants are not the same:

```
"    END-OF-FILE REACHED    "  
"END-OF-FILE REACHED"
```

VAX BASIC stores every character between delimiters exactly as you type it into the source program, including:

- Lowercase letters (a–z)
- Leading, trailing, and embedded spaces
- Tabs
- Special characters

The delimiting quotation marks are not printed when the program is executing. The value of the string constant does not include the delimiting quotation marks. For example:

```
PRINT "END-OF-FILE REACHED"  
:  
:  
:  
END
```

### **Output**

```
END-OF-FILE REACHED
```

VAX BASIC does, however, print double or single quotation marks when they are enclosed in a second paired set. For example:

```
PRINT 'FAILURE CONDITION: "RECORD LENGTH"  
:  
:  
:  
END
```

### **Output**

```
FAILURE CONDITION: "RECORD LENGTH"
```



---

## 1.5.3 Named Constants

VAX BASIC allows you to name constants. You can assign a name to a constant that is either internal or external to your program and refer to the constant by name throughout the program. This naming feature is useful for the following reasons:

- If a commonly used constant must be changed, you need to make only one change in your program.
- A logically named constant makes your program easier to understand.

You can use named constants anywhere you can use a constant, for example, to specify the number of elements in an array.

You cannot change the value of an explicitly named constant during program execution. To change the value of a constant, you must change the program statement that names the constant and declares its value, and then recompile the program.

---

### 1.5.3.1 Naming Constants Within a Program Unit

You name constants within a program unit with the DECLARE statement, as is shown in Example 1-3.

#### Example 1-3: Naming Constants Within a Program Unit

---

```
DECLARE DOUBLE CONSTANT preferred_rate = .147
DECLARE SINGLE CONSTANT normal_rate = .162
DECLARE DOUBLE CONSTANT risky_rate = .175
.
.
new_bal = old_bal * (1 + preferred_rate)^years_payment
```

---

When interest rates change, only three lines have to be changed rather than every line that contains an interest rate constant.

Constant names must conform to the rules for naming internal, explicitly declared variables listed in Section 1.4.1. Note that constant names cannot have embedded spaces.

The value associated with a named constant can be a compile-time expression as well as a literal value, as shown in Example 1-4.

### Example 1-4: Associating Values with Named Constants

---

```
DECLARE STRING CONSTANT Congrats =          &
      "+-----+" + LF + CR + &
      "| Congratulations! |" + CR + CR + &
      "+-----+"
      .
      .
      .
PRINT Congrats
      .
      .
      .
PRINT Congrats
```

---

Named constants can save you programming time because you do not have to retype the value every time you want to display it. Named constants can save you execution time because the named constant is known at compilation time.

Valid operators in `DECLARE CONSTANT` expressions include all valid arithmetic, relational, and logical operators except exponentiation. You cannot use built-in functions in `DECLARE CONSTANT` expressions.

VAX BASIC allows constants of all data types except RFA to be named as expressions. Because you cannot declare a constant of the RFA data type you cannot name one as an expression. Example 1-5 illustrates the concept of naming constants as expressions:

### Example 1-5: Naming Constants as Expressions

---

```
DECLARE DOUBLE CONSTANT  &
      min_value = 0,      &
      max_value = PI/2
```

---

You can specify only one data type in a `DECLARE CONSTANT` statement. To declare a constant of a different data type, you must use a second `DECLARE CONSTANT` statement.

---

### 1.5.3.2 Naming Constants External to a Program Unit

To declare constants outside the program unit, use the `EXTERNAL` statement, as shown in Example 1–6.

#### Example 1–6: Declaring Constants Outside the Program Unit

---

```
EXTERNAL LONG CONSTANT SS$_NORMAL
EXTERNAL WORD CONSTANT IS_SUCCESS
```

---

The first line declares the VMS status code `SS$_NORMAL` to be an external `LONG` constant. The second line declares `IS_SUCCESS`, a success code, to be an external `WORD` constant. Note that VAX BASIC allows only external `BYTE`, `WORD`, `LONG`, and `SINGLE` constants. The VMS Linker supplies the values for the constants specified in `EXTERNAL` statements.

External constant names cannot exceed 31 characters and must conform to the rules for naming external variables listed in Section 1.4.1. No external constant name can have embedded spaces. In VAX BASIC, the named constant might be a system status code or a global constant declared in a VAX MACRO or VAX BLISS program.

---

### 1.5.4 Explicit Literal Notation

You can specify the value and data type of numeric literals by using a special notation called explicit literal notation. The format of this notation is as follows:

[radix] "num-str-lit" [data-type]

*Radix* specifies an optional base, which can be any of the following:

- D    Decimal (base 10)
- B    Binary (base 2)
- O    Octal (base 8)
- X    Hexadecimal (base 16)
- A    ASCII

The VAX BASIC default radix is decimal. Binary, octal, and hexadecimal notation allow you to set or clear individual bits in the representation of an integer. This feature is useful in forming conditional expressions and in using logical operations. The ASCII radix causes VAX BASIC to translate a single ASCII character to its decimal equivalent. This decimal equivalent is

an INTEGER value; you specify whether the INTEGER subtype should be BYTE, WORD, or LONG.

*Num-str-lit* is a numeric string literal. It can be the digits 0 and 1 when the radix is binary, the digits 0 through 7 when the radix is octal, the digits 0 through F when the radix is hexadecimal, and the digits 0 through 9 when the radix is decimal. When the radix is ASCII, *num-str-lit* can be any valid ASCII character.

*Data-type* is an optional single letter that corresponds to a data type keyword, excluding INTEGER and REAL:

B	BYTE
W	WORD
L	LONG
F	SINGLE
D	DOUBLE
G	GFLOAT
H	HFLOAT
P	DECIMAL
C	CHARACTER

Note that *data-type* for the ASCII radix is limited to BYTE, WORD, or LONG. For example:

D"255"L	Specifies a LONG decimal constant with a value of 255
"4000"F	Specifies a SINGLE decimal constant with a value of 4000
-"125"B	Specifies a BYTE decimal constant with a value of -125
A"M"L	Specifies a LONG integer constant with a value of 77
A"m"B	Specifies a BYTE integer constant with a value of 109

A quoted numeric string alone, without a radix and a data type, is a string literal, not a numeric literal. For example:

"255"W	Specifies a WORD decimal constant with a value of 255
"255"	Is a string literal

If you specify a binary, octal, or hexadecimal radix, *data-type* must be an integer. If you do not specify a data type, VAX BASIC uses the default integer data type. For example:

B"11111111"B	Specifies a BYTE binary constant with a value of -1
B"11111111"W	Specifies a WORD binary constant with a value of 255

B"11111111 "	Specifies a binary constant of the default data type (BYTE, WORD, or LONG)
B"11111111 "F	Is illegal because F is not an integer data type
X"FF"B	Specifies a BYTE hexadecimal constant with a value of -1
X"FF"W	Specifies a WORD hexadecimal constant with a value of 255
X"FF"D	Is illegal because D is not an integer data type
O"377"B	Specifies a BYTE octal constant with a value of -1
O"377"W	Specifies a WORD octal constant with a value of 255
O"377"G	Is illegal because G is not an integer data type

When you specify a radix other than decimal, overflow checking is performed as if the numeric string were an unsigned integer. However, when this value is assigned to a variable or used in an expression, the VAX BASIC compiler treats it as a signed integer.

In the following example, VAX BASIC sets all 8 bits in storage location A. Because A is a BYTE integer, it has only 8 bits of storage. Because the 8-bit two's complement of 1 is 11111111, its value is -1. If the data type were W (WORD), VAX BASIC would set the bits to 000000011111111, and its value would be 255.

```
DECLARE BYTE A
A = B"11111111"B
PRINT A
```

## Output

-1

## NOTE

In VAX BASIC, a D can appear in both the radix position and the data type position. D in the radix position specifies that the numeric string is to be treated as a decimal number (base 10). D in the data type position specifies that the value is to be treated as a double-precision, floating-point constant. P in the data type position specifies a packed decimal constant. For example:

"255"D	Specifies a double-precision constant with a value of 255
"255.55"P	Specifies a DECIMAL constant with a value of 255.55

You can use explicit literal notation to represent a single-character string in terms of its 8-bit ASCII value. For example:

```
[radix] num-str-lit C
```

The letter C is an abbreviation for CHARACTER. The value of the numeric string must be from 0 through 255. This feature lets you create your own compile-time string constants containing nonprinting characters.

The following example declares a string constant named *control\_g* (ASCII decimal value 7). When VAX BASIC executes the PRINT statement, the terminal bell sounds.

```
DECLARE STRING CONSTANT control_g = "7"C
PRINT control_g
```

---

## 1.5.5 Predefined Constants

Predefined constants are symbolic representations of either ASCII characters or mathematical values. They are also called compile-time constants because their value is known at compilation rather than at run time.

Predefined constants help you

- Format program output to improve readability
- Make source code easier to understand

Table 1–7 lists the predefined constants supplied by VAX BASIC, their ASCII values, and their functions.

**Table 1–7: Predefined Constants**

<b>Constant</b>	<b>Decimal ASCII Value</b>	<b>Function</b>
BEL (Bell)	7	Sounds the terminal bell
BS (Backspace)	8	Moves the cursor one position to the left
HT (Horizontal Tab)	9	Moves the cursor to the next horizontal tab stop
LF (Line Feed)	10	Moves the cursor to the next line
VT (Vertical Tab)	11	Moves the cursor to the next vertical tab stop
FF (Form Feed)	12	Moves the cursor to the start of the next page

(continued on next page)

**Table 1–7 (Cont.): Predefined Constants**

<b>Constant</b>	<b>Decimal ASCII Value</b>	<b>Function</b>
CR (Carriage Return)	13	Moves the cursor to the beginning of the current line
SO (Shift Out)	14	Shifts out for communications networking, screen formatting, and alternate graphics
SI (Shift In)	15	Shifts in for communications networking, screen formatting, and alternate graphics
ESC (Escape)	27	Marks the beginning of an escape sequence
SP (Space)	32	Inserts one blank space in program output
DEL (Delete)	127	Deletes the last character entered
PI	None	Represents the number PI with the precision of the default floating-point data type

You can use predefined constants in many ways. For instance, the following example shows how to print and underline a word on a hard copy terminal.

```
PRINT "NAME:" + BS + BS + BS + BS + BS + " _____"  
END
```

### **Output**

NAME:

The following example shows how to print and underline a word on a VT100 video display terminal.

```
PRINT ESC + "[4mNAME:" + ESC + "[0m"  
END
```

### **Output**

NAME:

Note that the “m” in the above example must be lowercase.

You can also create your own predefined constants with the DECLARE CONSTANT statement.

In the following example, the first DECLARE statement defines *underlined\_name* as a string constant. The second DECLARE statement defines *D\_PI* as a DOUBLE constant equal to the predefined constant *PI*. If the default REAL data size is SINGLE, the program can use both single-precision *PI* and double-precision *D\_PI*.

```
DECLARE STRING CONSTANT underlined_name = ESC + "[4mNAME:" + ESC + "[0m"  
DECLARE DOUBLE CONSTANT D_PI = PI  
PRINT underlined_name  
PRINT D_PI,,PI
```

---

## 1.6 Expressions

VAX BASIC expressions consist of operands (numbers, strings, constants, variables, functions, and array elements) separated by arithmetic, string, relational, and logical operators.

Almost all VAX BASIC expressions yield numeric values. The only exceptions are string concatenation expressions and invocations of string-valued functions. By using different combinations of numeric operators and operands, and by using the resulting values, you can produce

- Numeric expressions
- String expressions
- Conditional expressions

VAX BASIC evaluates expressions according to operator precedence and uses the results in program execution. Parentheses can be used to group operands and operators, thus controlling the order of evaluation.

The following sections explain the types of expressions you can create and the way VAX BASIC evaluates expressions.

---

### 1.6.1 Numeric Expressions

Numeric expressions consist of floating-point, integer, or packed decimal operands separated by arithmetic operators and optionally grouped by parentheses. Table 1–8 shows how numeric operators work in numeric expressions.



**Table 1–8: Arithmetic Operators**

<b>Operator</b>	<b>Example</b>	<b>Use</b>
+	A + B	Add B to A
–	A – B	Subtract B from A
*	A * B	Multiply A by B
/	A / B	Divide A by B
^	A^B	Raise A to the power B
**	A**B	Raise A to the power B

In general, two arithmetic operators cannot occur consecutively in the same expression. Exceptions are the unary plus and unary minus. The following expressions are valid.

```
A * + B
A * - B
A * (-B)
A * + - + - B
```

The following expression is not valid:

```
A - * B
```

An operation on two numeric operands of the same data type yields a result of that type. For example:

```
A% + B%      Yields an integer value of the default type
G3 * M5      Yields a floating-point value if the default type is REAL
```

If the result of the operation exceeds the range of the data type, VAX BASIC signals an overflow error message.

The following example causes VAX BASIC to signal the error “Integer error or overflow” because the sum of *A* and *B* (254) exceeds the range of –128 to +127 for BYTE integers. Similar overflow errors occur for REAL and DECIMAL data types whenever the result of a numeric operation is outside the range of the corresponding data type.

```
DECLARE BYTE A, B
A = 127
B = 127
PRINT A + B
END
```

It is possible to assign a value of one data type to a variable of a different data type. When this occurs, the data type of the variable overrides the data type of the assigned value. The following example assigns the value 32 to the integer variable `A%` even though the floating-point value of the expression is 32.13.

```
A% = 5.1 * 6.3
```

---

### 1.6.1.1 Floating-Point and Integer Promotion Rules

When an expression contains operands with different data types, the data type of the result is determined by VAX BASIC's data type promotion rules:

- With one exception, VAX BASIC promotes operands with different data types to the lowest common data type that can hold the largest or most precise possible value of either operand's data type. VAX BASIC then performs the operation using that data type, and yields a result of that data type.
- The exception is that when an operation involves SINGLE and LONG data types, VAX BASIC promotes the LONG data type to SINGLE rather than DOUBLE, performs the operation, and yields a result of the SINGLE data type.

Note that VAX BASIC does sign extension when converting BYTE and WORD integers to a higher INTEGER data type (WORD or LONG). The high order bit (the sign bit) determines how the additional bits are set when the BYTE or WORD is converted to WORD or LONG. If the high order bit is zero (positive), all higher-order bits in the converted BYTE or WORD are set to zero. If the high order bit is 1 (negative), all higher-order bits in the converted BYTE or WORD are set to 1.

Table 1–9 lists the data type results possible in numeric expressions that combine BYTE, WORD, LONG, SINGLE, and DOUBLE data. Table 1–10 lists the data type results possible in numeric expressions that combine the data types GFLOAT and HFLOAT. When the operands are DOUBLE and GFLOAT, VAX BASIC promotes both values to HFLOAT, and returns an HFLOAT value. The promotion of DOUBLE and GFLOAT to HFLOAT is necessary because a DOUBLE value is more precise than a GFLOAT value, but cannot contain the largest possible GFLOAT value. Consequently, VAX BASIC promotes these data types to a data type that can hold the largest and most precise value of either operand.

**Table 1–9: Result Data Types in VAX BASIC Expressions**

<b>BYTE</b>	<b>WORD</b>	<b>LONG</b>	<b>SINGLE</b>	<b>DOUBLE</b>	
<b>BYTE</b>	BYTE	WORD	LONG	SINGLE	DOUBLE
<b>WORD</b>	WORD	WORD	LONG	SINGLE	DOUBLE
<b>LONG</b>	LONG	LONG	LONG	SINGLE	DOUBLE
<b>SINGLE</b>	SINGLE	SINGLE	SINGLE	SINGLE	DOUBLE
<b>DOUBLE</b>	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE

For example, if one operand is **SINGLE** and one operand is **DOUBLE**, VAX BASIC promotes the **SINGLE** value to **DOUBLE**, performs the specified operation, and returns the result as a **DOUBLE** value. This promotion is necessary because the **SINGLE** data type has less precision than the **DOUBLE** value, whereas the **DOUBLE** data type can represent all possible **SINGLE** values. If VAX BASIC did not promote the **SINGLE** value and the operation yielded a result outside of the **SINGLE** range, loss of precision and significance would occur.

The data types **BYTE**, **WORD**, **LONG**, **SINGLE**, and **DOUBLE** form a simple hierarchy: if all operands in an expression are of these data types, the result of the expression is the highest data type used in the expression.

**Table 1–10: VAX BASIC Result Data Types**

	<b>GFLOAT</b>	<b>HFLOAT</b>
<b>BYTE</b>	GFLOAT	HFLOAT
<b>WORD</b>	GFLOAT	HFLOAT
<b>LONG</b>	GFLOAT	HFLOAT
<b>SINGLE</b>	GFLOAT	HFLOAT
<b>DOUBLE</b>	HFLOAT	HFLOAT
<b>GFLOAT</b>	GFLOAT	HFLOAT
<b>HFLOAT</b>	HFLOAT	HFLOAT

### 1.6.1.2 DECIMAL Promotion Rules

VAX BASIC allows the **DECIMAL(d,s)** data type. The number of digits (*d*) and the scale or position of the decimal point (*s*) in the result of **DECIMAL** operations depends on the data type of the other operand. If one operand is **DECIMAL** and the other is **DECIMAL** or **INTEGER**, the *d* and *s* values of the result are determined as follows.

- If both operands are typed **DECIMAL**, and if both operands have the same digit (d) and scale (s) values, no conversions occur and the result of the operation has exactly the same d and s values as the operands. Note, however, that overflow can occur if the result exceeds the range specified by the d value.
- If both operands are **DECIMAL** but have different digit and scale values, **VAX BASIC** uses the larger number of specified digits for the result.

In the following example, variable *A* allows three digits to the left of the decimal point and two digits to the right. Variable *B* allows one digit to the left of the decimal point and three digits to the right.

```
DECLARE DECIMAL(5,2) A
DECLARE DECIMAL(4,3) B
```

The result allows three digits to the left of the decimal point and three digits to the right.

- If one operand is **DECIMAL** and one is **INTEGER**, the **INTEGER** value is converted to a **DECIMAL(d,s)** data type as follows:
  - **BYTE** is converted to **DECIMAL(3,0)**.
  - **WORD** is converted to **DECIMAL(5,0)**.
  - **LONG** is converted to **DECIMAL(10,0)**.

**VAX BASIC** then determines the d and s values of the result by evaluating the d and s values of the operands as described above.

Note that only **INTEGER** data types are converted to the **DECIMAL** data type. If one operand is **DECIMAL** and one is floating-point, the **DECIMAL** value is converted to a floating-point value. The total number of digits (d) in the **DECIMAL** value determines its new data type, as shown in Table 1–11.

**Table 1–11: Result Data Types for DECIMAL Data**

Number of <b>DECIMAL</b> Digits in Operand	Floating-Point Operands			
	<b>SINGLE</b>	<b>DOUBLE</b>	<b>GFLOAT</b>	<b>HFLOAT</b>
1–6	<b>SINGLE</b>	<b>DOUBLE</b>	<b>GFLOAT</b>	<b>HFLOAT</b>
7–15	<b>DOUBLE</b>	<b>DOUBLE</b>	<b>GFLOAT</b>	<b>HFLOAT</b>

(continued on next page)

**Table 1–11 (Cont.): Result Data Types for DECIMAL Data**

<b>Number of DECIMAL Digits in Operand</b>	<b>Floating-Point Operands</b>			
	<b>SINGLE</b>	<b>DOUBLE</b>	<b>GFLOAT</b>	<b>HFLOAT</b>
16	DOUBLE	DOUBLE	HFLOAT	HFLOAT
17–31	HFLOAT	HFLOAT	HFLOAT	HFLOAT

If the value of *d* is between 7 and 15, the operand is converted to:

- DOUBLE if the floating-point operand is SINGLE or DOUBLE
- GFLOAT if the floating-point operand is GFLOAT
- HFLOAT if the floating-point operand is HFLOAT

Thus, a DECIMAL(8,5) operand is converted to DOUBLE if the other operand is SINGLE or DOUBLE, to GFLOAT if the other operand is GFLOAT, and to HFLOAT if the other operand is HFLOAT. Note also that exponentiation of a DECIMAL data type returns a REAL value.

See the *VAX BASIC User Manual* for tutorial information on data type interactions, conversions, and promotion rules in VAX BASIC numeric expressions.

---

## 1.6.2 String Expressions

String expressions are string entities separated by the plus sign (+). When used in a string expression, the plus sign concatenates strings. For example:

```
INPUT "Type two words to be combined";A$, B$
C$ = A$ + B$
PRINT C$
END
```

### Output

```
Type two words to be combined? long
? word

longword

Ready
```

---

## 1.6.3 Conditional Expressions

Conditional expressions can be either relational or logical expressions. Numeric relational expressions compare numeric operands to determine whether the expression is true or false. String relational expressions compare string operands to determine which string expression occurs first in the ASCII collating sequence.

Logical expressions contain integer operands and logical operators. VAX BASIC determines whether the specified logical expression is true or false by testing the numeric result of the expression. Note that in conditional expressions, as in any numeric expression, when BYTE and WORD operands are converted to WORD and LONG, the specified operation is performed in the higher data type, and the result returned is also of the higher data type. When one of the operands is a negative value, this conversion will produce accurate but perhaps confusing results, because VAX BASIC performs a sign extension when converting BYTE and WORD integers to a higher integer data type. See Section 1.6.1.1 for information on integer conversion rules.

---

### 1.6.3.1 Numeric Relational Expressions

Operators in numeric relational expressions compare the values of two operands and returns either a -1 if the relation is true (as shown in Example 1), or a zero if the relation is false (as shown in Example 2). The data type of the result is the default integer type.

#### Example 1

```
A = 10
B = 15
X% = (A <> B)
IF X% = -1%
THEN PRINT 'Relationship is true'
ELSE IF X% = 0
    THEN PRINT 'Relationship is false'
    END IF
END IF
```

#### Output

```
Relationship is true
```

## Example 2

```
A = 10
B = 15
X% = A = B
IF X% = -1%
THEN PRINT 'Relationship is true'
ELSE IF X% = 0
    THEN PRINT 'Relationship is false'
    END IF
END IF
```

## Output

Relationship is false

Table 1–12 shows how numeric operators work in numeric relational expressions.

**Table 1–12: Numeric Relational Operators**

Operator	Example	Meaning
=	A = B	A is equal to B.
<	A < B	A is less than B.
>	A > B	A is greater than B.
<= or <=	A <= B	A is less than or equal to B.
>= or >=	A >= B	A is greater than or equal to B.
<> or <>	A <> B	A is not equal to B.
==	A == B	A and B will PRINT the same if they are equal to six significant digits. However, if one value prints in explicit notation and the other value prints in E format notation, the relation will always be false.

### 1.6.3.2 String Relational Expressions

Operators in string relational expressions determine how VAX BASIC compares strings. The VAX BASIC compiler determines the value of each character in the string by converting it to its ASCII value. ASCII values are listed in Appendix C in this manual. VAX BASIC compares the strings character by character, left to right, until it finds a difference in ASCII value.

In the following example, VAX BASIC compares *A\$* and *B\$* character by character. The strings are identical up to the third character. Because the ASCII value of *Z* (90) is greater than the ASCII value of *C* (67), *A\$* is less

than  $B\$$ . VAX BASIC evaluates the expression  $A\$ < B\$$  as true (-1) and prints "ABC comes before ABZ".

```
A$ = 'ABC'
B$ = 'ABZ'
IF A$ < B$
THEN PRINT 'ABC comes before ABZ'
ELSE IF A$ == B$
THEN PRINT 'The strings are identical'
ELSE IF A$ > B$
THEN PRINT 'ABC comes after ABZ'
ELSE PRINT 'Strings are equal but not identical'
END IF
END IF
END
```

If two strings of differing lengths are identical up to the last character in the shorter string, VAX BASIC pads the shorter string with spaces (ASCII value 32) to generate strings of equal length, unless the operator is the double equal sign (==). If the operator is the double equal sign, VAX BASIC does not pad the shorter string.

In the following example, VAX BASIC compares "ABCDE" to "ABC " to determine which string comes first in the collating sequence. "ABC " comes before "ABCDE" because the ASCII value for space (32) is lower than the ASCII value of *D* (68). Then VAX BASIC compares "ABC " with "ABC" using the double equal sign and determines that the strings do not match exactly without padding. The third comparison uses the single equal sign. VAX BASIC pads "ABC" with spaces and determines that the two strings match with padding.

```
A$ = 'ABCDE'
B$ = 'ABC'
PRINT 'B$ comes before A$' IF B$ < A$
PRINT 'A$ comes before B$' IF A$ < B$
C$ = 'ABC '
IF B$ == C$
THEN PRINT 'B$ exactly matches C$'
ELSE PRINT 'B$ does not exactly match C$'
END IF
IF B$ = C$
THEN PRINT 'B$ matches C$ with padding'
ELSE PRINT 'B$ does not match C$'
END IF
```

## Output

```
B$ comes before A$
B$ does not exactly match C$
B$ matches C$ with padding
```



Table 1–13 shows how numeric operators work in string relational expressions.

**Table 1–13: String Relational Operators**

<b>Operator</b>	<b>Example</b>	<b>Meaning</b>
=	A\$ = B\$	Strings A\$ and B\$ are identical after the shorter string has been padded with spaces to equal the length of the longer string.
<	A\$ < B\$	String A\$ occurs before string B\$ in ASCII sequence.
>	A\$ > B\$	String A\$ occurs after string B\$ in ASCII sequence.
<= or =<	A\$ <= B\$	String A\$ is identical to or precedes string B\$ in ASCII sequence.
>= or =>	A\$ >= B\$	String A\$ is identical to or follows string B\$ in ASCII sequence.
<> or ><	A\$ <> B\$	String A\$ is not identical to string B\$.
==	A\$ == B\$	Strings A\$ and B\$ are identical in composition and length, without padding.

VAX BASIC treats unquoted strings typed in response to the INPUT statement differently from quoted strings; it does so by ignoring leading and trailing spaces and tabs. For example, it evaluates the quoted strings “ABC” and “ABC ” as equal but not identical because the == operator does not pad the shorter string with spaces. When you input those same strings as unquoted strings in response to the INPUT prompt, VAX BASIC evaluates them as equal and identical because it ignores the trailing spaces. The LINPUT statement, on the other hand, treats unquoted strings as string literals, so the trailing spaces are part of the string, and VAX BASIC evaluates the strings as equal, but not identical.

### 1.6.3.3 Logical Expressions

A logical expression can have one of the following formats:

- A unary logical operator and one integer operand
- Two integer operands separated by a binary logical operator
- One integer operand

Logical expressions are valid only when the operands are integers. If the expression contains two integer operands of differing data types, the resulting integer has the same data type as the higher integer operand. For instance, the result of an expression that contains a BYTE integer and a WORD integer would be a WORD integer. Table 1–9 shows how integer data types interact with each other in expressions.

VAX BASIC determines whether the condition is true or false by testing the result of the logical expression to see whether any bits are set. If no bits are set, the value of the expression is zero and it is evaluated as false; if any bits are set, the value of the expression is nonzero, and the expression is evaluated as true. VAX BASIC generally accepts any nonzero value in logical expressions as true. However, logical operators can return unanticipated results unless  $-1$  is specified for true values and zero for false. Table 1–14 lists the logical operators.

#### NOTE

DIGITAL recommends that you use logical operators on the results of relational expressions to avoid obtaining unanticipated results.

**Table 1–14: Logical Operators**

<b>Operator</b>	<b>Example</b>	<b>Meaning</b>
NOT	NOT A%	The bit-by-bit complement of A%. If A% is true ( $-1$ ), NOT A% is false ( $0$ ).
AND	A% AND B%	The logical product of A% and B%. A% AND B% is true only if both A% and B% are true.
OR	A% OR B%	The logical sum of A% and B%. A% OR B% is false only if both A% and B% are false; otherwise, A% OR B% is true.
XOR	A% XOR B%	The logical exclusive OR of A% and B%. A% XOR B% is true if either A% or B% is true but not if both are true.

(continued on next page)

**Table 1–14 (Cont.): Logical Operators**

<b>Operator</b>	<b>Example</b>	<b>Meaning</b>
EQV	A% EQV B%	The logical equivalence of A% and B%. A% EQV B% is true if A% and B% are both true or both false; otherwise the value is false.
IMP	A% IMP B%	The logical implication of A% and B%. A% IMP B% is false only if A% is true and B% is false; otherwise, the value is true.

The truth tables in Figure 1–2 summarize the results of these logical operations. Zero is false; –1 is true.

**Figure 1–2: Truth Tables**

---

<b>A%</b>	<b>NOT A%</b>	<b>A%</b>	<b>B%</b>	<b>A% OR B%</b>	
0	-1	0	0	0	
-1	0	0	-1	-1	
		-1	0	-1	
		-1	-1	-1	
<b>A%</b>	<b>B%</b>	<b>A% AND B%</b>	<b>A%</b>	<b>B%</b>	<b>A% EQV B%</b>
0	0	0	0	0	-1
0	-1	0	0	-1	0
-1	0	0	-1	0	0
-1	-1	-1	-1	-1	-1
<b>A%</b>	<b>B%</b>	<b>A% XOR B%</b>	<b>A%</b>	<b>B%</b>	<b>A% IMP B%</b>
0	0	0	0	0	-1
0	-1	-1	0	-1	-1
-1	0	-1	-1	0	0
-1	-1	0	-1	-1	-1

ZK-5548-GE

---

The operators XOR and EQV are logical complements.

In the following example, the values of *A%* and *B%* both test as true because they are nonzero values. However, the logical AND of these two variables returns an unanticipated result of false.

```
A% = 2%
B% = 4%
IF A% THEN PRINT 'A% IS TRUE'
IF B% THEN PRINT 'B% IS TRUE'
IF A% AND B% THEN PRINT 'A% AND B% IS TRUE'
ELSE PRINT 'A% AND B% IS FALSE'
END
```

## Output

```
A% IS TRUE
B% IS TRUE
A% AND B% IS FALSE
```

The program returns this seemingly contradictory result because logical operators work on the individual bits of the operands. The 8-bit binary representation of 2% is as follows:

```
0 0 0 0 0 0 1 0
```

The 8-bit binary representation of 4% is as follows:

```
0 0 0 0 0 1 0 0
```

Each value tests as true because it is nonzero. However, the AND operation on these two values sets a bit in the result only if the corresponding bit is set in both operands. Therefore, the result of the AND operation on 4% and 2% is as follows:

```
0 0 0 0 0 0 0 0
```

No bits are set in the result, so the value tests as false (zero).

If the value of *B%* is changed to 6%, the resulting value tests as true (nonzero) because both 6% and 2% have the second bit set. Therefore, VAX BASIC sets the second bit in the result and the value tests as nonzero and true.

The 8-bit binary representation of -1 is as follows:

```
1 1 1 1 1 1 1 1
```

The result of -1% AND -1% is -1% because VAX BASIC sets bits in the result for each corresponding bit that is set in the operands. The result tests as true because it is a nonzero value, as shown in the following example:

```
A% = -1%
B% = -1%
IF A% THEN PRINT 'A% IS TRUE'
IF B% THEN PRINT 'B% IS TRUE'
IF A% AND B% THEN PRINT 'A% AND B% IS TRUE'
                    ELSE PRINT 'A% AND B% IS FALSE'
END
```

## Output

```
A% IS TRUE
B% IS TRUE
A% AND B% IS TRUE
```

Your program may also return unanticipated results if you use the NOT operator with a nonzero operand that is not -1.

In the following example, VAX BASIC evaluates both *A%* and *B%* as true because they are nonzero. *NOT A%* is evaluated as false (zero) because the binary complement of  $-1$  is zero. *NOT B%* is evaluated as true because the binary complement of  $2$  has bits set and is therefore a nonzero value.

```
A%=-1%
B%=2
IF A% THEN PRINT 'A% IS TRUE'
      ELSE PRINT 'A% IS FALSE'
IF B% THEN PRINT 'B% IS TRUE'
      ELSE PRINT 'B% IS FALSE'
IF NOT A% THEN PRINT 'NOT A% IS TRUE'
      ELSE PRINT 'NOT A% IS FALSE'
IF NOT B% THEN PRINT 'NOT B% IS TRUE'
      ELSE PRINT 'NOT B% IS FALSE'
END
```

### Output

```
A% IS TRUE
B% IS TRUE
NOT A% IS FALSE
NOT B% IS TRUE
```

---

## 1.6.4 Evaluating Expressions

VAX BASIC evaluates expressions according to operator precedence. Each arithmetic, relational, and string operator in an expression has a position in the hierarchy of operators. The operator's position informs VAX BASIC of the order in which to perform the operation. Parentheses can change the order of precedence.

Table 1–15 lists all operators as VAX BASIC evaluates them. Note that

- Operators with equal precedence are evaluated logically from left to right.
- VAX BASIC evaluates expressions enclosed in parentheses first, even when the operator in parentheses has a lower precedence than that outside the parentheses.
- The addition (+) and multiplication (\*) operators are evaluated in algebraic order.

**Table 1–15: Numeric Operator Precedence**

<b>Operator</b>	<b>Precedence</b>
** or ^	1
– (unary minus) or + (unary plus)	2
* or /	3
+ or –	4
+ (concatenation)	5
all relational operators	6
NOT	7
AND	8
OR, XOR	9
IMP	10
EQV	11

For example, VAX BASIC evaluates the expression  $A = 15^2 + 12^2 - (35 * 8)$  in five steps:

1.  $15^2 = 225$                       Exponentiation (leftmost expression)
2.  $12^2 = 144$                       Exponentiation
3.  $225 + 144 = 369$                 Addition
4.  $(35 * 8) = 280$                 Multiplication
5.  $369 - 280 = 89$                 Subtraction

There is one exception to this order of precedence: when an operator that does not require operands on either side of it (such as NOT) immediately follows an operator that does require operands on both sides (such as the addition operator (+)), VAX BASIC evaluates the second operator first. For example:

$A\% + \text{NOT } B\% + C\%$

This expression is evaluated as:

$(A\% + (\text{NOT } B\%)) + C\%$

VAX BASIC evaluates the expression NOT B before it evaluates the expression A + NOT B. When the NOT expression does not follow the addition (+) expression, the normal order of precedence is followed. For example:

$\text{NOT } A\% + B\% + C\%$

This expression is evaluated as:

```
NOT ((A% + B%) + C %)
```

VAX BASIC evaluates the two expressions (A% + B%) and ((A% + B%) + C%) because the + operator has a higher precedence than the NOT operator.

VAX BASIC evaluates nested parenthetical expressions from the inside out.

In the following example, VAX BASIC evaluates the parenthetical expression *A* quite differently from expression *B*. For expression *A*, VAX BASIC evaluates the innermost parenthetical expression (25 + 5) first, then the second inner expression (30 / 5), then (6 \* 7), and finally (42 + 3). For expression *B*, VAX BASIC evaluates (5 / 5) first, then (1 \* 7), then (25 + 7 + 3) to obtain a different value.

```
A = (((25 + 5) / 5) * 7) + 3
PRINT A
B = 25 + 5 / 5 * 7 + 3
PRINT B
```

### Output

```
45
35
```

---

## 1.7 Program Documentation

Documentation within a program clarifies and explains source program structure. These explanations, or comments, can be combined with code to create a more readable program without affecting program execution. Comments can appear in two forms:

- Comment fields (including empty statements)
- REM statements

---

### 1.7.1 Comment Fields

A comment field begins with an exclamation point (!) and ends with a carriage return. You supply text after the exclamation point to document your program. You can specify comment fields while creating VAX BASIC programs at DCL level as well as in the BASIC environment. In both cases, VAX BASIC does not execute text in a comment field. Example 1-7 shows how to specify a comment field.



### Example 1–7: Specifying a Comment Field

---

```
! FOR loop to initialize list Q
FOR I = 1 TO 10
    Q(I) = 0 ! This is a comment
NEXT I
! List now initialized
```

---

VAX BASIC executes only the FOR...NEXT loop. The comment fields, preceded by exclamation points, are not executed.

Example 1–8 shows how you can use comment fields to help make your program more readable and allow you to format your program into readily visible logical blocks. Example 1–8 also shows how comment fields can be used as target lines for GOTO and GOSUB statements.

### Example 1–8: Using Comments Fields to Format a Program

---

```
!
! Square root program
!
INPUT 'Enter a number';A
PRINT 'SQR of ',A,' is ';SQR(A)
!
! More square roots?
!
INPUT 'Type "Y" to continue, press RETURN to quit';ANS$
GOTO 10 IF ANS$ = "Y"
!
END
```

---

You can also use an exclamation point to terminate a comment field, but this practice is not recommended. You should make sure that there are no exclamation points in the comment field itself; otherwise, VAX BASIC treats the text remaining on the line as source code.

#### NOTE

Comment fields in DATA statements are invalid; the VAX BASIC compiler treats the comments as additional data.

Empty statements consist of a line number and an exclamation point. Empty statements can make your program more legible by increasing the amount of “white space” and visually separating logical program segments. In Example 1–9, lines 100 and 300 are empty statements.

### Example 1–9: Using Empty Statements in a VAX BASIC Program

---

```
100  !  
    ! FOR loop to initialize list Q  
    !  
200  FOR I = 1 TO 10  
    Q(I) = 0 ! This is a comment  
    NEXT I  
300  !  
    ! List is now initialized
```

---

In general, empty statements can be used to make a program more legible and organized.

---

## 1.7.2 REM Statements

A REM statement begins with the REM keyword and ends when VAX BASIC encounters a new line number. The text you supply between the REM keyword and the next line number documents your program. Like comment fields, REM statements do not affect program execution. VAX BASIC ignores all characters between the keyword REM and the next line number. Therefore, the REM statement can be continued without the ampersand continuation character and should be the only statement on the line or the last of several statements in a multi-statement line. Example 1–10 shows the use of the REM statement.

## Example 1–10: Using REM Statements in VAX BASIC Programs

---

```
REM This is an example
A=5
B=10
REM A equals 5
    B equals 10
PRINT A, B
```

---

The REM statement is nonexecutable. When you transfer control to a REM statement, VAX BASIC executes the next executable statement that lexically follows the referenced statement.

### NOTE

Because VAX BASIC treats all text between the REM statement and the next line number as commentary, REM should be used very carefully in programs that follow the implied continuation rules. REM statements are disallowed in programs without line numbers.

In the following example, the conditional GOTO statement in line 20 transfers program control to line 10. VAX BASIC ignores the REM comment on line 10 and continues program execution at line 20.

```
10 REM ** Square root program
20 INPUT 'Enter a number';A
   PRINT 'SQR of ';A;' is ';SQR(A)
   INPUT 'Type "Y" to continue, press RETURN to quit';ANS$
   GOTO 10 IF ANS$ = "Y"
40 END
```



## Chapter 2

# Environment Commands

---

Environment commands are commands that you use in the BASIC environment. With environment commands, you can display, edit, and merge VAX BASIC programs, set compiler defaults, move VAX BASIC source programs to and from storage, and execute programs. This chapter lists alphabetically all of the compiler commands that can be used within the BASIC environment. For information on immediate mode and calculator mode statements, see the *VAX BASIC User Manual*.

**! your-comment**

---

## **! your-comment**

You can enter comments while in the BASIC environment by typing an exclamation point (!) and the comment.

---

### **Format**

**! *your-comment***

---

### **Syntax Rules**

1. The exclamation point must be the first character on the line.
2. You cannot continue a comment over more than one line.

---

### **Remarks**

None.

---

### **Examples**

#### **Example 1**

Ready

! Comments here ...

**! your-comment**

## **Example 2**

```
$ TYPE BUILD_SPECIAL.COM
$ SET VERIFY
$ BASIC
!+
! Set the compilation options by uncommenting
! the appropriate ones.
!-
! SET LIST
SET WORD
SET DEBUG
!+
! Get the source module.
!-
OLD SPECIAL
!+
! Compile it.
!-
COMPILE
!+
! All done.
!-
EXIT
```

## \$ system-command

---

## \$ system-command

You can execute a DCL command while in the BASIC environment by typing a dollar sign (\$) before the command. VAX BASIC passes the command to the operating system for execution. The context of the BASIC environment and the program currently in memory do not change.

---

### Format

\$ *system-command*

---

### Syntax Rules

VAX BASIC passes *system-command* directly to the VMS operating system without checking for validity.

---

### Remarks

1. The terminal displays any error messages or output that the command generates.
2. Control returns to the BASIC environment after the command executes. The context (source file status, loaded modules, and so on) of the BASIC environment and the program currently in memory do not change unless the command causes the operating system to abort VAX BASIC or log you out.
3. The command you specify executes within the context of a subprocess. Consequently, commands such as the DCL command SET execute only within the subprocess and do not affect the process running VAX BASIC.



---

## Example

Read :

```
$ SHOW PROTECTION
  SYSTEM=RWED, OWNER=RWED, GROUP=RWED, WORLD=RE
```

Ready

**b**

# APPEND

---

## APPEND

The APPEND command merges an existing VAX BASIC source program with the program currently in memory.

---

### Format

**APPEND** *[file-spec]*

---

### Syntax Rules

*File-spec* is the name of the VAX BASIC program you want to merge with the program currently in memory. The default file type is BAS.

---

### Remarks

1. You cannot specify the APPEND command on programs that do not contain line numbers.
2. If you type APPEND without specifying a file name, VAX BASIC prompts with

Append file name--

You should respond with a file name. If you respond with a carriage return and no file name, VAX BASIC searches for a file named NONAME.BAS. If the VAX BASIC compiler cannot find NONAME.BAS, VAX BASIC signals the error "Can't find file or account" (ERR=5).

3. You can append the contents of *file-spec* to a source program that is either called into memory with the OLD command or created in the BASIC environment. If there is no program in memory, VAX BASIC appends the file to an empty program with the default file name NONAME.

## APPEND

4. If *file-spec* contains a VAX BASIC line with the same line number as a line of the program in memory, the line in the appended file replaces the line of the program in memory. Otherwise, VAX BASIC inserts appended lines into the program in memory in sequential, ascending line number order.
5. The APPEND command does not change the name of the program in memory.
6. If you have not saved the appended version of the program, VAX BASIC signals the warning "Unsaved change has been made, CTRL/Z or EXIT to exit" the first time you try to leave the BASIC environment.

---

### Example

```
Ready
New FIRST_TRY.BAS
Ready
10 PRINT "First program"
APPEND NEW_PROG.BAS
Ready
LIST
10 PRINT "First Program"
20 PRINT "This section has been appended"
.
.
.
```

# ASSIGN

---

## ASSIGN

The ASSIGN command equates a logical name to a complete file specification, a device, or another logical name within the context of the BASIC environment.

---

### Format

**ASSIGN** *equiv-name[:]* *log-name[:]*

---

### Syntax Rules

1. *Equiv-name* specifies the file specification, device, or logical name to be assigned a logical name. If you specify a physical device name, you must terminate it with a colon (:).
2. *Log-name* is the 1- to 63-character logical name to be associated with *equiv-name*. You can specify a logical name for any portion of a file specification. If the logical name translates to a device name, and will be used in place of a device name in a file specification, you must terminate it with a colon (:).
3. If *log-name* has more than 63 characters, VAX BASIC signals the error "Invalid logical name".

---

### Remarks

1. When the logical name assignment supersedes another logical name previously assigned, VAX BASIC displays the message "Previous logical name assignment replaced".

# ASSIGN

2. Logical names assigned with the ASSIGN command are placed in the process logical name table and remain there until you exit the BASIC environment.

---

## Example

```
ASSIGN [HENRY.BAS] PRO:
```

# COMPILE

---

## COMPILE

The **COMPILE** command converts a VAX BASIC source program to an object module and writes the object file to disk.

---

### Format

**COMPILE** [ *file-spec* ] [ /*qualifier* ]...

#### Command Qualifiers

/[NO]ANSI\_STANDARD  
/[NO]AUDIT [ sep text-entry ]  
/[NO]BOUNDS\_CHECK  
/BYTE  
/[NO]CROSS\_REF [ sep [NO]KEYWORDS ]  
/[NO]DEBUG  
/DECIMAL\_SIZE sep (d,s)  
/DOUBLE  
/[NO]FLAG [ sep ( flag-clause,... ) ]  
/GFLOAT  
/HFLOAT  
/[NO]LINES  
/[NO]LIST  
/LONG  
/[NO]MACHINE\_CODE  
/[NO]OBJECT  
/[NO]OVERFLOW [ sep (data-type,...) ]  
/[NO]ROUND  
/[NO]SETUP  
/[NO]SHOW [ sep ( show-item,... ) ]  
/SINGLE  
/[NO]SYNTAX\_CHECK  
/[NO]TRACEBACK  
/TYPE\_DEFAULT sep default-clause

#### Defaults

/NOANSI\_STANDARD  
/NOAUDIT  
/BOUNDS\_CHECK  
/LONG  
/NOCROSS\_REF  
/NODEBUG  
/DECIMAL\_SIZE=(15,2)  
/SINGLE  
/NOFLAG  
/SINGLE  
/SINGLE  
/LINES  
/NOLIST  
/LONG  
/NOMACHINE  
/OBJECT  
/OVERFLOW=(INTEGER,DECIMAL)  
/NOROUND  
/SETUP  
/SHOW  
/SINGLE  
/NOSYNTAX\_CHECK  
/TRACEBACK  
/TYPE\_DEFAULT=REAL

/VARIANT sep int-const  
/[NO]WARNINGS [ sep warn-clause ]  
/WORD

/VARIANT=0  
/WARNINGS  
/LONG

---

## Syntax Rules

1. *File-spec* specifies a name for the output file or files. If you do not provide a *file-spec*, the VAX BASIC compiler uses the name of the program currently in memory for the file name, a default file type of OBJ for the object file, and a default file type of LIS for the listing file, if a listing file is requested.
2. *File-spec* can precede or follow any qualifier.
3. */Qualifier* specifies a qualifier keyword that sets a VAX BASIC default.
4. You can abbreviate all positive qualifiers to the first three letters of the qualifier keyword. You can abbreviate a negative qualifier to NO and the first three letters of the qualifier keyword.
5. In cases of ambiguous or erroneous qualifiers, VAX BASIC signals “Unknown qualifier”, and the program does not compile. When qualifiers conflict, VAX BASIC compiles the program using the last specified conflicting qualifier. For example, the following command line causes VAX BASIC to compile the program currently in memory but does not cause VAX BASIC to create an OBJ file.

COMPILE/OBJ/NOOBJ

6. There must be a program in memory, or the COMPILE command does not execute; VAX BASIC does not signal an error or warning.

---

## Remarks

1. The following qualifiers cannot be used within the VAX BASIC environment with the COMPILE command:
  - /ANALYSIS\_DATA
  - /CHECK

# COMPILE

- /DESIGN
  - /DEPENDENCY\_DATA
  - /DIAGNOSICS
  - /INTEGER\_SIZE
  - /OLD\_VERSION=CDD\_ARRAY
  - /REAL\_SIZE
  - /SCALE
2. If an object file for the program already exists in your directory, VAX BASIC creates a new version of the OBJ file.
  3. You should not specify both a file name and file type. For example, if you enter the following command line, VAX BASIC creates two versions of NEWOBJ.FIL:

```
COMPILE NEWOBJ.FIL/LIS/OBJ
```

The first version, NEWOBJ.FIL;1, is the listing file; the second version, NEWOBJ.FIL;2, is the object file. If you specify only a file name, VAX BASIC uses the OBJ and LIS file type defaults when creating these files.

4. Use the COMPILE/NOOBJECT command to check your program for errors without producing an object file.
5. When you exit from the BASIC environment, all options set with qualifiers return to the system default values. Use the SHOW command to display your system defaults before setting any qualifiers.

---

## Command Qualifiers

### ***/NOANSI\_STANDARD***

The /ANSI\_STANDARD qualifier causes VAX BASIC to compile programs according to the ANSI Minimal BASIC standard and to flag syntax that does not conform to the standard. The /NOANSI\_STANDARD qualifier causes VAX BASIC not to compile the program according to the ANSI Minimal BASIC standard. The default is /NOANSI\_STANDARD.

See the *VAX BASIC User Manual* for more information on the ANSI Minimal BASIC Standard.



**`/[NO]AUDIT [ { : } { str-lit file-spec } ]`**

The `/AUDIT` qualifier causes VAX BASIC to include a history list entry in the CDD data base when a CDD definition is extracted. *Str-lit* is a quoted string. *File-spec* is a text file. The history entry includes

- The contents of *str-lit*, or up to the first 64 lines in the file specified by *file-spec*
- The name of the program module, process, user name, and user UIC that accessed the CDD
- The time and date of the access
- A note that access was made by the VAX BASIC compiler
- A note that access was an extraction

If you specify `/NOAUDIT` VAX BASIC does not include a history list entry. `/NOAUDIT` is the default.

**`/[NO]BOUNDS_CHECK`**

The `/BOUNDS_CHECK` qualifier causes VAX BASIC to perform range checks on array subscripts. With bounds checking enabled, VAX BASIC checks that all subscript references are within the array boundaries set when the array was declared. If the subscript bounds are not within the bounds initially declared for the array, VAX BASIC signals an error message. If you specify `/NOBOUNDS_CHECK` VAX BASIC does not check that all subscript references are within the array bounds set. `/BOUNDS_CHECK` is the default.

**`/BYTE`**

The `/BYTE` qualifier causes VAX BASIC to allocate 8 bits of storage as the default for all integer data not explicitly typed in the program. Untyped integer values are treated as `BYTE` values and must be in the `BYTE` range or VAX BASIC signals the error “Integer error or overflow.” Table 1–2 in this manual lists VAX BASIC data types and ranges. By default, the VAX BASIC compiler allocates 32 bits of storage.

**`/[NO]CROSS_REFERENCE [ { : } [NO]KEYWORDS ]`**

If you use the `/CROSS_REFERENCE` qualifier with the `/LIST` qualifier when you compile your program, the VAX BASIC compiler includes cross-reference information in the program listing file. If

# COMPILE

you specify `/CROSS_REFERENCE=KEYWORDS`, VAX BASIC also cross-references VAX BASIC keywords used in the program. If you specify `/NOCROSS_REFERENCE`, VAX BASIC does not include a cross reference section in the compiler listing. The default is `/NOCROSS_REFERENCE`.

## **`/[NO]DEBUG`**

The `/DEBUG` qualifier appends to the object file information on symbolic references and line numbers. This information is used by the VMS Debugger when you debug your program. When you specify the `/DEBUG` qualifier on the `COMPILE` command, you cause the debugger to be invoked automatically when the program is run at DCL level (unless you specify `RUN/NODEBUG`). If you specify `COMPILE/NODEBUG`, information on program symbols and line numbers is not included in the object file. The default is `/NODEBUG`.

See the *VAX BASIC User Manual* for more information on using the VMS Debugger.

## **`/DECIMAL_SIZE { : } (d,s)`**

The `/DECIMAL_SIZE` qualifier allows you to specify the default size and precision for all `DECIMAL` data not explicitly assigned size and precision in the program. You specify the total number of digits (`d`) and the number of digits to the right of the decimal point (`s`). VAX BASIC signals the error “Decimal error or overflow” (`ERR=181`) when `DECIMAL` values are outside the range specified with this qualifier. See Table 1–2 in this manual for more information on the storage and range of packed decimal data. The default is `/DECIMAL_SIZE=(15,2)`.

## **`/DOUBLE`**

The `/DOUBLE` qualifier causes VAX BASIC to allocate 64 bits of storage in `D_floating` format as the default size for all floating-point data not explicitly typed in the program. Untyped floating-point values are treated as `DOUBLE` values and must be in the `DOUBLE` range or VAX BASIC signals the error “Floating-point error or overflow.” Table 1–2 in this manual lists VAX BASIC data types and ranges. The default is `/SINGLE`.

## **`/[NO]FLAG [ { : } ( { [NO]BP2COMPATIBILITY } ,... ) ]`**

The `/FLAG` qualifier causes VAX BASIC to provide compile-time information about program elements that are not compatible with `BASIC-PLUS-2` or that `DIGITAL` designates as not recommended for new program

development. For more information on source code that is incompatible with BASIC-PLUS-2, see Appendix A in this manual.

If you specify the DECLINING clause, VAX BASIC will flag the following source code as declining:

- CVT\$\$ (use EDIT\$)
- CVT\$%, CVT\$F, CVT%\$, CVTF\$, AND SWAP% (use multiple MAP statements)
- DEF\* functions (use DEF functions)
- FIELD statements (use MAP DYNAMIC and REMAP)
- GOTO line-num% (do not use the integer suffix with a line number)

The default is /NOFLAG.

## ***/GFLOAT***

The /GFLOAT qualifier causes VAX BASIC to allocate 64 bits of storage in G\_floating format as the default size for all floating-point data not explicitly typed in the program. Untyped floating-point values are treated as G\_floating values and must be in the G\_floating range or VAX BASIC signals "Floating-point error or overflow." Table 1-2 in this manual lists VAX BASIC data types and ranges. The default is /SINGLE.

## ***/HFLOAT***

The /HFLOAT qualifier causes VAX BASIC to allocate 128 bits of storage in H\_floating format as the default size for all floating-point data not explicitly typed in the program. Untyped floating-point values are treated as H\_floating values and must be in the H\_floating range or VAX BASIC signals "Floating-point error or overflow." Table 1-2 in this manual lists VAX BASIC data types and ranges. The default is /SINGLE.

## ***/[NO]LINES***

The /LINES qualifier includes line number information in object modules. If you specify /NOLINES VAX BASIC does not include line number information in object modules. If you specify /NOLINES in a program containing a RESUME statement or the run-time ERL function, VAX BASIC issues a warning that the /NOLINES qualifier has been overridden. The default is /LINES.

# COMPILE

## ***/[NO]LIST***

The ***/LIST*** qualifier causes VAX BASIC to produce a compiler listing file. This compiler listing generated by the ***/LIST*** qualifier contains a memory allocation map. By default, the name of the listing file is the same as the name of the first program module specified, and has a default file type of LIS. If you specify ***/NOLIST*** VAX BASIC does not generate a compiler listing. ***/NOLIST*** is the default.

## ***/LONG***

The ***/LONG*** qualifier causes VAX BASIC to allocate 32 bits of storage as the default size for all integer data not explicitly typed in the program. Untyped integer values are treated as LONG values and must be in the LONG range or VAX BASIC signals the error "Integer error or overflow." Table 1-2 in this manual lists VAX BASIC data types and ranges. ***/LONG*** is the default.

## ***/[NO]MACHINE\_CODE***

When you specify the ***/MACHINE\_CODE*** qualifier with the ***/LIST*** qualifier in the **COMPILE** command, VAX BASIC includes the machine code generated by the compilation in the program listing file. If you specify ***/NOMACHINE\_CODE***, VAX BASIC does not include a machine code section in the listing file. ***/NOMACHINE\_CODE*** is the default.

## ***/[NO]OBJECT***

The ***/OBJECT*** qualifier generates an object module with the same file name as the program and a default file type of OBJ. The ***/NOOBJECT*** qualifier allows you to check your program for errors without creating an object file. ***/OBJECT*** is the default.

## ***/[NO]OVERFLOW [ { : } ( { INTEGER } ,... ) ]*** ***= { DECIMAL } ,... ) ]***

The ***/OVERFLOW*** qualifier causes VAX BASIC to report arithmetic overflow for operations on integer or packed decimal data, or both. If you specify ***/NOOVERFLOW***, VAX BASIC does not report arithmetic overflows. The default is ***/OVERFLOW=(INTEGER,DECIMAL)***.

## ***/[NO]ROUND***

The ***/ROUND*** qualifier causes VAX BASIC to round rather than truncate DECIMAL values. If you specify ***/NOROUND***, VAX BASIC truncates DECIMAL values. The default is ***/NOROUND***.

## ***/[NO]SETUP***

The ***/SETUP*** qualifier causes VAX BASIC to make calls to the Run-Time Library to set up the stack for VAX BASIC variables, set up dynamic string and array descriptors, initialize variables, and enable VAX BASIC error handling. If you specify the ***/NOSETUP*** qualifier, VAX BASIC will attempt to optimize your program by omitting these calls. If your program contains any of the following elements, VAX BASIC provides an informational diagnostic and does not optimize your program:

- CHANGE statements
- DEF or DEF\* statements
- Dynamic string variables
- Executable DIM statements
- EXTERNAL string functions
- MAT statements
- MOVE statements for an entire array
- ON ERROR statements
- READ statements
- REMAP statements
- RESUME statements
- WHEN blocks
- All graphics statements
- String concatenation
- Built-in string functions
- Virtual array declarations

Note that program modules compiled with the ***/NOSETUP*** qualifier cannot perform I/O and have no error-handling capabilities. If an error occurs in such a module, the error is resigaled to the calling program. The default is ***/SETUP***.

# COMPILE

**`/[NO]SHOW [ { : } ( { [NO]CDD_DEFINITIONS  
[NO]ENVIRONMENT  
[NO]INCLUDE  
[NO]MAP  
[NO]OVERRIDE } ,... ) ]`**

The `/SHOW` qualifier (when used with the `/LIST` qualifier) tells VAX BASIC what to include in the compiler listing file. You can specify the following `/SHOW` qualifier items:

- `CDD_DEFINITIONS` causes VAX BASIC to include a section of translated CDD definitions
- `ENVIRONMENT` causes VAX BASIC to include a list compilation qualifiers in effect
- `INCLUDE` causes VAX BASIC to include a section on the contents of any `%INCLUDE` files
- `MAP` causes VAX BASIC to include a storage allocation map section
- `OVERRIDE` cancels the effect of all `%NOLIST` directives in the source program

For example, if you specify the following command, VAX BASIC includes a storage allocation map section in the compiler listing:

```
COMPILE/LIST/SHOW=MAP
```

If you specify a `/SHOW` qualifier but do not specify any `/SHOW` items, VAX BASIC includes *all* the aforementioned sections in the listing. If you specify `/NOSHOW`, VAX BASIC does not add any additional sections to the compiler listing. The default is `/SHOW`.

## **`/SINGLE`**

The `/SINGLE` qualifier causes VAX BASIC to allocate 32 bits of storage in `F_floating` format as the default size for all floating-point data not explicitly typed in the program. Untyped floating-point values are treated as `SINGLE` values and must be in the `SINGLE` range or VAX BASIC signals the error "Floating-point error or overflow." Table 1–2 in this manual lists VAX BASIC data types and ranges. The default is `/SINGLE`.

## ***/[NO]SYNTAX\_CHECK***

The */SYNTAX\_CHECK* qualifier causes VAX BASIC to perform syntax checking after each program line is typed. If you specify */NOSYNTAX\_CHECK*, VAX BASIC does not perform syntax checking after each program line is typed. The default is */NOSYNTAX\_CHECK*.

## ***/[NO]TRACEBACK***

The */TRACEBACK* qualifier causes VAX BASIC to include traceback information in the object file that allows reporting of the sequence of calls that transferred control to the statement where an error occurred. The */NOTRACEBACK* qualifier tells VAX BASIC not to include traceback information in the object file. The default is */TRACEBACK*.

***/TYPE\_DEFAULT*** {  $\begin{matrix} : \\ = \end{matrix}$  }  $\left\{ \begin{matrix} \textit{REAL} \\ \textit{INTEGER} \\ \textit{DECIMAL} \\ \textit{EXPLICIT} \end{matrix} \right\}$

The */TYPE\_DEFAULT* qualifier sets the default data type (REAL, INTEGER, or DECIMAL) for all data not explicitly typed in your program or specifies that all data must be explicitly typed (EXPLICIT).

- REAL specifies that all data not explicitly typed is floating-point data of the default size (SINGLE, DOUBLE, GFLOAT, or HFLOAT).
- INTEGER specifies that all data not explicitly typed is integer data of the default size (BYTE, WORD, or LONG).
- DECIMAL specifies that all data not explicitly typed is packed decimal data of the default size.
- EXPLICIT specifies that all data in a program must be explicitly typed. Implicitly declared variables cause VAX BASIC to signal an error.

The default is *TYPE\_DEFAULT=REAL*.

***/VARIANT*** {  $\begin{matrix} : \\ = \end{matrix}$  } *int-const*

The */VARIANT* qualifier establishes *int-const* as a value to be used in compiler directives. The variant value can be referenced in a lexical expression with the lexical function, *%VARIANT*. *Int-const* always has a data type of LONG. The default is */VARIANT=0*.

# COMPILE

***/[NO]WARNINGS [ { : } { [NO]WARNINGS  
[NO]INFORMATIONALS } ]***

The ***/WARNINGS*** qualifier causes VAX BASIC to display warning or informational messages, or both. If you specify ***/WARNINGS*** but do not specify a warning clause, VAX BASIC displays both warnings and informational messages. If you specify ***/NOWARNINGS***, VAX BASIC does not display warning and informational messages. The default is ***/WARNINGS***.

## ***/WORD***

The ***/WORD*** qualifier causes VAX BASIC to allocate 16 bits of storage as the default for all integer data not explicitly typed in the program. Untyped integer values are treated as ***WORD*** values and must be in the range -32768 to 32767 or VAX BASIC signals the error "Integer error or overflow." Table 1-2 in this manual lists VAX BASIC data types and ranges. The default is ***/LONG***.

In the following example, VAX BASIC compiles the program LETSGO and creates a new version of the object file as well as a listing file. In addition, VAX BASIC allocates 64 bits of storage in ***D\_FLOAT*** format as the default for all floating point data not explicitly typed in the program.

---

## Example

```
COMPILE LETSGO/DOUBLE/LIST
```



---

## CONTINUE

The CONTINUE command continues program execution after VAX BASIC executes a STOP statement or encounters a CTRL/C.

---

### Format

CONTINUE

---

### Syntax Rules

None.

---

### Remarks

1. After a STOP statement or a CTRL/C, you can enter immediate mode commands and resume program execution with the CONTINUE command.
2. After a STOP statement or a CTRL/C, you cannot resume program execution if you have made source code changes or additions.

---

### Example

```
%BAS-I-STO, Stop
-BAS-I-FROLINMOD, from line 25 in module ABC
Ready
CONTINUE
```

# DELETE

---

## DELETE

The DELETE command removes a specified line or range of lines from the program currently in memory.

---

### Format

DELETE *line-num* [ *- line-num*  
*,line-num...* ] ...

---

### Syntax Rules

None.

---

### Remarks

1. You cannot specify the DELETE command on programs that do not contain line numbers.
2. The separator characters (comma or hyphen) allow you to delete individual lines or a block of lines.
  - If you separate line numbers with commas, VAX BASIC deletes each specified line number.
  - If you separate line numbers with a hyphen, VAX BASIC deletes the inclusive range of lines. The lower line number must be specified first. If it is not specified first, the DELETE command has no effect.
3. You can combine individual line numbers and line ranges in a single DELETE command. Note, however, that a line number range must be followed by a comma and not another hyphen, or VAX BASIC signals an error.

4. VAX BASIC signals an error if there are no lines in the specified range or if you specify an illegal line number.

---

## Examples

### Example 1

```
DELETE 50
```

### Example 2

```
DELETE 70-80, 110, 124
```

### Example 3

```
DELETE 50, 60, 90-110
```

# EDIT

---

## EDIT

The EDIT command allows you to edit individual program lines in the BASIC environment while invoking an editor. EDIT with no arguments invokes a text editor and reads the current program into the editor's buffer.

---

### Format

**EDIT** [ *line-num* *search-clause* [ *replace-clause* ] ]

*search-clause:* *delim unq-str1 delim*

*replace-clause:* [ *unq-str2* ] [ *delim* [ *int-const1* ] [ *,int-const2* ] ]

---

### Syntax Rules

1. *Line-num* specifies the line to be edited.
2. *Search-clause* specifies the text you want to remove or replace. *Unq-str1* is the search string you want to remove or replace.
3. *Replace-clause* specifies the replacement text and the occurrence of the search string you want to replace.
  - *Unq-str2* is the replacement string.
  - *Int-const1* specifies the occurrence of *unq-str1* you want to replace. If you do not specify an occurrence, VAX BASIC replaces the first occurrence of *unq-str1*.
  - *Int-const2* specifies the line number of a block of program code where you want VAX BASIC to begin the search.
4. *Delim* can be any printing character not used in *unq-str1* or *unq-str2*. The examples for this command use the slash (/) as a delimiter.

---

## Remarks

1. The *delim* characters in *search-clause* must match, or VAX BASIC signals an error.
2. If the delimiter you use to signal the end of *replace-clause* does not match the delimiter used in *search-clause*, VAX BASIC does not signal an error and treats the end delimiter as part of *unq-str2*.
3. VAX BASIC replaces or removes text in a program line as follows:
  - If *unq-str1* is found, VAX BASIC replaces it with *unq-str2*.
  - If *unq-str1* is not found, VAX BASIC signals an error.
  - If *unq-str1* is null, VAX BASIC signals “No change made”.
  - If *unq-str2* is null, VAX BASIC deletes *unq-str1*.
  - VAX BASIC matches and replaces strings exactly as you type them. If *unq-str1* is uppercase, VAX BASIC searches for an uppercase string. If it is lowercase, VAX BASIC searches for a lowercase string.
4. VAX BASIC displays the edited line with changes after the EDIT command successfully executes.
5. If you specify a line number with no text parameters, VAX BASIC displays the line.
6. The EDIT command followed by a carriage return causes VAX BASIC to save your current source file in BASEDITMP.BAS and automatically invoke VAX EDT as the default text editor.
7. At DCL level, you can override the default text editor. To do this, assign the logical name BASIC\$EDIT to another editor such as VAXTPU or Language-Sensitive Editor before entering the BASIC environment. For instance, in the following example, BASIC\$EDIT is defined to be TPU\$EDIT. The EDIT command followed by a carriage return will then invoke VAXTPU as the default text editor.

```
$ DEFINE BASIC$EDIT TPU$EDIT
```
8. If you define BASIC\$EDIT to be an editor other than VAX EDT, VAXTPU, or Language-Sensitive Editor, VAX BASIC spawns a subprocess to invoke the editor assigned to BASIC\$EDIT.

# EDIT

9. When you finish editing your program and exit from the editor, the edited program is the program currently in memory, and the context of the BASIC environment is unchanged. Note that VAX BASIC deletes all versions of BASEDITMP.BAS when you return to VAX BASIC from the editor.

---

## Example

```
Ready
LIST 100
100 NEW_STRING$ = LEFT$(STRING$,12)
EDIT 100 /LEFT$/RIGHT$/3,2
LIST 100
100 NEW_STRING$ = RIGHT$(STRING$,12)
```

---

**EXIT**

The **EXIT** command or **CTRL/Z** clears the memory and returns control to the operating system.

---

**Format**

**EXIT**

---

**Syntax Rules**

None.

---

**Remarks**

If you type **EXIT** after creating a new program or editing an old program without first typing **SAVE** or **REPLACE**, VAX BASIC signals "Unsaved change has been made, **CTRL/Z** or **EXIT** to exit." The message warns you that the new or revised program will be lost if you do not **SAVE** or **REPLACE** it. If you type **EXIT** again, VAX BASIC exits from the BASIC environment whether you have saved your changes or not.

# EXIT

---

## Example

```
EXIT
%BASIC-W-CHANGES, unsaved change has been made, CTRL/Z or EXIT to exit
Ready

SAVE

EXIT
Ready
```



---

## **HELP**

The **HELP** command displays online documentation for VAX BASIC commands, keywords, statements, functions, and conventions.

---

### **Format**

**HELP** [*unq-str*] ...

---

### **Syntax Rules**

1. *Unq-str* is a VAX BASIC topic, keyword, command, statement, function, or convention.
2. The first *unq-str* must be one of the topics described in the **HELP** file. If it is not, VAX BASIC displays a list of topics for you to choose from.
3. You can specify a subtopic after the topic. Separate one *unq-str* from another with a space.
4. You can use the asterisk (\*) wildcard character in *unq-str*. VAX BASIC then matches any portion of the specified topic.

---

### **Remarks**

1. If you type **HELP** with no parameters, VAX BASIC displays a list of statements, functions, compiler directives, compiler commands and language topics.
2. If the *unq-str* you specify is not a unique topic or subtopic, VAX BASIC displays information on all topics or subtopics beginning with *unq-str*.
3. An asterisk (\*) indicates that you want to display information that matches any portion of the topic you specify. For example, if you type **HELP GO\***, VAX BASIC displays information on the **GOSUB** statement and the **GOTO** statement.

# HELP

4. When information on a particular topic or subtopic is not available, VAX BASIC signals the message "Sorry, no documentation on *unq-str*" and provides a list of alternative HELP topics to choose from.

---

## Example

Ready

Help GO\*

GOSUB

The GOSUB statement transfers control to a specified line number or label and stores the location of the GOSUB statement for eventual return from the subroutine.

Example

200 GOSUB 1100

Additional information available:

Syntax

GOTO

The GOTO statement transfers control to a specified line number or label.

Example

20 GOTO 200

Additional information available:

Syntax

Topic?

---

## IDENTIFY

The **IDENTIFY** command displays an identification header on the controlling terminal. The header contains the name and version number of VAX BASIC.

---

### Format

**IDENTIFY**

---

### Syntax Rules

None.

---

### Remarks

The message displayed by the **IDENTIFY** command includes the name of the VAX BASIC compiler and the version number.

---

### Example

```
IDENTIFY  
VAX BASIC V3.4
```

## **INQUIRE**

---

## **INQUIRE**

The **INQUIRE** command is a synonym for the **HELP** command. See the **HELP** command for more information.

---

## LIST and LISTNH

The LIST command displays the program lines of the program currently in memory. Line numbers are sequenced in ascending order. The LISTNH command displays program lines without the program header.

---

### Format

$$\text{LIST[NH]} \left[ \text{line-num} \left[ \begin{array}{l} - \text{line-num} \\ , \text{line-num...} \end{array} \right] \right] \dots$$

---

### Syntax Rules

A *line-num* followed by a hyphen (-) and a carriage return displays the specified line and all remaining lines in the program.

---

### Remarks

1. The LIST command displays program lines, along with a header containing the program name, the current time, and the date. To suppress the program header, type LISTNH.
2. LIST without parameters displays the entire program.
3. The separator characters (comma or hyphen) allow you to display individual lines or a block of lines.
  - If you separate line numbers with commas, VAX BASIC displays each specified line number.
  - If you separate line numbers with hyphens, VAX BASIC displays the inclusive range of lines. The lower line number must come first. If it does not, LIST has no effect.

## LIST and LISTNH

4. You can combine individual line numbers and line ranges in a single LIST command. Note, however, that a line number range must be followed by a comma and not another hyphen, or VAX BASIC signals an error.
5. A hyphen between the list command and *line-num* causes VAX BASIC to signal an error.
6. VAX BASIC displays the source program lines in the order you specify in the command line. VAX BASIC displays line 100 before line 10 if you type LIST 100,10.

---

### Example

#### Example

```
LIST 200-300
```

#### Output

```
200 %IF %VARIANT = 2 %THEN %ABORT  
300 %END %IF
```

---

## LOAD

The **LOAD** command makes a previously created object module or modules available for execution with the **RUN** command.

---

### Format

**LOAD** *file-spec* [ + *file-spec* ] ...

---

### Syntax Rules

*File-spec* must be a VAX BASIC object module or VAX BASIC signals an error. OBJ is the default file type. If you specify only the file name, VAX BASIC searches for an OBJ file in the current default directory.

---

### Remarks

1. Each device and directory specification applies to all following file specifications until you specify a new directory or device.
2. The **LOAD** command accepts multiple device, directory, and file specifications.
3. VAX BASIC does not process the loaded object files until you issue the **RUN** command. Consequently, errors in the loaded modules may not be detected until you execute them.
4. VAX BASIC signals an error in the following cases:
  - If the file is not found
  - If the file specification is not valid
  - If the file is not a VAX BASIC object module
  - If run-time memory is exceeded

Errors do not change the program currently in memory.

# LOAD

5. The LOAD command clears all previously loaded object modules from memory.
6. Typing the LOAD command does not change the program currently in memory.

---

## Example

```
LOAD PROGA + PROGB + PROGC
```



---

**LOCK**

The **LOCK** command changes default values for **COMPILE** command qualifiers. It is a synonym for the **SET** command. See the **SET** command for more information.

# NEW

---

## NEW

The **NEW** command clears VAX BASIC memory and allows you to assign a name to a new program.

---

### Format

**NEW** [*prog-name*]

---

### Syntax Rules

*Prog-name* is the name of the program you want to create. VAX BASIC allows program names to contain a maximum of 39 characters. You can use any combination of alphanumeric characters in your program name, as well as the dollar sign (\$), hyphen (-), and underscore (\_) characters.

---

### Remarks

1. VAX BASIC signals an error if *prog-name* exceeds 39 characters.
2. VAX BASIC signals “error in program name” if you specify a file type.
3. If you do not specify a *prog-name*, VAX BASIC prompts with:  
New file name--
4. The default name is NONAME. If you do not provide a *prog-name* in response to the prompt, VAX BASIC assigns the file name NONAME to your program.
5. When you type the **NEW** command, the program currently in memory is cleared. Program modules loaded with the **LOAD** command remain unchanged.

---

**Example**

NEW PROG1

# OLD

---

## OLD

The OLD command brings a previously created VAX BASIC program into memory.

---

### Format

OLD [ *file-spec* ]

---

### Syntax Rules

1. If you do not name a *file-spec*, VAX BASIC prompts for one. If you do not enter a *file-spec* in response to the prompt, VAX BASIC searches for a file named NONAME.BAS in the current default directory.
2. The default file type is BAS.

---

### Remarks

1. If the VAX BASIC compiler cannot find the file you specify, VAX BASIC signals the error "File not found".
2. When the specified file is found, it is placed in memory and any program currently in memory is erased. If VAX BASIC does not find the specified file, the program currently in memory does not change.

---

### Example

```
OLD CHECK  
Ready
```

---

## RENAME

The RENAME command allows you to assign a new name to the program currently in memory. VAX BASIC does not write the renamed program to a file until you save the program with the REPLACE or SAVE command.

---

### Format

**RENAME** [*prog-name*]

---

### Syntax Rules

1. *Prog-name* specifies the new program name. VAX BASIC allows program names to contain a maximum of 39 characters. You can use any combination of alphanumeric characters in your program name, as well as the dollar sign (\$), hyphen (-), and underscore (\_) characters.
2. If you specify a file type, VAX BASIC signals the error "Error in program name."

---

### Remarks

1. The program you want to rename must be in memory. If you type RENAME with no program in memory, VAX BASIC renames the default program, NONAME, to the specified *prog-name*.
2. If you do not specify a *prog-name*, VAX BASIC renames the program currently in memory NONAME.

## RENAME

3. You must type **SAVE** or **REPLACE** to write the renamed program to a file. If you do not type **SAVE** or **REPLACE**, VAX BASIC does not save the renamed program.
4. The **RENAME** command does not affect the original saved version of the program.

---

### Example

```
OLD TEST
Ready

RENAME NEWTEST
Ready

LIST
NEWTEST  06-OCT-1989 13:50
PRINT "This program is a simple test"
.
.
.

Ready

SAVE
%BASIC-I-FILEWRITE, NEWTEST written to file:
USER$$DISK:[SMITH.COMS]NEWTEST.BAS;5

Ready
```

In this example, the **OLD** command calls the program named **TEST** into memory. The **RENAME** command renames **TEST** to **NEWTEST** and the **SAVE** command writes **NEWTEST.BAS** to a file. The original file, **TEST.BAS** is not changed and is not deleted from your account.

---

## REPLACE

The REPLACE command writes the current program back to the file specified by the last OLD command.

---

### Format

**REPLACE**

---

### Syntax Rules

None.

---

### Remarks

1. If you do not have write access to the directory containing the original file, VAX BASIC signals an error message.
2. VAX BASIC creates and saves a new version of the file, incrementing the version number by 1 unless you supplied a specific version number with the OLD command.
3. A REPLACE command following a NEW command or a SCRATCH command causes VAX BASIC to write the program in memory to the current default directory.
4. A REPLACE command following a RENAME command writes the file to the directory specified in the OLD command with the file name specified in the RENAME command.

# REPLACE

---

## Example

```
$ DIR NODE::USER$$DISK:[BASICUSER] TEST.BAS
Directory USER$$DISK:[BASICUSER]
TEST.BAS;1
Total of 1 file.
$ BASIC
VAX BASIC V3.4
Ready
OLD NODE::USER$$DISK:[BASICUSER] TEST.BAS;
.
.
Ready
REPLACE
%BASIC-I-FILEWRITE, TEST written to file:
  USER$$DISK:[BASICUSER] TEST.BAS;2
Ready
EXIT
$ DIR NODE::USER$$DISK:[BASICUSER] TEST.BAS
Directory USER$$DISK:[BASICUSER]
TEST.BAS;1 TEST.BAS;2
Total of 2 files.
$
```



---

## RESEQUENCE

In a program with line numbers, the RESEQUENCE command allows you to resequence the line numbers of the program currently in memory. VAX BASIC also changes all references to the old line numbers so they reference the new line numbers.

---

### Format

**RESEQUENCE** [ *line-num1* [- *line-num2*] [*line-num3*] ] [STEP *int-const* ]

---

### Syntax Rules

1. *Line-num1* is the line number in the program currently in memory where resequencing begins. The default for *line-num1* is the first line of the program module.
2. *Line-num2* is the optional end of the range of line numbers to be resequenced. If you specify a range, VAX BASIC begins resequencing with *line-num1* and resequences through *line-num2*. If you do not specify *line-num2*, VAX BASIC resequences the specified line. If you do not specify either *line-num1* or *line-num2*, VAX BASIC resequences the entire program.
3. *Line-num3* specifies the new first line number; the default number for the new first line is 100. You can specify *line-num3* only when resequencing a range of lines.  
If *line-num3* causes existing lines to be deleted or surrounded, VAX BASIC signals an error.
4. *Int-const* specifies the numbering increment for the resequencing operation. The default for *int-const* is 10.

# RESEQUENCE

---

## Remarks

1. You cannot specify the RESEQUENCE command on programs that do not contain line numbers.
2. VAX BASIC signals an error when you try to resequence a program that contains a %IF directive. VAX BASIC also signals an error when you try to resequence a program that has a %INCLUDE directive if the file to be included contains a reference to a line number.
3. Before the RESEQUENCE command executes, VAX BASIC verifies the syntax of the program. If the program is not syntactically valid, the RESEQUENCE command does not execute.
4. VAX BASIC sorts the renumbered program in ascending order when the RESEQUENCE command executes.
5. If the renumbering creates a line number greater than the maximum line number of 32767, VAX BASIC signals an error.
6. VAX BASIC signals an error if resequencing causes a change in the order in which program statements are to execute and does not resequence the program.
7. VAX BASIC signals the error “Undefined line number” in the case of undefined line numbers and does not resequence the program.
8. VAX BASIC corrects all line numbers for statements that transfer control.
9. VAX BASIC does not modify the program currently in memory when the RESEQUENCE command generates an error.
10. In general, the RESEQUENCE command is not recommended for programs containing error handlers that test the value of ERL. However, the RESEQUENCE command correctly modifies the program if the tests that reference ERL are of this form:

ERL relational-operator int-lit

# RESEQUENCE

The RESEQUENCE command does not correctly renumber programs if the test compares ERL with an expression or a variable, or if ERL follows the relational operator. The following line number references, for example, would not be correctly renumbered:

```
IF ERL = 1000 + A% THEN ...
IF 1000 > ERL THEN ...
```

---

## Example

```
10 INPUT "Enter a numeric value";A%
20 IF A% = 20
30   THEN PRINT "Bye"
40     GOTO done
50   ELSE GOTO 10
60 END IF
```

### Output

```
15 INPUT "Enter a numeric value";A%
25 IF A% = 20
35   THEN PRINT "Bye"
45     GOTO done
55   ELSE GOTO 10
65 END IF
```

In this example, the command RESEQUENCE 10–60 STEP 5 causes VAX BASIC to resequence lines 10 through 60, incrementing each new line number by 5.

# RUN

---

## RUN

The RUN command allows you to execute a program from the BASIC environment without first invoking the VMS Linker to construct an executable image. In addition, the RUN command allows you to access user specified and system shareable image libraries for undefined symbols.

---

### Format

**RUN[NH]** [ *file-spec* ]

---

### Syntax Rules

None.

---

### Remarks

#### 1. Executing a Program

- If you specify only the file name, VAX BASIC searches for a file with a BAS file type in the current default directory.
- If you do not supply a *file-spec*, VAX BASIC executes the program currently in memory.
- VAX BASIC signals the warning message “No main program” if you do not supply a *file-spec* and do not have a program currently in memory.
- The RUNNH command is identical to RUN, except that it does not display the program header, current date, and time.

- When you specify a *file-spec* with the RUN command, VAX BASIC brings the program into memory and then executes it. You do not have to bring a program into memory with the OLD command in order to run it. The RUN command executes just as if the program had been brought into memory with the OLD command.
- If your program calls a subprogram, the subprogram must be compiled and placed in memory with the LOAD command. If your program tries to call a subprogram that has not been compiled and loaded, VAX BASIC signals an error.
- The RUN command does not create an object module file or a list file.
- When VAX BASIC encounters a STOP statement in the program, the program stops executing and control passes to the BASIC environment immediate mode.
- Any VAX BASIC statement that does not require the creation of new storage can be entered in immediate mode to debug the program. You cannot create new variables in immediate mode.
- Type the CONTINUE command to resume program execution.
- The RUN command uses whatever qualifiers have been set, with the exception of those that have no effect on a program running in the BASIC environment. These qualifiers are as follows:
  - NOCROSS
  - NODEBUG
  - NOLIST
  - NOMACHINE
  - NOOBJECT

These qualifiers are always in effect when you run a program in the environment.

## 2. Accessing Shareable Images

- To automatically access shareable image libraries, you must make an assignment to the logical name BASIC\$LIB*n*. For example:

```
$ ASSIGN DBAO:[BABCOCK]TESTLIB.OLB BASIC$LIB0
```
- After you enter a command line, VAX BASIC will automatically access your library to resolve undefined program symbols.

# RUN

- If you have more than one library for the VMS Linker to search, you must assign the first one as BASIC\$LIB0, the second one as BASIC\$LIB1, the third as BASIC\$LIB2, and so on.
- If you do not number libraries consecutively, the VMS Linker does not search past the first missing logical name.
- As long as routines are contained in shareable images in libraries, they are not required to be written in VAX BASIC to be accessed with the RUN command.
- VAX BASIC provides no default file specification for user-supplied shareable image libraries; the current default device and the directory are used.
- After all possible shareable image libraries have been accessed, VAX BASIC will subsequently search the default library SYS\$LIBRARY:OLB with the logical name IMAGELIB to resolve any additional undefined program symbols.

---

## Example

```
RUN PROG1
PROG1  06-OCT-1989 13:52
  1
  3
  6
 10
Ready

RUNNH PROG1
  1
  3
  6
 10
Ready
```

---

## **SAVE**

The **SAVE** command writes the VAX BASIC source program currently in memory to a file on the default or specified device.

---

### **Format**

**SAVE** [ *file-spec* ]

---

### **Syntax Rules**

None.

---

### **Remarks**

1. If you do not supply a *file-spec*, VAX BASIC saves the file with the name of the program currently in memory and the BAS default file type.
2. If you specify only the file name, VAX BASIC saves the program with the default file type in the current default directory.
3. When you type the **SAVE** command, VAX BASIC writes a new version of the program.
4. VAX BASIC stores the sorted program in ascending line number order.
5. You can store the program on a specified device. For example:

```
SAVE DUA1:NEWTEST.PRO
```

VAX BASIC saves the file NEWTEST.PRO on disk DUA1:.

# SAVE

---

## Example

```
SAVE PROG_SAMP.BAS
%BASIC-I-FILEWRITE, PROG_SAMP written to file:
USER$DISK[BASICUSER]PROG_SAMP.BAS;2
```



---

## SCALE

The **SCALE** command allows you to control accumulated round-off errors by multiplying numeric values by 10 raised to the scale factor before storing them.

---

### Format

**SCALE** *int-const*

---

### Syntax Rules

*Int-const* specifies the power of 10 you want to use as the scaling factor. *Int-const* must be an integer from 0 through 6 or VAX BASIC signals the error "Illegal argument for command."

---

### Remarks

1. **SCALE** with no argument causes VAX BASIC to signal the error "Illegal argument for command."
2. **SCALE** affects only values of the data type **DOUBLE**.
3. VAX BASIC multiplies values using the scale factor you specify. For example, the value 2.488888 is rounded as is shown in Table 2-1.

# SCALE

**Table 2-1: Multiplying a Numeric Value with the SCALE Command**

<b>Scale</b>	<b>Value Produced for 2.488888</b>
0	2.48889
1	2.4
2	2.48
3	2.488
4	2.4888
5	2.48888
6	2.48889

---

## Example

SCALE 2

---

## SCRATCH

The SCRATCH command clears any program currently in memory, removes any object files loaded with the LOAD command, and resets the program name to NONAME.

---

### Format

SCRATCH

---

### Syntax Rules

None.

---

### Remarks

None.

---

### Example

SCRATCH

# SEQUENCE

---

## SEQUENCE

The SEQUENCE command causes VAX BASIC to automatically generate line numbers for your program text. VAX BASIC supplies line numbers for your text until you end the procedure or reach the maximum line number of 32767.

---

### Format

**SEQUENCE** [ *line-num* ] [ , *int-const* ]

---

### Syntax Rules

1. *Line-num* specifies the line number where sequencing begins.
2. *Int-const* specifies the line number increment for your program. If you do not specify an increment, VAX BASIC defaults to the *int-const* specified in the last SEQUENCE command; if there is no previous SEQUENCE command, the default is 10.

---

### Remarks

1. You cannot specify the SEQUENCE command on programs that do not contain line numbers.
2. If you do not specify a *line-num*, the VAX BASIC default is the last line inserted by a SEQUENCE command; if there is no previous SEQUENCE command, the default is line number 100.
3. If you specify a *line-num* that already contains a statement, or if the sequencing operation generates a line number that already contains a statement, VAX BASIC signals "Attempt to sequence over existing statement", and returns to normal input mode.

## SEQUENCE

4. Type your program text in response to the line number prompt; the carriage return ends each line and causes VAX BASIC to generate a new line number.
5. If you press CTRL/Z in response to the line number prompt, VAX BASIC terminates the sequencing operation and prompts for another command.
6. When the maximum line number of 32767 is reached, VAX BASIC terminates the sequencing process and returns to normal input mode.
7. VAX BASIC does not check syntax during the sequencing process.

---

### Example

```
SEQUENCE 100,10
100 INPUT "Enter a numeric value";A%
110 IF A% = 20
```

In this example, the command SEQUENCE 100,10 causes VAX BASIC to automatically generate line numbers into the program text, beginning with the line number 100 and incrementing each line by 10.

# SET

---

## SET

The SET command allows you to specify VAX BASIC defaults for all VAX BASIC qualifiers. Qualifiers control the compilation process and the run-time environment. The defaults you set remain in effect for all subsequent operations until they are reset or until you exit from the compiler.

---

### Format

SET [/qualifier ]...

---

### Syntax Rules

1. /*Qualifier* specifies a qualifier keyword that sets a VAX BASIC default. See the COMPILE command for a list of all VAX BASIC qualifiers and their defaults.
2. VAX BASIC signals the error "Unknown qualifier" if you do not separate multiple qualifiers with commas (,) or slashes (/), or if you mix commas and slashes on the same command line. The same error is signaled if you separate qualifiers with a slash but do not prefix the first qualifier with a slash.

---

### Remarks

If you do not specify any qualifiers, VAX BASIC resets all defaults to the defaults specified with the DCL command BASIC.

---

## Examples

### Example 1

```
SET /DOUBLE/BYTE/LIST
```

### Example 2

```
SET DOUBLE,BYTE,LIST
```

In these examples, the SET command causes VAX BASIC to allocate 64 bits of storage for all floating-point data, and to allocate 8 bits of storage for all integer data. A source listing file is also created.

## SHOW

---

### SHOW

The **SHOW** command displays the current defaults for the VAX BASIC compiler on your terminal.

---

#### Format

**SHOW**

---

#### Syntax Rules

None.

---

#### Remarks

None.



---

**Example**

```
SHOW
VAX BASIC 3.4      Current Environment Status      09-OCT-1989 10:05:56.57
DEFAULT DATA TYPE INFORMATION:
  Data type : REAL
  Real size : SINGLE
  Integer size : LONG
  Decimal size : (15,2)
  Scale factor : 0
  NO Round decimal numbers
LISTING FILE INFORMATION INCLUDES:
  NO List
  NO Cross reference
  CDD Definitions
  Environment
  NO Override of %NOLIST
  NO Machine code
  Map
  INCLUDE files
COMPILATION QUALIFIERS IN EFFECT:
  Object file
  Overflow check integers
  Overflow check decimal numbers
  Bounds checking
  NO Syntax checking
  Lines
  Variant : 0
  Warnings
  Informationals
  Setup
  Object Libraries : NONE
FLAGGERS:
  NO Declining features
  NO BASIC PLUS 2 subset
DEBUG INFORMATION:
  Traceback records
  NO Debug symbol records
Ready
```

# UNSAVE

---

## UNSAVE

The UNSAVE command deletes a specified file from storage.

---

### Format

**UNSAVE** [ *file-spec* ]

---

### Syntax Rules

None.

---

### Remarks

1. If you do not supply a *file-spec*, VAX BASIC deletes a file that has the file name of the program currently in memory and a file type of BAS.
2. If you do not supply a *file-spec* and do not have a program in memory, VAX BASIC searches for the default file NONAME.BAS.
3. If you do not specify a complete file name with a file type, VAX BASIC deletes the file with the specified name and the BAS file type from the default device and directory. Other file types with the same file name are not deleted.

---

### Example

```
UNSAVE DB2:CHECK.DAT
```

## Chapter 3

# Compiler Directives

---

Compiler directives are instructions that cause VAX BASIC to perform certain operations as it translates the source program. This chapter describes all of the compiler directives supported by VAX BASIC. The directives are listed and discussed alphabetically.

# %ABORT

---

## %ABORT

The %ABORT directive terminates program compilation and displays a fatal error message that you can supply.

---

### Format

**%ABORT** [ *str-lit* ]

---

### Syntax Rules

None.

---

### Remarks

1. Only a line number or a comment field can appear on the same physical line as the %ABORT directive.
2. VAX BASIC stops the compilation and terminates the listing file as soon as it encounters a %ABORT directive. An optional *str-lit* is displayed on the terminal screen and in the compilation listing, if a listing has been requested.

---

### Example

```
%IF %VARIANT = 2 %THEN
    %ABORT "Cannot compile with variant 2"
%END %IF
```

---

## **%CROSS**

The `%CROSS` directive causes VAX BASIC to begin or resume accumulating cross-reference information for the listing file.

---

### **Format**

`%CROSS`

---

### **Syntax Rules**

None.

---

### **Remarks**

1. Only a line number or a comment field can appear on the same physical line as the `%CROSS` directive.
2. The `%CROSS` directive has no effect unless you request both a listing file and a cross-reference. For more information on listing file format, see the *VAX BASIC User Manual*.
3. When a cross-reference is requested, the VAX BASIC compiler starts or resumes accumulating cross-reference information immediately after encountering the `%CROSS` directive.

---

### **Example**

```
%CROSS
```

## **%DECLARED**

---

## **%DECLARED**

The **%DECLARED** directive is a built-in lexical function that allows you to determine whether a lexical variable has been defined with the **%LET** directive. If the lexical variable named in the **%DECLARED** function is defined in a previous **%LET** directive, the **%DECLARED** function returns the value **-1**. If the lexical variable is not defined in a previous **%LET** directive, the **%DECLARED** function returns the value **0**.

---

### **Format**

**%DECLARED** (*lex-var*)

---

### **Syntax Rules**

1. The **%DECLARED** function can appear only in a lexical expression.
2. The *lex-var* is the name of a lexical variable. Lexical variables are always **LONG** integers.
3. *Lex-var* must be enclosed in parentheses.

---

### **Remarks**

None.

---

## Example

```
! +
! Use the following code in %INCLUDE files
! which reference constants that may be already defined.
! -
%IF %DECLARED (%TRUE_FALSE_DEFINED) = 0
%THEN
    DECLARE LONG CONSTANT True = -1, False = 0
    %LET %TRUE_FALSE_DEFINED = -1
%END %IF
```

## %IDENT

---

## %IDENT

The `%IDENT` directive lets you identify the version of a program module. The identification text is placed in the object module and printed in the listing header.

---

### Format

`%IDENT` *str-lit*

---

### Syntax Rules

*Str-lit* is the identification text. *Str-lit* can consist of up to 31 ASCII characters. If it has more than 31 characters, VAX BASIC truncates the extra characters and signals a warning message.

---

### Remarks

1. Only a line number or a comment field can appear on the same physical line as the `%IDENT` directive.
2. The VAX BASIC compiler inserts the identification text in the first 31 character positions of the second line on each listing page. VAX BASIC also includes the identification text in the object module, if the compilation produces one, and in the map file created by the VMS Linker.
3. The `%IDENT` directive should appear at the beginning of your program if you want the identification text to appear on the first page of your listing. If the `%IDENT` directive appears after the first program statement, the text will appear on the next page of the listing file.



4. You can use the %IDENT directive only once in a module. If you specify more than one %IDENT directive in a module, VAX BASIC signals a warning and uses the identification text specified in the first directive.
5. No default identification text is provided.

---

## Example

```
%IDENT "Version 10"
```

```
.  
.  
.
```

### Output

```
TIME$MAIN  
Version 10
```

```
1 10 %IDENT "Version 10"
```

```
.  
.  
.
```

## **%IF-%THEN-%ELSE-%END %IF**

---

## **%IF-%THEN-%ELSE-%END %IF**

The **%IF-%THEN-%ELSE-%END %IF** directive lets you conditionally include source code or execute another compiler directive.

---

### **Format**

**%IF** *lex-exp* **%THEN** *code* [**%ELSE** *code* ]**%END %IF**

---

### **Syntax Rules**

1. *Lex-exp* is always a LONG integer.
2. *Lex-exp* can be:
  - A lexical constant named in a **%LET** directive.
  - An integer literal, with or without the percent sign suffix.
  - A lexical built-in function.
  - Any combination of the above, separated by valid lexical operators. Lexical operators include logical operators, relational operators, and the arithmetic operators for addition (+), subtraction (−), multiplication (\*), and division (/).
3. *Code* is VAX BASIC program code. It can be any VAX BASIC statement or another compiler directive, including another **%IF** directive. You can nest **%IF** directives to eight levels.

---

### **Remarks**

1. The **%IF** directive can appear anywhere in a program where a space is allowed, except within a quoted string. This means that you can use the **%IF** directive to make a whole statement, part of a statement, or a block of statements conditional.

## %IF-%THEN-%ELSE-%END %IF

2. %THEN, %ELSE, and %END %IF do not have to be on the same physical line as %IF.
3. If *lex-exp* is true, VAX BASIC processes the %THEN clause. If *lex-exp* is false, VAX BASIC processes the %ELSE clause. If there is no %ELSE clause, VAX BASIC processes the %END %IF clause. The VAX BASIC compiler includes statements in the %THEN or %ELSE clause in the source program and executes directives in order of occurrence.
4. You must include the %END %IF clause. Otherwise, VAX BASIC assumes the remainder of the program is part of the last %THEN or %ELSE clause and signals the error "MISENDIF, missing END IF directive" when compilation ends.

---

### Example

```
%IF (%VARIANT = 2)
%THEN DECLARE SINGLE hourly_pay(100)
%ELSE %IF (%VARIANT = 1)
    %THEN DECLARE DOUBLE salary_pay(100)
    %ELSE %ABORT "Can't compile with specified variant"
    %END %IF
%END %IF
.
.
.
PRINT %IF (%VARIANT = 2)
    %THEN 'Hourly Wage Chart'
        GOTO Hourly_routine
    %ELSE 'Salaried Wage Chart'
        GOTO Salary_routine
    %END %IF
```

# **%INCLUDE**

---

## **%INCLUDE**

The **%INCLUDE** directive lets you include VAX BASIC source text from another program file in the current program compilation. VAX BASIC also lets you access record definitions in the VAX Common Data Dictionary (CDD) and access commonly used routines from text libraries.

---

### **Format**

#### **Including a File**

**%INCLUDE** *str-lit*

#### **Including a CDD Definition**

**%INCLUDE** **%FROM** **%CDD** *str-lit*

#### **Including a File from a Text Library**

**%INCLUDE** *str-lit* **%FROM** **%LIBRARY** [*str-lit*]

---

### **Syntax Rules**

#### **1. Including a File**

*Str-lit* must be a valid file specification for the file to be included.

#### **2. Including a CDD Definition**

*Str-lit* specifies a VAX CDD path name enclosed in quotation marks. The path name can be in either DMU or CDO format. This directive lets you extract a RECORD definition from the dictionary.

#### **3. Including a File from a Text Library**

- *Str-lit* specifies a particular module to be included.
- The optional *str-lit* identifies a specific text library in which the included module resides. If the library name is not specified, VAX BASIC uses the default library name BASIC\$LIBRARY.

---

## Remarks

1. Any statement that appears after an END statement inside an included file causes VAX BASIC to signal an error.
2. Only a line number or a comment field can appear on the same physical line as the %INCLUDE directive.
3. The VAX BASIC compiler includes the specified source file in the program compilation at the point of the %INCLUDE directive and prints the included code in the program listing file if the compilation produces one.
4. The included file cannot contain line numbers. If it does, VAX BASIC signals the error “Line number may not appear in %INCLUDE file.”
5. All statements in the accessed file are associated with the line number of the program line that contains the %INCLUDE directive. This means that a %INCLUDE directive cannot appear before the first line number in a source program if you are using line numbers.
6. A file accessed by %INCLUDE can itself contain a %INCLUDE directive.
7. All %IF directives in an included file must have a matching %END %IF directive in the file.
8. You can control whether or not included text appears in the compilation listing with the /[NO]SHOW:INCLUDE qualifier to the COMPILE command. When you specify /SHOW:INCLUDE, the compilation listing file identifies any text obtained from an included file by placing a mnemonic in the first character position of the line on which the text appears. The “n” specifies that the text was either accessed from a source file or from a text library. The “I” tells you that the text was accessed with the %INCLUDE directive and *n* is a number that tells you the nesting level of the included text. See the *VAX BASIC User Manual* for more information on listing mnemonics.
9. **Including a File**  
If you do not specify a complete file specification, VAX BASIC uses the default device and directory and the file type BAS.

# **%INCLUDE**

## **10. Including a CDD Definition**

- In versions to VAX CDD/Plus Version 4.0, there are two types of CDD path names: *full* and *relative*. A full path name begins with CDD\$TOP and specifies the complete path to the record definition. A relative path name begins with any string other than CDD\$TOP and is appended to the current CDD\$DEFAULT.
- In VAX CDD/Plus Version 4.0 or higher, the pathnames described previously are known as DMU pathnames, as distinct from CDO pathnames. You can specify either a *full* DMU pathname, a *full* CDO pathname, or a *relative* pathname. A full pathname consists of a dictionary origin followed by a dictionary path. A full DMU pathname has CDD\$TOP as its origin. A full CDO pathname has an *anchor* as its origin. See CDD/Plus documentation for detailed information on pathnames.
- If the record definition being accessed is in a CDO-format dictionary, you can create a dependency relationship in the dictionary between a dictionary representation of your program and the record definitions that you include in the program. The dictionary representation of the program is called a compiled module entity.
- If you specify the /DEPENDENCY\_DATA qualifier to the compiler and your CDD\$DEFAULT points to a CDO-format dictionary, a compiled module entity is created for each compilation unit at compile time in CDD\$DEFAULT. No compiled module entity is created if both conditions are not true.
- If a compiled module entity exists for the program, an %INCLUDE %FROM %CDD directive specifying a record description in a CDO-format dictionary creates a relationship between the compiled module entity and the CDO-format record definition.
- If the record description specified in the pathname exists, it is copied to the program, whether a compiled module entity can be created or not.
- When you use the %INCLUDE directive to extract a record definition from the CDD, VAX BASIC translates the CDD definition to the syntax of the VAX BASIC RECORD statement.

## **%INCLUDE**

- You can use the `/SHOW:CDD_DEFINITIONS` qualifier to specify that translated CDD definitions (in `RECORD` statement syntax) are included in the compilation listing file. VAX BASIC places a “C” in column 1 when the translated `RECORD` statement appears in the listing file.
- When you specify `/SHOW:NOCDD_DEFINITIONS`, VAX BASIC does not include the CDD definition in the listing file. However, VAX BASIC still includes the names, data types, and offsets of the CDD record components in the program listing’s allocation map.
- See the *VAX BASIC User Manual* and the CDD/Plus documentation for more information on dictionary data definitions.

### **11. Including a File from a Text Library**

- The VAX BASIC compiler searches through the specified text library for the module named and compiles the module upon encountering the `%INCLUDE` directive.
- VAX BASIC allows only 16 text libraries to be opened at one time. Therefore, you cannot have `%INCLUDE` directives from a text library nested more than 16 levels deep. If you exceed this maximum, VAX BASIC signals an error message.
- If you do not specify a directory name and file type, VAX BASIC uses the default device and directory and the file type `TLB`.
- VAX BASIC provides the text library `BASIC$STARLET`. `BASIC$STARLET` contains condition codes and other symbols defined in the system object and shareable image libraries. Using the definitions from `BASIC$STARLET` allows you to reference condition codes and other system-defined symbols as local, rather than global symbols. To create your own text libraries using the VMS Librarian Utility, see the *VMS Librarian Utility Manual*.

---

## **Examples**

### **Example 1**

```
!Including a File
%INCLUDE "YESNO"
```

# **%INCLUDE**

## **Example 2**

```
!Including a CDD Definition  
%INCLUDE %FROM %CDD "CDD$TOP.EMPLOYEE"
```

## **Example 3**

```
!Including a CDD Definition with a CDO-format pathname  
%INCLUDE %FROM %CDD "MYNODE::MY$DISK:[MY_DIR]PERSONNEL.EMPLOYEE"  
!The anchor is MYNODE::MY$DISK:[MY_DIR]
```

## **Example 4**

```
!Including a File from a Text Library  
%INCLUDE "EOF_CHECK" %FROM %LIBRARY "SYS$LIBRARY:BASIC_LIB.TLB"
```



---

## **%LET**

The **%LET** directive declares and provides values for lexical variables. You can use lexical variables only in conditional expressions in the **%IF-%THEN-%ELSE** directive and in lexical expressions in subsequent **%LET** directives.

---

### **Format**

**%LET** *%lex-var = lex-exp*

---

### **Syntax Rules**

1. *Lex-var* is the name of a lexical variable. Lexical variables are always LONG integers.
2. *Lex-var* must be preceded by a percent sign (%) and cannot end with a dollar sign (\$) or percent sign.
3. *Lex-exp* can be any of the following:
  - A lexical variable named in a previous **%LET** directive.
  - An integer literal, with or without the percent sign suffix.
  - A lexical built-in function.
  - Any combination of the above, separated by valid lexical operators. Lexical operators can be logical operators, relational operators, and the arithmetic operators for addition (+), subtraction (−), multiplication (\*), and division (/).

## %LET

---

### Remarks

1. Only a line number or a comment field can appear on the same physical line as the %LET directive.
2. You cannot change the value of *lex-var* within a program unit once it has been named in a %LET directive. For more information on coding conventions, see the *VAX BASIC User Manual*.

---

### Example

```
%LET %DEBUG_ON = 1%
```

---

## **%LIST**

The **%LIST** directive causes the VAX BASIC compiler to start or resume accumulating compilation information for the program listing file.

---

### **Format**

**%LIST**

---

### **Syntax Rules**

None.

---

### **Remarks**

1. Only a line number or a comment field can appear on the same physical line as the **%LIST** directive.
2. The **%LIST** directive has no effect unless you requested a listing file. For more information on listing file format, see the *VAX BASIC User Manual*.
3. As soon as it encounters the **%LIST** directive, the VAX BASIC compiler starts or resumes accumulating information for the program listing file. Thus, the directive itself appears as the next line in the listing file.

---

### **Example**

```
%LIST
```

## **%NOCROSS**

---

## **%NOCROSS**

The `%NOCROSS` directive causes the VAX BASIC compiler to stop accumulating cross-reference information for the program listing file.

---

### **Format**

`%NOCROSS`

---

### **Syntax Rules**

None.

---

### **Remarks**

1. Only a line number or a comment field can appear on the same physical line as the `%NOCROSS` directive.
2. The VAX BASIC compiler stops accumulating cross-reference information for the program listing file immediately after encountering the `%NOCROSS` directive.
3. The `%NOCROSS` directive has no effect unless you request a listing file and cross-reference information.
4. DIGITAL recommends that you do not embed a `%NOCROSS` directive within a statement. Embedding a `%NOCROSS` directive within a statement makes the accumulation of cross-reference information unpredictable. For more information on listing file format, see the *VAX BASIC User Manual*.

---

**Example**

`%NOCROSS`

# **%NOLIST**

---

## **%NOLIST**

The **%NOLIST** directive causes the **VAX BASIC** compiler to stop accumulating compilation information for the program listing file.

---

### **Format**

**%NOLIST**

---

### **Syntax Rules**

None.

---

### **Remarks**

1. Only a line number or a comment field can appear on the same physical line as the **%NOLIST** directive.
2. As soon as it encounters the **%NOLIST** directive, the **VAX BASIC** compiler stops accumulating information for the program listing file. Thus, the directive itself does not appear in the listing file.
3. The **%NOLIST** directive has no effect unless you requested a listing file.
4. In **VAX BASIC**, you can override all **%NOLIST** directives in a program with the **/SHOW:OVERRIDE** qualifier. For more information on listing file format, see the *VAX BASIC User Manual*.

---

### **Example**

```
%NOLIST
```

---

## **%PAGE**

The `%PAGE` directive causes VAX BASIC to begin a new page in the program listing file immediately after the line that contains the `%PAGE` directive.

---

### **Format**

`%PAGE`

---

### **Syntax Rules**

None.

---

### **Remarks**

1. Only a line number or a comment field can appear on the same physical line as the `%PAGE` directive.
2. The `%PAGE` directive has no effect unless you request a listing file.

---

### **Example**

```
%PAGE
```

## %PRINT

---

## %PRINT

The %PRINT directive lets you insert a message into your source code that the VAX BASIC compiler prints during compilation.

---

### Format

**%PRINT** *str-lit*

---

### Syntax Rules

None.

---

### Remarks

1. Only a line number or a comment field can appear on the same physical line as the %PRINT directive.
  2. VAX BASIC will print the message specified as soon as it encounters a %PRINT directive. *Str-lit* is displayed on the terminal screen and in the compilation listing.
- 

### Example

```
%IF %DEBUG = 1% %THEN  
%PRINT "This is a debug compilation"
```

#### Output

```
%BASIC-S-USERPRINT, This is a debug compilation
```



---

## %REPORT

The %REPORT directive lets you record a dependency relationship between the compiled module entity for your program and the data definitions in CDD/Plus dictionaries. The data definitions are not copied into the program.

---

### Format

**%REPORT %DEPENDENCY** *str-lit* [ *relationship-type* ]

---

### Syntax Rules

1. *str-lit* specifies a path name in a CDO-format dictionary. It can be either a DMU-format pathname or a CDO-format pathname, enclosed in quotation marks. This specifies a dictionary entity, such as a form definition or an Rdb/VMS database, that the program references.
2. *relationship-type* specifies a valid CDD/Plus protocol; it must be enclosed in quotation marks if specified. The default *relationship-type* is CDD\$COMPILED\_DEPENDS\_ON.

---

### Remarks

1. For this directive to be meaningful, you must specify the /DEPENDENCY\_DATA qualifier at compile time. If /DEPENDENCY is not specified, the compiler will simply check syntax and otherwise ignore the %REPORT directive.
2. Your current CDD\$DEFAULT and *str-lit* must refer to CDO-format dictionaries (not necessarily the same one).

## **%REPORT**

3. If you specify the `/DEPENDENCY_DATA` qualifier to the compiler, and if `CDD$DEFAULT` points to a CDO-format dictionary, a compiled module entity is created in `CDD$DEFAULT` for each compilation unit. No compiled module entity is created if both conditions are not true.
4. The `%REPORT %DEPENDENCY` directive creates a dependency relationship in the dictionary between the compiled module entity for the program and the CDO-format dictionary entity to which it refers.

---

### **Example**

```
!Establish access to the form PINK_SLIP in a dictionary
!on a specified node, and report the program's dependency
!relationship with the form.
%REPORT %DEPENDENCY "MYNODE::MY$DISK:[MYDIR]PERSONNEL.FORMS.PINK_SLIP"
!Relationship is CDD$COMPILED_DEPENDS_ON, the default.
```

---

## **%SBTTL**

The **%SBTTL** directive lets you specify a subtitle for the program listing file.

---

### **Format**

**%SBTTL** *str-lit*

---

### **Syntax Rules**

*Str-lit* can contain up to 45 characters.

---

### **Remarks**

1. VAX BASIC truncates extra characters from *str-lit* and does not signal a warning or error.
2. Only a line number or a comment field can appear on the same physical line as the **%SBTTL** directive.
3. The specified subtitle appears underneath the title on the second line of all pages of source code in the listing file until the VAX BASIC compiler encounters another **%SBTTL** or **%TITLE** directive. VAX BASIC clears the subtitle field before the allocation map section of the listing is generated. This way, you only get a subtitle on the listing pages that contain source code.
4. Because VAX BASIC associates a subtitle with a title, a new **%TITLE** directive sets the current subtitle to the null string. In this case, no subtitle appears in the listing until VAX BASIC encounters another **%SBTTL** directive.

## %SBTTL

5. If you want a subtitle to appear on the first page of your listing, the %SBTTL directive should appear at the beginning of your program, immediately after the %TITLE directive. Otherwise, the subtitle will start to appear only on the second page of the listing.
6. If you want the subtitle to appear on the page of the listing that contains the %SBTTL directive, the %SBTTL directive should immediately follow a %PAGE directive or a %TITLE directive that follows a %PAGE directive.
7. The %SBTTL directive has no effect unless you request a listing file.

---

### Example

```
100    %TITLE "Learning to Program in VAX BASIC"
      %SBTTL "Using FOR-NEXT Loops"
      REM    THIS PROGRAM IS A SIMPLE TEST
200    DATA  1, 2, 3, 4
      .
      .
      .
      NEXT I%
300    END
```

### Output

```
TEST$MAIN                                Learning to Program in VAX BASIC
                                           Using FOR-NEXT Loops
      1          100    %TITLE "Learning to Program in VAX BASIC"
      2          %SBTTL "Using FOR-NEXT Loops"
      3          REM THIS PROGRAM IS A SIMPLE TEST
      4          200    DATA 1, 2, 3, 4
      .
      .
      .
     10          NEXT I%
     11          300    END
```

---

## **%TITLE**

The **%TITLE** directive lets you specify a title for the program listing file.

---

### **Format**

**%TITLE** *str-lit*

---

### **Syntax Rules**

*Str-lit* can contain up to 45 characters.

---

### **Remarks**

1. VAX BASIC truncates extra characters from *str-lit* and does not signal a warning or error.
2. Only a line number or a comment field can appear on the same physical line as the **%TITLE** directive.
3. The specified title appears on the first line of every page of the listing file until VAX BASIC encounters another **%TITLE** directive in the program.
4. The **%TITLE** directive should appear on the first line of your program, before the first statement, if you want the specified title to appear on the first page of your listing.
5. If you want the specified title to appear on the page that contains the **%TITLE** directive, the **%TITLE** directive should immediately follow a **%PAGE** directive.
6. Because VAX BASIC associates a subtitle with a title, a new **%TITLE** directive sets the current subtitle to the null string.
7. The **%TITLE** directive has no effect unless you request a listing file.

# %TITLE

---

## Example

```
100    %TITLE "Learning to Program in VAX BASIC"
      REM THIS PROGRAM IS A SIMPLE TEST
200    DATA 1, 2, 3, 4
      .
      .
      .
      NEXT I%
300    END
```

### Output

```
TEST$MAIN                                Learning to Program in VAX BASIC
  1          100    %TITLE "Learning to Program in VAX BASIC"
  2          %SBTTL "Using FOR-NEXT Loops"
  3          REM THIS PROGRAM IS A SIMPLE TEST
  4          200    DATA 1, 2, 3, 4
      .
      .
      .
  10         NEXT I%
  11        300    END
```

---

## **%VARIANT**

The **%VARIANT** directive is a built-in lexical function that allows you to conditionally control program compilation. **%VARIANT** returns an integer value when you reference it in a lexical expression. You set the variant value with the **/VARIANT** qualifier when you compile the program or with the **SET %VARIANT** command.

---

### **Format**

**%VARIANT**

---

### **Syntax Rules**

None.

---

### **Remarks**

1. The **%VARIANT** function can appear only in a lexical expression.
2. The **%VARIANT** function returns the integer value specified either with the **COMPILE /VARIANT** command, the **SET /VARIANT** command, or the DCL command **BASIC**. The returned integer always has a data type of **LONG**.

# %VARIANT

---

## Example

```
%LET %VMS = 0
%LET %RSX = 1
%LET %RSTS = 2

%IF %VARIANT = %VMS
  %THEN
    .
    .
    .
%ELSE %IF %VARIANT = %RSX OR %VARIANT = %RSTS
  %THEN
    .
    .
    .
    %ELSE %ABORT "Illegal compilation variant"
  %END %IF
%END %IF
```



# Statements and Functions

---

This chapter provides reference material on all of the VAX BASIC statements and functions. The statements and functions are listed in alphabetical order and each description contains the following sections:

<b>Definition</b>	A description of what the statement does.
<b>Format</b>	The required syntax for the statement.
<b>Syntax Rules</b>	Any rules governing the use of parameters, separators, or other syntax items.
<b>Remarks</b>	Explanatory remarks concerning the effect of the statement on program execution and any restrictions governing its use.
<b>Example</b>	One or more examples of the statement in a VAX BASIC program. Where appropriate, sample output is also shown.

# ABS

---

## ABS

The ABS function returns a floating-point number that equals the absolute value of a specified floating-point expression.

---

### Format

*real-var* = **ABS** (*real-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. The argument of the ABS function must be a real expression. When the argument is a real expression, VAX BASIC returns a value of the same floating-point size. When the argument is not a real expression, VAX BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.
2. The returned floating-point value is always greater than or equal to zero. The absolute value of 0 is zero. The absolute value of a positive number equals that number. The absolute value of a negative number equals that number multiplied by  $-1$ .

---

**Example**

```
G = 5.1273
A = ABS(-100 * G)
B = -39
PRINT ABS(B), A
```

**Output**

```
39          512.73
```

## ABS%

---

## ABS%

The **ABS%** function returns an integer that equals the absolute value of a specified integer expression.

---

### Format

*int-var* = **ABS%** (*int-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to an integer of the default integer size.
2. The returned value is always greater than or equal to zero. The absolute value of 0 is zero. The absolute value of a positive number equals that number. The absolute value of a negative number equals that number multiplied by  $-1$ .

---

**Example**

```
G% = 5.1273
A = ABS%(-100% * G%)
B = -39
PRINT ABS%(B), A
```

**Output**

```
39          512
```

# ASCII

---

## ASCII

The ASCII function returns the ASCII value in decimal of a string's first character.

---

### Format

$$int-var = \left\{ \begin{array}{l} \text{ASC} \\ \text{ASCII} \end{array} \right\} (str-exp)$$

---

### Syntax Rules

None.

---

### Remarks

1. The ASCII value of a null string is zero.
2. The ASCII function returns an integer value of the default size between 0 and 255.

---

### Example

```
DECLARE STRING time_out
time_out = "Friday"
PRINT ASCII(time_out)
```

#### Output

70

---

## ATN

The ATN function returns the arctangent (that is, angular value) of a specified tangent in radians or degrees.

---

### Format

*real-var* = ATN (*real-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. ATN returns a value from  $-\text{PI}/2$  through  $\text{PI}/2$ .
2. The returned angle is expressed in radians or degrees, depending on which angle clause you choose with the OPTION statement.
3. The argument of the ATN function must be a real expression. When the argument is a real expression, VAX BASIC returns a value of the same floating-point size. When the argument is not a real expression, VAX BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

# ATN

---

## Example

```
OPTION ANGLE = RADIANS
DECLARE SINGLE angle_rad, angle_deg, T
INPUT "Tangent value";T
angle_rad = ATN(T)
PRINT "The smallest angle with that tangent is" ;angle_rad; "radians"
angle_deg = angle_rad/(PI/180)
PRINT "and"; angle_deg; "degrees"
```

### Output

```
Tangent value? 2
The smallest angle with that tangent is 1.10715 radians
and 63.435 degrees
```



---

# BUFSIZ

The BUFSIZ function returns the record buffer size, in bytes, of a specified channel.

---

## Format

*int-var* = **BUFSIZ** (*chnl-exp*)

---

## Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number.
2. The value assigned to *int-var* is a LONG integer.

---

## Remarks

1. If the specified channel is closed, BUFSIZ returns a value of zero.
2. BUFSIZ of channel #0 always returns the value 132.

---

## Example

```
DECLARE LONG buffer_size
buffer_size = BUFSIZ(0)
PRINT "Buffer size equals";buffer_size
```

### Output

```
Buffer size equals 132
```

# CALL

---

## CALL

The CALL statement transfers control to a subprogram, external function, or other callable routine. You can pass arguments to the routine and can optionally specify passing mechanisms. When the called routine finishes executing, control returns to the calling program.

---

### Format

**CALL** *routine* [*pass-mech*] [ (*actual-param* ,... ) ]

*routine*:            { *sub-name*  
                          *any-callable-routine* }

*pass-mech*:        { **BY VALUE**  
                          **BY REF**  
                          **BY DESC** }

*actual-param*:    { *exp*  
                          *array* ( [,]... ) } [*pass-mech*]

---

### Syntax Rules

1. *Routine* is the name of a SUB subprogram or any other callable procedure, such as a system service or an RTL routine you want to call. It cannot be a variable name. See the VAX BASIC for more information on using system services, RTL routines, and other procedures.

2. *Pass-mech* specifies how arguments are passed to the called routine. If you do not specify a *pass-mech*, VAX BASIC passes arguments as indicated in Table 4–1.
- **BY VALUE** specifies that VAX BASIC passes the argument's 32-bit value.
  - **BY REF** specifies that VAX BASIC passes the argument's address. This is the default for all arguments except strings and entire arrays.
  - **BY DESC** specifies that VAX BASIC passes the address of a VAX BASIC descriptor. For information about the format of a VAX BASIC descriptor for strings and arrays, see Appendix C. For more information on other types of descriptors, see the *VAX Architecture Handbook*.

**Table 4–1: VAX BASIC Parameter-Passing Mechanisms**

<b>Parameter</b>	<b>BY VALUE</b>	<b>BY REF</b>	<b>BY DESC</b>
<b>Integer and Real Data</b>			
Variables	Yes	Yes <sup>1</sup>	Yes
Constants	Yes	Local copy <sup>1</sup>	Local copy
Expressions	Yes	Local copy <sup>1</sup>	Local copy
Elements of a nonvirtual array	Yes	Yes <sup>1</sup>	Yes
Virtual array elements	Yes	Local copy <sup>1</sup>	Local copy
Nonvirtual entire array	No	Yes	Yes <sup>1</sup>
Virtual entire array	No	No	No

<sup>1</sup>Specifies the default parameter-passing mechanism.

(continued on next page)

# CALL

**Table 4-1 (Cont.): VAX BASIC Parameter-Passing Mechanisms**

<b>Parameter</b>	<b>BY VALUE</b>	<b>BY REF</b>	<b>BY DESC</b>
<b>Packed Decimal Data</b>			
Variables	No	Yes <sup>1</sup>	Yes
Constants	No	Local copy <sup>1</sup>	Local copy
Expressions	No	Local copy <sup>1</sup>	Local copy
Nonvirtual array elements	No	Yes <sup>1</sup>	Yes
Virtual array elements	No	Local copy <sup>1</sup>	Local copy
Nonvirtual entire arrays	No	Yes	Yes <sup>1</sup>
Virtual entire arrays	No	No	No
<b>String Data</b>			
Variables	No	Yes	Yes <sup>1</sup>
Constants	No	Local copy	Local copy <sup>1</sup>
Expressions	No	Local copy	Local copy <sup>1</sup>
Nonvirtual array elements	No	Yes	Yes <sup>1</sup>
Virtual array elements	No	Local copy	Local copy <sup>1</sup>
Nonvirtual entire arrays	No	Yes	Yes <sup>1</sup>

<sup>1</sup>Specifies the default parameter-passing mechanism.

(continued on next page)

Table 4–1 (Cont.): VAX BASIC Parameter-Passing Mechanisms

Parameter	BY VALUE	BY REF	BY DESC
<b>String Data</b>			
Virtual entire arrays	No	No	No
<b>Other Parameters</b>			
RECORD variables	No	Yes <sup>1</sup>	No
RFA variables	No	Yes <sup>1</sup>	No

<sup>1</sup>Specifies the default parameter-passing mechanism.

3. You should use parameter-passing mechanisms only when calling non-BASIC routines or when a subprogram expects to receive a string or entire array by reference.
4. When *pass-mech* appears before the parameter list, it applies to all arguments passed to the called routine. You can override this passing mechanism by specifying a *pass-mech* for individual arguments in the *actual-param* list.
5. *Actual-param* lists the arguments to be passed to the called routine.
6. You can pass expressions or entire arrays. Optional commas in parentheses after the array name specify the dimensions of the array. The number of commas is equal to the number of dimensions – 1. Thus, no comma specifies a one-dimensional array, one comma specifies a two-dimensional array, two commas specify a three-dimensional array, and so on.

# CALL

7. You cannot pass entire virtual arrays.
8. The name of the routine can be from 1 to 31 characters and must conform to the following rules:
  - The first character of an unquoted name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs (\$), periods (.), or underscores (\_).
  - A quoted name can consist of any combination of printable ASCII characters.
9. VAX BASIC allows you to pass up to 255 parameters.

---

## Remarks

1. You can specify a null argument as an *actual-param* for non-BASIC routines by omitting the argument and the *pass-mech*, but not the commas or parentheses. This forces VAX BASIC to pass a null argument and allows you to access system routines from VAX BASIC.
2. Arguments in the *actual-param* list must agree in data type and number with the formal parameters specified in the subprogram.
3. An argument is modifiable when changes to it are evident in the calling program. Changing a modifiable parameter in a subprogram means the parameter is changed for the calling program as well. Variables and entire arrays passed by descriptor or by reference are modifiable.
4. An argument is nonmodifiable when changes to it are not evident in the calling program. Changing a nonmodifiable argument in a subprogram does not affect the value of that argument in the calling program. Arguments passed by value, constants, and expressions are nonmodifiable. Passing an argument as an expression (by placing it in parentheses) changes it from a modifiable to a nonmodifiable argument. Virtual array elements passed as parameters are non-modifiable.

# CALL

5. VAX BASIC will automatically convert numeric actual parameters to match the declared data type. If the actual parameter is a variable, VAX BASIC signals the informational message "Mode for parameter <n> of routine <name> changed to match declaration" and passes the argument by local copy. This prevents the called routine from modifying the contents of the variable.
6. For expressions and virtual array elements passed by reference, VAX BASIC makes a local copy of the value, and passes the address of this local copy. For dynamic string arrays, VAX BASIC passes a descriptor of the array of string descriptors. The compiler passes the address of the argument's actual value for all other arguments passed by reference.
7. Only BYTE, WORD, LONG, and SINGLE values can be passed by value. BYTE and WORD values passed by value are converted to LONG values.
8. If you attempt to call an external function, VAX BASIC treats the function as if it were invoked normally and validates all parameters. Note that you cannot call a STRING, HFLOAT, or RFA function. See the EXTERNAL statement for more information on how to invoke functions.

---

## Example

```
EXTERNAL SUB LIB$PUT_OUTPUT (string)
DECLARE STRING msg_str
msg_str = "Successful call to LIB$PUT_OUTPUT!"
CALL LIB$PUT_OUTPUT (msg_str)
```

### Output

```
Successful call to LIB$PUT_OUTPUT!
```

## CAUSE ERROR

---

## CAUSE ERROR

The **CAUSE ERROR** statement allows you to artificially generate a VAX BASIC run-time error and transfer program control to a VAX BASIC error handler.

---

### Format

**CAUSE ERROR** *err-num*

---

### Syntax Rules

*Err-num* must be a valid VAX BASIC run-time error number.

---

### Remarks

All error numbers are listed in the *VAX BASIC User Manual*.



---

## Example

```
WHEN ERROR IN
  .
  .
  .
CAUSE ERROR 11%
  .
  .
  .
USE
  SELECT ERR
    CASE = 11
      PRINT "End of file"
      CONTINUE
    CASE ELSE
      EXIT HANDLER
  END SELECT
END WHEN
```

# CCPOS

---

## CCPOS

The CCPOS function returns the current character or cursor position of the output record on a specified channel.

---

### Format

*int-var* = **CCPOS** (*chnl-exp*)

---

### Syntax Rules

*Chnl-exp* must specify an open file or terminal.

---

### Remarks

1. If *chnl-exp* is zero, CCPOS returns the current character position of the controlling terminal.
2. The *int-var* returned by the CCPOS function is of the default integer size.
3. The CCPOS function counts only characters. If you use cursor addressing sequences such as escape sequences, the value returned will not be the cursor position.
4. The first character position on a line is zero.

---

## Example

```
DECLARE LONG curs_pos  
PRINT "Hello";  
curs_pos = CCPOS (0)  
PRINT curs_pos
```

### Output

Hello 5

# CHAIN

---

## CHAIN

The CHAIN statement transfers control from the current program to another executable image. CHAIN closes all files, then requests that the new program begin execution. Control does not return to the original program when the new image finishes executing.

### NOTE

The CHAIN statement is not recommended for new program development. DIGITAL recommends that you use subprograms, external functions and pictures for program segmentation.

---

### Format

**CHAIN** *str-exp*

---

### Syntax Rules

*Str-exp* represents the file specification of the program to which control is passed. It can be a quoted or unquoted string.

---

### Remarks

1. *Str-exp* must refer to an executable image or VAX BASIC signals an error.
2. If you do not specify a file type, VAX BASIC searches for an EXE file type.
3. You cannot chain to a program on another node.
4. Execution starts at the beginning of the specified program.

5. Before chaining takes place, all active output buffers are written, all open files are closed, and all storage is released.
6. Because a CHAIN statement passes control from the executing image, the values of any program variables are lost. This means that you can pass parameters to a chained program only by using files or a system-specific feature such as LIB\$GET\_COMMON and LIB\$PUT\_COMMON.

---

## Example

```
DECLARE STRING time_out
time_out = "Friday"
PRINT ASCII(time_out)
CHAIN "CCPOS"
```

### Output

```
70
The current cursor position is 0
```

In this example, the executing image ASCII.EXE passes control to the chained program, CCPOS.EXE. The value that results from ASCII.EXE is 70. The second line of output reflects the value that results from CCPOS.EXE.

# CHANGE

---

## CHANGE

The **CHANGE** statement either converts a string of characters to their ASCII integer values or converts a list of numbers to a string of ASCII characters.

---

### Format

#### String Variable to Array

**CHANGE** *str-exp* **TO** *num-array-name*

#### Array to String Variable

**CHANGE** *num-array-name* **TO** *str-var*

---

### Syntax Rules

1. *Str-exp* is a string expression.
2. *Num-array-name* should be a one-dimensional array. If you specify a two-dimensional array, VAX BASIC converts only the first row of that array. VAX BASIC does not support conversion to or from arrays of more than two dimensions.

---

### Remarks

1. VAX BASIC does not support RECORD elements as a destination string or as a source or destination array for the **CHANGE** statement.
2. **String Variable to Array**
  - This format converts each character in the string to its ASCII value.
  - VAX BASIC assigns the value of the string's length to the first element of the array.

- VAX BASIC assigns the ASCII value of the first character in the string to the second element, ( 1 ) or (0,1), of the array, the ASCII value of the second character to the third element, ( 2 ) or (0,2), and so on.
- If the string is longer than the bounds of the array, VAX BASIC does not translate the excess characters, and signals the error “Subscript out of range” (ERR=55). The first element of array still contains the length of the string.

### 3. Array to String Variable

- This format converts the elements of the array to a string of characters.
- The length of the string is determined by the value in the zero element, ( 0 ) or (0,0), of the array. If the value of element zero is greater than the array bounds, VAX BASIC signals the error “Subscript out of range” (ERR=55).
- VAX BASIC changes the first element, ( 1 ) or (0,1), of array to its ASCII character equivalent, the second element, ( 2 ) or (0,2), to its ASCII equivalent, and so on. The length of the returned string is determined by the value in the zero element of the array. For example, if the array is dimensioned as ( 10 ), but the zero element ( 0 ) contains the value 5, VAX BASIC changes only elements ( 1 ), ( 2 ), ( 3 ), ( 4 ), and ( 5 ) to string characters.
- VAX BASIC truncates floating-point values to integers before converting them to characters.
- Values in array elements are treated modulo 256.

# CHANGE

---

## Example

```
DECLARE STRING ABCD, A
DIM INTEGER array_changes(6)
ABCD = "ABCD"
CHANGE ABCD TO array_changes
FOR I% = 0 TO 4
PRINT array_changes(I%)
NEXT I%
CHANGE array_changes TO A
PRINT A
```

### Output

```
4
65
66
67
68
ABCD
```



---

## CHR\$

The CHR\$ function returns a 1-character string that corresponds to the ASCII value you specify.

---

### Format

*str-var* = CHR\$ (*int-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. CHR\$ returns the character whose ASCII value equals *int-exp*. If *int-exp* is greater than 255, VAX BASIC treats it modulo 256. For example, CHR\$(325) is the same as CHR\$(69).
2. All arguments between 0 and 255 are considered unsigned 8-bit integers. For example, -1 is treated as 255.
3. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to an integer of the default size.

# CHR\$

---

## Example

```
DECLARE INTEGER num_exp
INPUT "Enter the ASCII value you wish to be converted";num_exp
PRINT "The equivalent character is ";CHR$(num_exp)
```

### Output

```
Enter the ASCII value you wish to be converted? 89
The equivalent character is Y
```

---

## CLOSE

The CLOSE statement ends I/O processing to a device or file on the specified channel.

---

### Format

**CLOSE** [**#**]chnl-exp,...

---

### Syntax Rules

*Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It can be preceded by an optional number sign (#).

---

### Remarks

1. VAX BASIC writes the contents of any active output buffers to the file or device before it closes that file or device.
2. Channel #0 (the controlling terminal) cannot be closed. An attempt to do so has no effect.
3. An error handler that is invoked by a GET statement cannot successfully close a file.
4. If you close a magnetic tape file that is open for output, VAX BASIC writes an end-of-file on the magnetic tape.
5. If you try to close a channel that is not currently open, VAX BASIC does not signal an error and the CLOSE statement has no effect.

# CLOSE

---

## Example

```
OPEN "COURSE_REC.DAT" FOR INPUT AS #2
INPUT #2, course_nam, course_num, course_desc, course_instr
.
.
.
CLOSE #2
```

In this example, COURSE\_REC.DAT is opened for input. After you have entered all of the required information, the file is closed.

---

## COMMON

The **COMMON** statement defines a named, shared storage area called a **COMMON** block or program section (PSECT). VAX BASIC program modules can access the values stored in the **COMMON** block by specifying a **COMMON** block with the same name.

---

### Format

```
{ COM
  COMMON } [ ( com-name ) ] {[data-type] com-item},...
```

*com-item*:

```
{
  num-unsubs-var
  num-array-name [(int-const1 TO) int-const2, ... ]
  str-unsubs-var = int-const
  str-array-name [(int-const1 TO) int-const2, ... ] [= int-const]
  record-var
  FILL [(int-const)] [= int-const]
  FILL% [(int-const)]
  FILL$ [(int-const)] [= int-const]
}
```

---

### Syntax Rules

1. A **COMMON** block can have the same name as a program variable.
2. A **COMMON** block and a map in the same program module cannot have the same name.
3. All **COMMON** elements must be separated with commas.
4. *Com-name* is optional. If you specify a *com-name*, it must be in parentheses. If you do not specify a *com-name*, the default is "\$BLANK".

## COMMON

5. *Com-name* can be from 1 through 31 characters. The first character of the name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs (\$), periods (.), or underscores (\_).
6. *Data-type* can be any VAX BASIC data type keyword or a data type defined in the RECORD statement. Data type keywords, size, range, and precision are listed in Table 1–2 in this manual.
7. When you specify a data type, all following *com-items*, including FILL items, are of that data type until you specify a new data type.
8. If you do not specify any data type, *com-items* take the current default data type and size.
9. *Com-item* declares the name and format of the data to be stored.
  - *Num-unsubs-var* and *num-array-name* specify a numeric variable or a numeric array.
  - *Record-var* specifies a record instance.
  - *Str-unsubs-var* and *str-array-name* specify a fixed-length string variable or array. You can specify the number of bytes to be reserved for the variable with the *=int-const* clause. The default string length is 16.
  - When you specify either a numeric or a string array, VAX BASIC allows you to declare both lower and upper bounds. The upper bound is required; the lower bound is optional.
    - *Int-const1* specifies the lower bounds of the array.
    - *Int-const2* specifies the upper bounds of the array and, when accompanied by *int-const1*, must be preceded by the keyword TO.
    - *Int-const1* must be less than or equal to *int-const2*.
    - If you do not specify *int-const1*, VAX BASIC uses zero as the default lower bound.
    - *Int-const1* and *int-const2* can be either negative or positive values.
  - The FILL, FILL%, and FILL\$ keywords allow you to reserve parts of the record buffer within or between data elements and to define the format of the storage. *Int-const* specifies the number of FILL items to be reserved. The *=int-const* clause allows you to specify the

number of bytes to be reserved for string FILL items. Table 4–2 describes FILL item format and storage allocation.

#### NOTE

In the applicable formats of FILL, (*int-const*) represents a repeat count, not an array subscript. FILL (*n*) represents *n* elements, not *n* + 1.

**Table 4–2: FILL Item Formats and Storage Allocations**

<b>FILL Format</b>	<b>Storage Allocation</b>
FILL	Allocates storage for one element of the default data type unless preceded by a <i>data-type</i> ; the number of bytes allocated depends on the default or the specified data type.
FILL( <i>int-const</i> )	Allocates storage for the number of floating-point elements specified by <i>int-const</i> unless preceded by a <i>data type</i> ; the number of bytes allocated for each element depends on the default floating-point data size or the specified <i>data type</i> .
FILL%	Allocates storage for one integer element; the number of bytes allocated depends on the default integer size.
FILL%( <i>int-const</i> )	Allocates storage for the number of integer elements specified by <i>int-const</i> ; the number of bytes allocated for each element depends on the default integer size.
FILL\$	Allocates 16 bytes of storage for a string element. The dollar sign can be omitted if the FILL keyword is preceded by the STRING data type.

(continued on next page)

# COMMON

**Table 4–2 (Cont.): FILL Item Formats and Storage Allocations**

<b>FILL Format</b>	<b>Storage Allocation</b>
FILL\$(int-const)	Allocates 16 bytes of storage for the number of string elements specified by <i>int-const</i> . The dollar sign can be omitted if the FILL keyword is preceded by the STRING data type.
FILL\$=int-const	Allocates the number of bytes of storage specified by <i>int-const</i> for a string element. The dollar sign can be omitted if the FILL keyword is preceded by the STRING data type.
FILL\$(int-const1)=int-const2	Allocates the number of bytes of storage specified by <i>int-const2</i> for the number of string elements specified by <i>int-const1</i> . The dollar sign can be omitted if the FILL keyword is preceded by the STRING data type.

## Remarks

1. Variables in a common are not initialized by VAX BASIC.
2. A COMMON area and a MAP area with the same name, in different program modules, specify the same storage area.
3. VAX BASIC does not execute COMMON statements. The COMMON statement allocates and defines the data storage area at compilation time.
4. When you link your program, the size of the COMMON area is the size of the largest COMMON area with that name. VAX BASIC concatenates COMMON statements with the same *com-name* within a single program module into a single PSECT. The total space allocated is the sum of the space allocated in the concatenated COMMON statements.



## COMMON

If you specify the same *com-name* in several program modules, the size of the PSECT will be determined by the program module that has the greatest amount of space allocated in the concatenated COMMON statements.

5. The COMMON statement must lexically precede any reference to variables declared in it.
6. A COMMON area can be accessed by more than one program module, as long as you define the *com-name* in each module that references the COMMON area.
7. Variable names in a COMMON statement in one program module need not match those in another program module.
8. Variables and arrays declared in a COMMON statement cannot be declared elsewhere in the program by any other declarative statements.
9. The data type specified for *com-items* or the default data type and size determines the amount of storage reserved in a COMMON block:
  - BYTE integers reserve 1 byte.
  - WORD integers reserve 2 bytes.
  - LONG integers reserve 4 bytes.
  - SINGLE floating-point numbers reserve 4 bytes.
  - DOUBLE floating-point numbers reserve 8 bytes.
  - GFLOAT floating-point numbers reserve 8 bytes.
  - HFLOAT floating-point numbers reserve 16 bytes.
  - DECIMAL(d,s) packed decimal numbers reserve  $(d+1)/2$  bytes.
  - STRING reserves 16 bytes (the default) or the number of bytes you specify with *=int-const*.
10. For multi-dimensional arrays, values are assigned in row-column order.

# COMMON

---

## Example

```
COMMON (sales_rec) DECIMAL net_sales (1965 TO 1975)      &
                  STRING row = 2,                          &
                      report_name = 24                    &
                  DOUBLE FILL,                             &
                  LONG part_bins
```

---

## **COMP%**

The **COMP%** function compares two numeric strings and returns a -1, 0, or 1, depending on the results of the comparison.

---

### **Format**

*int-var* = **COMP%** (*str-exp1*, *str-exp2*)

---

### **Syntax Rules**

*Str-exp1* and *str-exp2* are numeric strings. They must have one of the following formats:

- An optional minus sign (-), ASCII digits, and an optional decimal point (.)
- An optional minus sign, ASCII digits, an optional decimal point, the letter E, an optional minus sign, and a 2-digit exponent

---

### **Remarks**

1. If *str-exp1* is greater than *str-exp2*, **COMP%** returns a 1.
2. If the string expressions are equal, **COMP%** returns a 0.
3. If *str-exp1* is less than *str-exp2*, **COMP%** returns a -1.
4. The value returned by the **COMP%** function is an integer of the default size.

# COMP%

---

## Example

```
DECLARE STRING num_string, old_num_string, &  
            INTEGER result  
num_string = "-24.5"  
old_num_string = "33"  
result = COMP%(num_string, old_num_string)  
PRINT "The value is ";result
```

### Output

The value is -1

---

## CONTINUE

The **CONTINUE** statement causes VAX BASIC to clear an error condition and resume execution at the statement following the statement that caused the error or at the specified target.

---

### Format

**CONTINUE** [ *target* ]

---

### Syntax Rules

If you specify a target, it must be a label or line number that appears either inside the associated protected region, inside a **WHEN** block protected region that surrounds the current protected region, or in an unprotected region of code.

---

### Remarks

1. **CONTINUE** with no target causes VAX BASIC to transfer control to the statement immediately following the statement that caused the error. The next remark is an exception to this rule.
2. If an error occurs on a **FOR**, **NEXT**, **WHILE**, **UNTIL**, **SELECT** or **CASE** statement, control is transferred to the statement immediately following the corresponding **NEXT** or **END SELECT** statement, as in the following code.

# CONTINUE

```
10 WHEN ERROR IN
    A=10
    B=1
20   FOR I=A TO B STEP 2
30     GET #1
40     C=1
    NEXT I
50   C=0
    USE
    .
    .
    .
    CONTINUE
  END WHEN
```

If an error occurs on line 20, the CONTINUE statement transfers control to line 50. If an error occurs on line 30, program control resumes at line 40.

3. The CONTINUE statement must be lexically inside of a handler.
4. If you specify a CONTINUE statement within a detached handler, you cannot specify a target.

---

## Example

```
WHEN ERROR USE err_handler
.
.
.
END WHEN
.
.
.
HANDLER err_handler
  SELECT ERR
    CASE = 50
      PRINT "Insufficient data"
      CONTINUE
    CASE ELSE
      EXIT HANDLER
  END SELECT
END HANDLER
```

---

## COS

The COS function returns the cosine of an angle in radians or degrees.

---

### Format

*real-var* = COS (*real-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. The returned value is between  $-1$  and  $1$ . This value is expressed in either radians or degrees depending on which angle clause you choose with the OPTION statement.
2. VAX BASIC expects the argument of the COS function to be a real expression. When the argument is a real expression, VAX BASIC returns a value of the same floating-point size. When the argument is not a real expression, VAX BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

# COS

---

## Example

```
DECLARE SINGLE cos_value  
cos_value = 26  
PRINT COS(cos_value)
```

### Output

```
.646919
```



---

## CTRLC

The CTRLC function enables CTRL/C trapping. When CTRL/C trapping is enabled, a CTRL/C typed at the terminal causes control to be transferred to the error handler currently in effect.

---

### Format

*int-var* = CTRLC

---

### Syntax Rules

None.

---

### Remarks

1. When VAX BASIC encounters a CTRL/C, control passes to the error handler currently in effect. If there is no error handler in a program, the program aborts.
2. In a series of linked subprograms, setting CTRL/C for one subprogram enables CTRL/C trapping for all subprograms.
3. When you trap a CTRL/C with an error handler, your program may be in an inconsistent state; therefore, you should handle the CTRL/C error and exit the program as quickly as possible.
4. CTRL/C trapping is asynchronous; that is, VAX BASIC suspends execution and signals "Programmable ^C trap" (ERR=28) as soon as it detects a CTRL/C. Consequently, a statement can be interrupted while it is executing. A statement so interrupted may be only partially executed and variables may be left in an undefined state.

# CTRLC

5. VAX BASIC can trap more than one CTRL/C error in a program as long as the error does not occur while the error handler is executing. If a second CTRL/C is detected while the error handler is processing the first CTRL/C, the program aborts.
6. The CTRLC function always returns a value of zero.
7. The function RCTRLC disables CTRL/C trapping. See the description of the RCTRLC function for further details.

---

## Example

```
WHEN ERROR USE repair_work
Y% = CTRLC
.
.
.
END WHEN
HANDLER repair_work
IF (ERR=28) THEN PRINT "Interrupted by CTRLC!"
.
.
.
END HANDLER
```

---

**CVT\$\$**

The **CVT\$\$** function is a synonym for the **EDIT\$** function. See the **EDIT\$** function for more information.

**NOTE**

**DIGITAL** recommends that you use the **EDIT\$** function rather than the **CVT\$\$** function for new program development.

---

**Format**

*str-var* = **CVT\$\$** (*str-exp*, *int-exp*)

## CVTxx

---

## CVTxx

The CVT\$% function maps the first two characters of a string into a 16-bit integer. The CVT%\$ function translates a 16-bit integer into a 2-character string. The CVT\$F function maps a 4- or 8-character string into a floating-point variable. The CVTF\$ function translates a floating-point number into a 4- or 8-byte character string. The number of characters translated depends on whether the floating-point variable is single- or double-precision.

### NOTE

CVT functions are supported only for compatibility with BASIC-PLUS. DIGITAL recommends that you use VAX BASIC's dynamic mapping feature or multiple MAP statements for new program development.

---

### Format

*int-var* = CVT\$% (*str-var*)

*real-var* = CVT\$F (*str-var*)

*str-var* = CVT%\$ (*int-var*)

*str-var* = CVTF\$ (*real-var*)

---

### Syntax Rules

CVT functions reverse the order of the bytes when moving them to or from a string. Therefore, you can mix MAP and MOVE statements, but you cannot use FIELD and CVT functions on a file if you also plan to use MAP or MOVE statements.

---

## Remarks

### 1. CVT\$%

- If the CVT\$% *str-var* has fewer than two characters, VAX BASIC pads the string with nulls.
- If the default data type is LONG, only two bytes of data are extracted from *str-var*; the high-order byte is sign-extended into a longword.
- The value returned by the CVT\$% function is an integer of the default size.

### 2. CVT%\$

- Only two bytes of data are inserted into *str-var*.
- If you specify a floating-point variable for *int-var*, VAX BASIC truncates it to an integer of the default size. If the default size is BYTE and the value of *int-var* exceeds 127, VAX BASIC signals an error.

### 3. CVT\$F

- CVT\$F maps 4 characters when the program is compiled with /SINGLE and eight characters when the program is compiled with /DOUBLE.
- If *str-var* has fewer than four or eight characters, VAX BASIC pads the string with nulls.
- The *real-var* returned by the CVT\$F function is the default floating-point size. If the default size is GFLOAT or HFLOAT, VAX BASIC signals the error “Floating CVT illegal for GFLOAT or HFLOAT”.

# CVTxx

## 4. CVTF\$

- The CVTF\$ function maps single-precision numbers to a 4-character string and double-precision numbers to an 8-character string.
- VAX BASIC expects the argument of the CVTF\$ function to be a real expression. When the argument is a real expression, VAX BASIC returns a value of the same floating-point size. When the argument is not a real expression, VAX BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size. If the default floating-point size is GFLOAT or HFLOAT, VAX BASIC signals the error "Floating CVT illegal for GFLOAT or HFLOAT".

---

## Examples

### Example 1

```
DECLARE STRING test_string, another_string
DECLARE LONG first_number, next_number
test_string = "AT"
PRINT CVT$(test_string)
another_string = "at"
PRINT CVT$(another_string)
first_number = 16724
PRINT CVT$(first_number)
next_number = 24948
PRINT CVT$(next_number)
END
```

### Output

```
16724
24948
AT
at
```

**Example 2**

```
DECLARE STRING test_string, another_string
DECLARE SINGLE first_num, second_num
test_string = "DESK"
first_num = CVT$F(test_string)
PRINT first_num
another_string = "desk"
second_num = CVT$F(another_string)
PRINT second_num
PRINT CVTF$(first_num)
PRINT CVTF$(second_num)
END
```

```
$ BASIC/SINGLE CVTF
$ LINK CVTF
$ RUN CVTF
```

**Output**

```
.218256E+12
.466242E+31
DESK
desk
```

## DATA

---

## DATA

The DATA statement creates a data block for the READ statement.

---

### Format

**DATA**  $\left[ \begin{array}{l} \textit{num-lit} \\ \textit{str-lit} \\ \textit{unq-str} \end{array} \right], \dots$

---

### Syntax Rules

1. *Num-lit* specifies a numeric literal.
2. *Str-lit* is a character string that starts and ends with double or single quotation marks. The quotation marks must match.
3. *Unq-str* is a character sequence that does not start or end with double quotation marks and does not contain a comma.
4. Commas separate data elements. If a comma is part of a data item, the entire item must be enclosed in quotation marks.

---

### Remarks

1. Because VAX BASIC treats comment fields in DATA statements as part of the DATA sequence, you should not include comments.
2. A DATA statement must be the last or the only statement on a physical line.
3. DATA statements must end with a line terminator.



## DATA

4. When a DATA statement is continued with an ampersand (&), VAX BASIC interprets all characters between the keyword DATA and the ampersand as part of the data. Any code that appears on a noncontinued line is considered a new statement.
5. You cannot use the percent sign suffix for integer constants that appear in DATA statements. An attempt to do so causes VAX BASIC to signal the error, "Data format error" (ERR=50).
6. DATA statements are local to a program module.
7. VAX BASIC does not execute DATA statements. Instead, control is passed to the next executable statement.
8. A program can have more than one DATA statement. VAX BASIC assembles data from all DATA statements in a single program unit into a lexically ordered single data block.
9. VAX BASIC ignores leading and trailing blanks and tabs unless they are in a string literal.
10. Commas are the only valid data delimiters. You must use a quoted string literal if a comma is to be part of a string.
11. VAX BASIC ignores DATA statements without an accompanying READ statement.
12. VAX BASIC signals the error "Data format error" if the DATA item does not match the data type of the variable specified in the READ statement or if a data element that is to be read into an integer variable ends with a percent sign (%). If a string data element ends with a dollar sign (\$), VAX BASIC treats the dollar sign as part of the string.

---

### Example

```
10 DECLARE INTEGER A,B,C
   READ A,B,C
   DATA 1,2,3
   PRINT A + B + C
```

### Output

6

## DATE\$

---

## DATE\$

The DATE\$ function returns a string containing a day, month, and year in the form *dd-Mmm-yy*.

---

### Format

*str-var* = DATE\$ (*int-exp*)

---

### Syntax Rules

1. *Int-exp* can have up to six digits in the form *yyyddd*, where the characters *yyy* specify the number of years since 1970 and the characters *ddd* specify the day of that year.
2. You must fill all three of the *d* positions with digits or zeros before you can fill the *y* positions. For example:
  - DATE\$(121) returns the date 01-May-70, day 121 of the year 1970.
  - DATE\$(1201) returns the date 20-Jul-71, day 201 of the year 1971.
  - DATE\$(12001) returns the date 01-Jan-82, day one of the year 1982.
  - DATE\$(10202) returns the date 21-Jul-80, day 202 of the year 1980.

---

### Remarks

1. If *int-exp* equals zero, DATE\$ returns the current date.
2. The *str-var* returned by the DATE\$ function consists of nine characters and expresses the day, month, and year in the form *dd-Mmm-yy*.
3. If you specify an invalid date, such as day 385, results are unpredictable.

4. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to an integer of the default size.

---

## Example

```
DECLARE STRING todays_date  
todays_date = DATE$(0)  
PRINT todays_date
```

### Output

```
09-Oct-89
```

## DECIMAL

---

## DECIMAL

The **DECIMAL** function converts a numeric expression or numeric string to the **DECIMAL** data type.

---

### Format

*decimal-var* = **DECIMAL** (*exp* [, *int-const1*, *int-const2* ] )

---

### Syntax Rules

1. *Int-const1* specifies the total number of digits (the precision) and *int-const2* specifies the number of digits to the right of the decimal point (the scale). If you do not specify these values, VAX BASIC uses the *d* (digits) and *s* (scale) defaults for the **DECIMAL** data type.
2. *Int-const1* and *int-const2* must be positive integers from 1 through 31. *Int-const2* cannot exceed the value of *int-const1*.
3. *Exp* can be either numeric or numeric string. If a numeric string, it can contain the ASCII digits 0 through 9, uppercase E, a plus sign (+), a minus sign (-), and a period (.).

---

### Remarks

1. If *exp* is a string, VAX BASIC ignores leading and trailing spaces and tabs.
2. The **DECIMAL** function returns a zero when a string argument contains only spaces and tabs, or when it is null.

---

## Example

```
DECLARE STRING CONSTANT format_string = "##.###"  
DECLARE STRING num_value, DECIMAL(5,3) B  
INPUT "Enter a numeric value";num_value  
B = DECIMAL(num_value,5,3)  
PRINT USING format_string, B
```

### Output

```
Enter a numeric value? 6  
6.000
```

# DECLARE

---

## DECLARE

The DECLARE statement explicitly assigns a name and a data type to a variable, an entire array, a function, or a constant.

---

### Format

#### Variables

**DECLARE** *data-type* { *decl-item* [, [*data-type*] *decl-item* ] },...

#### DEF Functions

**DECLARE** *data-type* **FUNCTION** { *def-name* [ ( [ *def-param* ],... ) ] },...

#### Named Constants

**DECLARE** *data-type* **CONSTANT** { *const-name* = *const-exp* },...

*decl-item*:  $\left\{ \begin{array}{l} \textit{array-name} ( [ \textit{int-const1 TO } ] \textit{int-const2}, \dots ) \\ \textit{record-var} \\ \textit{unsubs-var} \end{array} \right\}$

*def-param*: *data-type*

---

## Syntax Rules

1. *Data-type* can be any VAX BASIC data type keyword or a data type defined by a RECORD statement. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.
2. **Variables**
  - *Decl-item* names an array, a record, or a variable.

# DECLARE

- A *decl-item* named in a DECLARE statement cannot be named in another DECLARE statement, or in a DEF, EXTERNAL, FUNCTION, SUB, COMMON, MAP, DIM, HANDLER, or PICTURE statement.
- Each *decl-item* is associated with the preceding data type. A data type is required for the first *decl-item*.
- *Decl-items* of data type STRING are dynamic strings.
- When you declare an array, VAX BASIC allows you to specify both lower and upper bounds for each dimension of the array. The upper bound is required; the lower bound is optional.
  - *Int-const1* specifies the lower bounds of the array.
  - *Int-const2* specifies the upper bounds of the array and, when accompanied by *int-const1*, must be preceded by the keyword TO.
  - *Int-const1* must be less than or equal to *int-const2*.
  - If you do not specify *int-const1*, VAX BASIC uses zero as the default lower bound.
  - *Int-const1* and *int-const2* can be any combination of negative or positive values or zero.

### 3. DEF Functions

- *Def-name* names the DEF function.
- *Data-type* specifies the data type of the value the function returns.
- *Def-params* specify the number and, optionally, the data type of the DEF parameters. Parameters define the arguments the DEF expects to receive when invoked.
  - When you specify a data type, all following parameters are of that data type until you specify a new data type.
  - If you do not specify any data type, parameters take the current default data type and size.
  - The number of parameters equals the number of commas plus 1. For example, empty parentheses specify one parameter of the default type and size; one comma inside the parentheses specifies two parameters of the default type and size; and so on. One data type inside the parentheses specifies one parameter of the specified data type; two data types separated by one comma specifies two parameters of the specified type, and so on.

# DECLARE

## 4. Named Constants

- *Const-name* is the name you assign to the constant.
- *Data-type* specifies the data type of the constant. The value of the *const* must be numeric if the data type is numeric and string if the data type is STRING. If the data type is STRING, *const* must be a quoted string or another string constant.
- *Const-exp* cannot be a datatype that was defined with the RECORD statement.
- String constants cannot exceed 498 characters.
- VAX BASIC allows *const-exp* to be an expression for all data types except DECIMAL. Expressions are not allowed as values when you name DECIMAL constants.
- Allowable operators in DECLARE CONSTANT expressions include all valid arithmetic, relational, and logical operators except exponentiation. Built-in functions cannot be used in DECLARE CONSTANT expressions. The following examples use valid expressions as values:

```
DECLARE DOUBLE CONSTANT max_value = (PI/2)
DECLARE STRING CONSTANT left_arrow = "<-----" + LF + CR
```

---

## Remarks

1. The DECLARE statement is not executable.
2. The DECLARE statement must lexically precede any reference to the variables, functions, or constants named in it.
3. To declare a virtual or run-time array, use the DIMENSION statement.
4. **Variables**
  - Subsequent *decl-items* are associated with the specified data type until you specify another data type.
  - All variables named in a DECLARE statement are initialized to zero if numeric or to the null string if string.



## 5. DEF Functions

- The DECLARE FUNCTION statement allows you to name a function defined in a DEF or DEF\* statement, specify the data type of the value the function returns, and declare the number and data type of the DEF parameters.
- Data type keywords must be separated by commas.
- The first specification of a data type for a *def-param* is the default for subsequent arguments until you specify another *def-param*. For example:

```
DECLARE DOUBLE FUNCTION interest(DOUBLE,SINGLE,,)
```

This example declares two parameters of the default type and size, one DOUBLE parameter, and three SINGLE parameters for the function named *interest*.

## 6. Named Constants

- The DECLARE CONSTANT statement allows you to name a constant value and assign a data type to that value. Note that you can specify only one data type in a DECLARE CONSTANT statement. To declare a constant of another data type, you must use a second DECLARE CONSTANT statement.
- During program execution, you cannot change the value assigned to the constant.
- The specified *data-type* determines the data type of the constant. For example:

```
DECLARE LONG CONSTANT True = -1, False = 0
DECLARE REAL CONSTANT ZZZ = 123.0
DECLARE BYTE CONSTANT YYY = '123'L
PRINT True, False, ZZZ, YYY
```

### Output

```
-1           0           123           123
```

In this example, VAX BASIC truncates the LONG value assigned to YYY to a BYTE variable.

# DECLARE

## NOTE

Data types specified in a DECLARE statement override any defaults specified in COMPILE command qualifiers or OPTION statements.

---

## Examples

### Example 1

```
!DEF Functions  
DECLARE INTEGER FUNCTION amount(,,DOUBLE,BYTE,,)
```

### Example 2

```
!Named Constants  
DECLARE DOUBLE CONSTANT interest_rate = 15.22
```

---

## DEF

The DEF statement lets you define a single- or multi-line function.

---

### Format

#### Single-line DEF

**DEF** [ *data-type* ] *def-name* [ ( [ *data-type* ] *var* ,... ) ] = *exp*

#### Multi-Line DEF

**DEF** [ *data-type* ] *def-name* [ ( [ *data-type* ] *var* ,... ) ] [ *statement* ]...  
[ *statement* ]...

{ **END DEF**  
**FNEND** } [ *exp* ]

---

## Syntax Rules

1. *Data-type* can be any VAX BASIC data type keyword or a data type defined in the RECORD statement. Data type keywords, size, range, and precision are listed in Table 1–2 in this manual.
2. The data type that precedes the *def-name* specifies the data type of the value returned by the DEF function.
3. *Def-name* is the name of the DEF function. The *def-name* can contain from 1 to 31 characters.
4. If the *def-name* also appears in a DECLARE FUNCTION statement, the following rules apply:
  - A function data type is required.

# DEF

- The first character of the *def-name* must be an alphabetic character (A through Z). The remaining characters can be any combination of letters, digits (0 through 9), dollar signs (\$), underscores (\_), or periods (.
5. If the *def-name* does not appear in a DECLARE FUNCTION statement, but the DEF statement appears before the first reference to the *def-name*, the following rules apply:
    - The function data type is optional.
    - The first character of the *def-name* must be an alphabetic letter (A through Z). The remaining characters can be any combination of letters, digits, dollar signs, underscores, or periods.
    - If a function data type is not specified, the last character in the *def-name* must be a percent sign for an INTEGER function, or a dollar sign for a STRING function.
  6. If the *def-name* does not appear in a DECLARE FUNCTION statement, and the DEF statement appears after the first reference to the *def-name*, the following rules apply:
    - The function data type cannot be present.
    - The first two characters of the *def-name* must be FN. The remaining characters can be any combination of letters, digits, dollar signs, underscores, or periods, with one restriction: the last character must be a percent sign for an INTEGER function, or a dollar sign for a STRING function.
    - There must be at least one character between the FN characters and the ending dollar sign or percent character. FN\$ and FN% are not valid function names.
  7. *Var* specifies optional formal DEF parameters. Because the parameters are local to the DEF function, any reference to these variables outside the DEF body creates a different variable.
  8. You can specify the data type of DEF parameters with a data type keyword or with a data type defined in a RECORD statement. If you do not include a data type, the parameters are of the default type and size. Parameters that follow a data type keyword are of the specified type and size until you specify another data type.
  9. You can specify up to 255 parameters in a DEF statement.

**10. Single-Line DEF**

*Exp* specifies the operations the function performs.

**11. Multi-Line DEF**

- *Statements* specifies the operations the function performs.
- The END DEF or FNEND statement is required to end a multi-line DEF.
- VAX BASIC does not allow you to specify any statements that indicate the beginning or end of any SUB, FUNCTION, PICTURE, HANDLER (attached handlers are legal), PROGRAM or DEF in a function definition.
- *Exp* specifies the function result. *Exp* must be compatible with the DEF data type.

---

**Remarks**

1. When VAX BASIC encounters a DEF statement, control of the program passes to the next executable statement after the DEF.
2. The function is invoked when you use the function name in an expression.
3. You cannot specify how parameters are passed. When you invoke a function, VAX BASIC evaluates parameters from left to right and passes parameters to the function so that they cannot be modified. Numeric parameters are passed by value and string parameters are passed by descriptor, where the descriptor points to a local copy. A DEF function can reference variables that are declared within the compilation unit in which the function resides, but it cannot reference variables in other DEF or DEF\* functions. A DEF function can, therefore, modify other variables in the program, but not variables within another DEF function.
4. A DEF function is local to the program, subprogram, function, or picture that defines it.
5. You can declare a DEF either by defining it, by using the DECLARE FUNCTION statement, or by implicitly declaring it with a reference to the function in an expression.

## DEF

6. If your program invokes a function with a name that does not start with FN before the DEF statement defines the function, VAX BASIC signals an error.
7. If the number of parameters, types of parameters, or type of result declared in the invocation disagree with the number or types of parameters defined in the DEF statement, VAX BASIC signals an error.
8. DATA statements in a multi-line DEF are not local to the function; they are local to the program module containing the function definition.
9. The function value is initialized to zero or the null string each time you invoke the function.
10. You can invoke a DEF function within an attached or detached handler.
11. DEF definitions cannot appear inside a protected region. However, a DEF can contain one or more protected regions.
12. In DEF definitions that contain handlers, the following rules apply:
  - If the function was invoked from a protected region, the EXIT HANDLER statement transfers control to the handler specified for that protected region.
  - If the function was not invoked from a protected region, the EXIT HANDLER statement transfers control to the default error handler.
13. If an exception is not handled within a DEF function, control is transferred to the module that invoked the DEF function.
14. ON ERROR statements within a DEF function are local to the function.
15. A CONTINUE, GOTO, GOSUB, ON ERROR GOTO, or RESUME statement in a multi-line function definition must refer to a line number or label in the same function definition.
16. You cannot transfer control into a multi-line DEF except by invoking the function.
17. DEF functions can be recursive. However, VAX BASIC does not detect infinitely recursive DEF functions during compilation.

---

## Examples

### Example 1

```
!Single-Line DEF
DEF DOUBLE add (DOUBLE A, B, SINGLE C, D, E) = A + B + C + D + E
INPUT 'Enter five numbers to be added';V,W,X,Y,Z
PRINT 'The sum is';ADD(V,W,X,Y,Z)
```

### Output

```
Enter five numbers to be added? 1,2,3,4,5
The sum is 15
```

### Example 2

```
PROGRAM I_want_a_raise
    OPTION TYPE = EXPLICIT,
            CONSTANT TYPE = DECIMAL,
            SIZE = DECIMAL (6,2)
    DECLARE DECIMAL CONSTANT Overtime_factor = 0.50
    DECLARE DECIMAL My_hours, My_rate, Overtime
    DECLARE DECIMAL FUNCTION Calculate_pay (DECIMAL,DECIMAL)

    INPUT "Your hours this week";My_hours
    INPUT "Your hourly rate";My_rate

    PRINT "My pay this week is"; Calculate_pay ( My_hours, My_rate )

    DEF DECIMAL Calculate_pay (DECIMAL Hours, Rate)
        IF Hours = 0.0
        THEN
            EXIT DEF 0.0
        END IF

        Overtime = Hours - 40.0

        IF Overtime < 0.0
        THEN
            Overtime = 0.0
        END IF

        END DEF (Hours * Rate) + (Overtime * (Overtime_factor * Rate) )

    END PROGRAM
```

# DEF

## Output

Your hours this week? 45.7

Your pay rate? 20.35

Your pay for the week is 987.95



---

**DEF\***

The DEF\* statement lets you define a single- or multi-line function.

**NOTE**

The DEF\* statement is not recommended for new program development. DIGITAL recommends that you use the DEF statement for defining single- and multi-line functions.

---

**Format**
**Single-line DEF\***

**DEF\*** [ *data-type* ] *def-name* [ ( [ *data-type* ] *var* , ... ) ] = *exp*

**Multi-Line DEF\***

**DEF\*** [ *data-type* ] *def-name* [ ( [ *data-type* ] *var* , ... ) ] [ *statement* ]...  
[ *statement* ]...

{ **END DEF** } [ *exp* ]  
{ **FNEND** }

---

**Syntax Rules**

1. *Data-type* can be any VAX BASIC data type keyword or a data type defined in the RECORD statement. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.
2. The data type that precedes the *def-name* specifies the data type of the value returned by the DEF\* function.
3. *Def-name* is the name of the DEF\* function. The *def-name* can contain from 1 to 31 characters.

## DEF\*

4. If the *def-name* also appears in a DECLARE FUNCTION statement, the following rules apply:
  - A function data type is required.
  - The first character of the *def-name* must be an alphabetic character (A through Z). The remaining characters can be any combination of letters, digits (0 through 9), dollar signs (\$), underscores (\_), or periods (.).
5. If the *def-name* does not appear in a DECLARE FUNCTION statement, but the DEF\* statement appears before the first reference to the *def-name*, the following rules apply:
  - The function data type is optional.
  - The first character of the *def-name* must be an alphabetic character (A through Z). The remaining characters can be any combination of letters, digits, dollar signs, underscores, or periods.
  - If a function data type is not specified, the last character in the *def-name* must be a percent sign for an INTEGER function, or a dollar sign for a STRING function.
6. If the *def-name* does not appear in a DECLARE FUNCTION statement, and the DEF\* statement appears after the first reference to the *def-name*, the following rules apply:
  - The function data type cannot be present.
  - The first two characters of the *def-name* must be FN. The remaining characters can be any combination of letters, digits, dollar signs, underscores, or periods, with one restriction: the last character must be a percent sign for an INTEGER function, or a dollar sign for a STRING function.
  - There must be at least one character between the FN characters and the ending dollar sign or percent character. FN\$ and FN% are not valid function names.
7. *Var* specifies optional formal function parameters.
8. You can specify the data type of function parameters with a data type keyword. If you do not specify a data type, parameters are of the default type and size. Parameters that follow a data type are of the specified type and size until you specify another data type.
9. You can specify up to 8 parameters in a DEF\* statement.

**10. Single-Line DEF\***

*Exp* specifies the operations the function performs.

**11. Multi-Line DEF\***

- *Statements* specifies the operations the function performs.
- The END DEF or FNEND statement is required to end a multi-line DEF\*.
- VAX BASIC does not allow you to specify any statements that indicate the beginning or end of any SUB, FUNCTION, PICTURE, HANDLER, PROGRAM or DEF in a function definition.
- *Exp* specifies the function result. *Exp* must be compatible with the DEF data type.

---

**Remarks**

1. When VAX BASIC encounters a DEF\* statement, control of the program passes to the next executable statement after the DEF\*.
2. A function defined by the DEF\* statement is invoked when you use the function name in an expression.
3. You cannot specify how parameters are passed. When you invoke a DEF\* function, VAX BASIC evaluates parameters from left to right and passes parameters to the function so that they cannot be modified. Numeric parameters are passed by value, and string parameters are passed by descriptor, where the descriptor points to a local copy. A DEF\* function can reference variables in the program unit where the function is declared, but it cannot reference variables in other DEF or DEF\* functions. A DEF\* function can, therefore, modify variables in its program unit, but not variables within another DEF\* function.
4. The following differences exist between DEF\* and DEF statements:
  - You can use the GOTO, ON GOTO, GOSUB, and ON GOSUB statements to a branch outside a multi-line DEF\*, but they are not recommended.

## DEF\*

- Although other variables used within the body of a DEF\* function are not local to the DEF\* function, DEF\* formal parameters are. However, if you change the value of formal parameters within a DEF\* function and then transfer control out of the DEF\* function without executing the END DEF or FNEND statement, variables outside the DEF\* that have the same names as DEF\* formal parameters are also changed.
  - You can pass up to 255 parameters to a DEF function. DEF\* functions accept a maximum of 8 parameters.
  - A DEF\* function value is not initialized when the DEF\* function is invoked. Therefore, if a DEF\* function is invoked, and no new function value is assigned, the DEF\* function returns the value of its previous invocation.
  - The error handler of the program module that contains the DEF\* is the default error handler for a DEF\* function. Parameters return to their original values when control passes to the error handler.
5. A DEF\* is local to the program unit or subprogram that defines it.
  6. You can declare a DEF\* either by defining it, by using the DECLARE FUNCTION statement, or by implicitly declaring it with a reference to the function in an expression.
  7. If the number of parameters, types of parameters, or type of result declared in the invocation disagree with the number or types of parameters defined in the DEF\* statement, VAX BASIC signals an error.
  8. DEF\* functions can be recursive.
  9. DATA statements in a multi-line DEF\* are not local to the function; they are local to the program module containing the function definition.
  10. You can invoke a DEF\* function within an attached or detached handler.
  11. DEF\* definitions cannot appear inside a protected region, but they can contain one or more protected regions.

12. In DEF\* functions that contain handlers, the following rules apply:
- If the function was invoked from a protected region, the EXIT HANDLER statement transfers control to the handler specified for that protected region.
  - If the function was not invoked from a protected region, the EXIT HANDLER statement transfers control to the default error handler.

## Examples

### Example 1

```
!Single-Line DEF*
DEF* STRING CONCAT (STRING A,B) = A + B
DECLARE STRING word1,word2
INPUT "Enter two words";word1,word2
PRINT CONCAT (word1,word2)
```

### Output

```
Enter two words? TO
? DAY
TODAY
```

### Example 2

```
!Multi-Line DEF*
DEF* DOUBLE example (DOUBLE A, B, SINGLE C, D, E)
    EXIT DEF IF B = 0
    example = (A/B) + C - (D*E)
END DEF
INPUT "Enter 5 numbers";V,W,X,Y,Z
PRINT example (V,W,X,Y,Z)
```

### Output

```
Enter 5 numbers? 2,4,6,8,1
-1.5
```

# DELETE

---

## DELETE

The DELETE statement removes a record from a relative or indexed file.

---

### Format

DELETE #*chnl-exp*

---

### Syntax Rules

*Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).

---

### Remarks

1. The DELETE statement removes the current record from a file. Once the record is removed, you cannot access it.
2. The file specified by *chnl-exp* must have been opened with ACCESS MODIFY or WRITE.
3. You can delete a record only if the last I/O statement executed on the specified channel was a successful GET or FIND operation.
4. The DELETE statement leaves the current record pointer undefined and the next record pointer unchanged.
5. VAX BASIC signals an error when the I/O channel is illegal or not open, when no current record exists, when access is illegal or illogical, or when the operation is illegal.

---

## Example

```
DECLARE STRING record_num
.
.
.
OPEN "CUS.DAT" FOR INPUT AS #1, RELATIVE FIXED      &
      ACCESS MODIFY, RECORDSIZE 40
.
.
.
INPUT "WHICH RECORD WOULD YOU LIKE TO EXAMINE";record_num
GET #1, RECORD record_num
DELETE #1
.
.
.
```

In this example, CUS.DAT is opened for input with ACCESS MODIFY. Once you enter the number of the record you want to retrieve and the GET statement executes successfully, the current record number is deleted.

## DET

---

## DET

The DET function returns the value of the determinant of the last matrix inverted with the MAT INV function.

---

### Format

*real-var* = DET

---

### Syntax Rules

None.

---

### Remarks

1. When a matrix is inverted with the MAT INV statement, VAX BASIC calculates the determinant as a by-product of the inversion process. The DET function retrieves this value.
2. If your program does not contain a MAT INV statement, the DET function returns a value of zero.
3. The value returned by the DET function is a floating-point value of the default size.



---

**Example**

```
MAT INPUT first_array(3,3)
MAT PRINT first_array;
PRINT
MAT inv_array = INV (first_array)
determinant = DET
MAT PRINT inv_array;
PRINT
PRINT determinant
PRINT
MAT mult_array = first_array * inv_array
MAT PRINT mult_array;
```

**Output**

```
? 1,0,0,0,1,0,0,0,1
  1 0 0
  0 1 0
  0 0 1

  1 0 0
  0 1 0
  0 0 1

  1

  1 0 0
  0 1 0
  0 0 1
```

## DIF\$

---

## DIF\$

The DIF\$ function returns a numeric string whose value is the difference between two numeric strings.

---

### Format

*str-var* = DIF\$ (*str-exp1*, *str-exp2*)

---

### Syntax Rules

Each *str-exp* can contain up to 54 ASCII digits, an optional decimal point, and an optional leading sign.

---

### Remarks

1. The DIF\$ function does not support E-format notation.
2. VAX BASIC subtracts *str-exp2* from *str-exp1* and stores the result in *str-var*.
3. The difference between two integers takes the precision of the larger integer.
4. The difference between two decimal fractions takes the precision of the more precise fraction, unless trailing zeros generate that precision.

5. The difference between two floating-point numbers takes precision as follows:
  - The difference of the integer parts takes the precision of the larger part.
  - The difference of the decimal fraction part takes the precision of the more precise part.
6. VAX BASIC truncates leading and trailing zeros.

---

## Example

```
PRINT DIF$ ("689", "-231")
```

### Output

```
920
```

## DIMENSION

---

# DIMENSION

The DIMENSION statement creates and names a static, dynamic, or virtual array. The array subscripts determine the dimensions and the size of the array. You can specify the data type of the array and associate the array with an I/O channel.

---

### Format

#### Nonvirtual, Nonexecutable

$$\left\{ \begin{array}{l} \text{DIM} \\ \text{DIMENSION} \end{array} \right\} \{ [ \text{data-type} ] \text{array-name} ( [ \text{int-const1 TO } ] \\ \text{int-const2, ... } ), \dots )$$

#### Executable

$$\left\{ \begin{array}{l} \text{DIM} \\ \text{DIMENSION} \end{array} \right\} \{ [ \text{data-type} ] \text{array-name} \\ ( [ \text{int-var1 TO } ] \text{int-var2, ... } ) \}, \dots$$

#### Virtual

$$\left\{ \begin{array}{l} \text{DIM} \\ \text{DIMENSION} \end{array} \right\} \{ \# \text{chnl-exp}, \{ [ \text{data-type} ] \text{array-name} \\ ( \text{int-const, ... } ) [ = \text{int-const} ] \}, \dots$$

---

## Syntax Rules

1. An array name in a DIM statement cannot also appear in a COMMON, MAP, or DECLARE statement.
2. *Data-type* can be any VAX BASIC data type keyword or a data type defined in a RECORD statement. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.

## DIMENSION

3. If you do specify a data type and the array name ends in a percent sign (%) or dollar sign (\$) suffix character, the variable must be a string or integer data type.
4. If you do not specify a data type, the array name determines the type of data the array holds. If the array name ends in a percent sign, the array stores integer data of the default integer size. If the array name ends in a dollar sign, the array stores string data. Otherwise, the array stores data of the default type and size.
5. An array can have up to 32 dimensions. Nonvirtual array sizes are limited by the virtual memory limits of your system.
6. When you declare a nonvirtual array, VAX BASIC allows you to specify both lower and upper bounds. The upper bound is required; the lower bound is optional.
  - *Int-const1* or *int-var1* specifies the lower bounds of the array.
  - *Int-const2* or *int-var2* specifies the upper bounds of the array and, when accompanied by *int-const1* or *int-var1*, must be preceded by the keyword TO.
  - *Int-const1* must be less than or equal to *int-const2*. *Int-var1* must be less than or equal to *int-var2*.
  - If you do not specify *int-const1* or *int-var1*, VAX BASIC uses zero as the default lower bound.
  - Array dimensions can have either positive or negative values.
7. **Nonvirtual, Nonexecutable**
  - When all the dimension specifications are integer constants, as in DIM A(15,10,20), the DIM statement is nonexecutable and the array size is static. A static array cannot appear in another DIM statement because VAX BASIC determines storage requirements at compilation time.
  - A nonexecutable DIM statement must lexically precede any reference to the array it dimensions. That is, you must dimension a static array before you can reference array elements.
8. **Virtual**
  - The virtual array must be dimensioned and the file must be open before you can reference the array.

## DIMENSION

- When the data type is `STRING`, the `=int-const` clause specifies the length of each array element. The default string length is 16 characters. Virtual string array lengths are rounded to the next higher power of 2. Therefore, specifying an element length of 12 results in an actual length of 16. For example:

```
DIM #1, STRING vir_array(100) = 12
OPEN "STATS.BAS" FOR OUTPUT as #1, VIRTUAL
```

### Output

```
%BASIC-W-STRLENINC, virtual array string VIR_ARRAY length increased
from 12 to 16
```

### 9. Executable

When any of the dimension specifications are integer variables as in `DIM A(10%,20%,Y%)`, the `DIM` statement is executable and the array is dynamic. A dynamic array can be redimensioned with a `DIM` statement any number of times because `VAX BASIC` allocates storage at run time when each `DIM` statement is executed.

---

## Remarks

1. You can create an array implicitly by referencing an array element without using a `DIM` statement. This causes `VAX BASIC` to create an array with dimensions of (10), (10,10), (10,10,10), and so on, depending on the number of bounds specifications in the referenced array element. You cannot create virtual or executable arrays implicitly.
2. `VAX BASIC` allocates storage for arrays by row, from right to left.
3. **Nonvirtual, Nonexecutable**
  - You can declare arrays with the `COMMON`, `MAP`, and `DECLARE` statements. Arrays so declared cannot be redimensioned with the `DIM` statement. Furthermore, string arrays declared with a `COMMON` or `MAP` statement are always fixed-length.
  - If you reference an array element declared in an array whose subscripts are smaller than the lower bound or larger than the upper bound specified in the `DIM` statement, `VAX BASIC` signals the error "Subscript out of range" (ERR=55).

## 4. Virtual

- For new development, DIGITAL does not recommend virtual arrays.
- When the rightmost subscript varies faster than the subscripts to the left, fewer disk accesses are necessary to access array elements in virtual arrays.
- Using the same DIM statement for multiple virtual arrays allocates all arrays in a single disk file. The arrays are stored in the order they were declared.
- Any program or subprogram can access a virtual array by declaring it in a virtual DIMENSION statement. For example:

```
DIM #1, A(10)
DIM #1, B(10)
```

In this example, array *B* overlays array *A*. You must specify the same channel number, data types, and limits in the same order as they occur in the DIM statement that created the virtual array.

- VAX BASIC stores a string in a virtual array by padding it with trailing nulls to the length of the array element. It removes these nulls when it retrieves the string from the virtual array. Remember that string array element sizes are always rounded to the next power of 2.
- The OPEN statement for a virtual array must include the ORGANIZATION VIRTUAL clause for the channel specified in the DIMENSION statement.
- VAX BASIC does not initialize virtual arrays and treats them as statically allocated arrays. You cannot redimension virtual arrays.
- Refer to the *VAX BASIC User Manual* for more information on virtual arrays.

## 5. Executable

- You create an executable, dynamic array by using integer variables for array bounds, as in DIM A(Y%,X%). This eliminates the need to dimension an array to its largest possible size. Array bounds in an executable DIM statement can be constants or variables, but not expressions. At least one bound must be a variable.
- You cannot reference an array named in an executable DIM statement until after the DIM statement executes.

# DIMENSION

- You can redimension a dynamic array to make the bounds of each dimension larger or smaller, but you cannot change the number of dimensions. For example, you cannot redimension a four-dimensional array to be a five-dimensional array.
- The executable DIM statement cannot be used to dimension virtual arrays, arrays received as formal parameters, or arrays declared in COMMON, MAP, or nonexecutable DIM statements.
- An executable DIM statement always reinitializes the array to zero (for numeric arrays) or to the null string if string.
- If you reference an array element declared in an executable DIM statement whose subscripts are not within the bounds specified in the last execution of the DIM, VAX BASIC signals the error "Subscript out of range" (ERR=55).

---

## Examples

### Example 1

```
!Nonvirtual, Nonexecutable  
DIM STRING name_list(20 TO 100), BYTE age(100)
```

### Example 2

```
!Virtual  
DIM #1%, STRING name_list(500), REAL amount(10,10)
```

### Example 3

```
!Executable  
DIM DOUBLE inventory (base,markup)  
.  
.  
.  
DIM DOUBLE inventory (new_base,new_markup)
```



---

## ECHO

The ECHO function causes characters to be echoed at a terminal that is opened on a specified channel.

---

### Format

*int-var* = ECHO (*chnl-exp*)

---

### Syntax Rules

*Chnl-exp* must specify a terminal.

---

### Remarks

1. The ECHO function is the complement of the NOECHO function; each function disables the effect of the other.
2. The ECHO function has no effect on an unopened channel.
3. The ECHO function always returns a value of zero.

# ECHO

---

## Example

```
DECLARE INTEGER Y,                                &
        STRING pass_word
Y = NOECHO(0%)
SET NO PROMPT
INPUT "Enter your password: ";pass_word
Y = ECHO(0%)
IF pass_word = "Darlene"
THEN
    PRINT CR+LF+"YOU ARE CORRECT !"
END IF
```

### Output

```
Enter your password?
YOU ARE CORRECT !
```

---

## EDIT\$

The EDIT\$ function performs one or more string editing functions, depending on the value of its integer argument.

---

### Format

*str-var* = EDIT\$ (*str-exp*, *int-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. VAX BASIC edits *str-exp* to produce *str-var*.
2. The editing that VAX BASIC performs depends on the value of *int-exp*. Table 4-3 describes EDIT\$ values and functions.
3. All values are additive; for example, you can perform the editing functions of values 8, 16, and 32 by specifying a value of 56.
4. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to an integer of the default size.

# EDIT\$

**Table 4-3: EDIT\$ Values**

<b>Value</b>	<b>Edit Performed</b>
1	Discards each character's parity bit (bit 7)
2	Discards all spaces and tabs
4	Discards all carriage returns <CR>, line feeds <LF>, form feeds <FF>, deletes <DEL>, escapes <ESC>, and nulls <NUL>
8	Discards leading spaces and tabs
16	Converts multiple spaces and tabs to a single space
32	Converts lowercase letters to uppercase letters
64	Converts left bracket ([) to left parenthesis [(] and right bracket (]) to right parenthesis [)]
128	Discards trailing spaces and tabs (same as TRM\$ function)
256	Suppresses all editing for characters within quotation marks; if the string has only one quotation mark, VAX BASIC suppresses all editing for the characters following the quotation mark

---

## Example

```
DECLARE STRING old_string, new_string
old_string = "a value of 32 converts lowercase letters to uppercase"
new_string = EDIT$(old_string,32)
PRINT new_string
```

### Output

```
A VALUE OF 32 CONVERTS LOWERCASE LETTERS TO UPPERCASE
```

**END**

---

## **END**

The **END** statement marks the physical and logical end of a main program, a program module, or a block of statements.

---

### **Format**

**END** [*block*]

*block*: {  
DEF[ *exp* ]  
FUNCTION[ *exp* ]  
GROUP  
RECORD  
VARIANT  
IF  
HANDLER  
PICTURE  
PROGRAM[ *int-exp* ]  
SELECT  
WHEN  
SUB  
}

---

### **Syntax Rules**

None.

# END

---

## Remarks

1. The **END** statement with no *block* keyword marks the end of a main program. The **END** or **END PROGRAM** statement must be the last statement on the last lexical line of the main program.
2. The **END** statement followed by a *block* keyword marks the end of a program, a **VAX BASIC SUB**, **FUNCTION**, or **PICTURE** subprogram, a **DEF**, an **IF**, a **HANDLER**, a **PROGRAM**, a **SELECT** statement block or a **WHEN** block.
3. **END RECORD**, **END GROUP**, and **END VARIANT** mark the end of a **RECORD** statement, or a **GROUP** component or **VARIANT** component of a **RECORD** statement.
4. **END DEF and END FUNCTION**
  - When **VAX BASIC** executes an **END DEF** or an **END FUNCTION** statement, it returns the function value to the statement that invoked the function and releases all storage associated with the **DEF** or **FUNCTION**.
  - If you specify an optional expression with the **END DEF** or **END FUNCTION** statement, the expression must be compatible with the **DEF** or **FUNCTION** data type. The expression is the function result unless an **EXIT DEF** or **EXIT FUNCTION** statement is executed. This expression supersedes all function assignments.
  - The **END DEF** statement restores the error handler in effect when the **DEF** was invoked (this is not true of the **DEF\*** statement).
  - The **END FUNCTION** statement does not affect **I/O** operations or files.
5. **END HANDLER**

The **END HANDLER** statement causes **VAX BASIC** to transfer control to the statement following the **WHEN** block with the exception cleared.
6. **END PROGRAM**
  - The **END PROGRAM** statement allows you to end a program module.

- An optional integer expression specifies the exit status of the program that is reported to DCL. This status is overridden by a status expression in an EXIT PROGRAM statement.
- You can specify an END PROGRAM statement without a matching PROGRAM statement.

**7. END WHEN**

- The END WHEN statement ends a WHEN block and transfers control to the statement following the WHEN block.
- If the END WHEN statement ends an attached handler, control is transferred to the statement following the WHEN block with the exception cleared.

**8. END SUB**

- The END SUB statement does not affect I/O operations or files.
- The END SUB statement releases the storage allocated to local variables and returns control to the calling program.
- The END SUB statement cannot be executed in an error handler unless the END SUB is in a subprogram called by the error handler of another routine.

9. When an END or END PROGRAM statement marking the end of a main program executes, VAX BASIC closes all files and releases all program storage.

10. If you use ON ERROR error handling, you must clear any errors with the RESUME statement before executing an END PROGRAM, END SUB, END FUNCTION or END PICTURE statement.

11. Except for the END PROGRAM statement, VAX BASIC signals an error when a program contains an END *block* statement with no corresponding and preceding *block* keyword.

# END

---

## Example

```
10  INPUT "Guess a number";A%
    IF A% = 24
    THEN
        PRINT, "YOU GUESSED IT!"
    END IF
    IF A% < 24
    THEN
        PRINT, "BIGGER IS BETTER!"
    GOTO 10
    END IF
    IF A% > 24
    THEN
        PRINT, "SMALLER IS BETTER!"
        GOTO 10
    END IF
END PROGRAM
```



---

## ERL

The ERL function returns the number of the BASIC line where the last error occurred.

---

### Format

*int-var* = ERL

---

### Syntax Rules

The value of *int-var* returned by the ERL function is a LONG integer.

---

### Remarks

1. If the ERL function is used before an error occurs or after an error is handled, the results are undefined.
2. The ERL function overrides the /NOLINE qualifier. If a program compiled with the /NOLINE qualifier in effect contains an ERL function, VAX BASIC signals the message "ERL overrides NOLINE".

# ERL

---

## Example

```
10 DECLARE LONG int_exp
   WHEN ERROR USE error_routine
20 INPUT "Enter an integer expression";int_exp
30 PRINT DATE$(int_exp)
   END WHEN
   HANDLER error_routine
   IF ERL = 20
   THEN
       PRINT "Invalid input...try again"
       RETRY
   ELSE
       PRINT "UNEXPECTED ERROR"
       EXIT HANDLER
   END IF
   END HANDLER
   END PROGRAM
```

### Output

```
Enter an integer expression? ABCD
Error occurred on line 20
Enter an integer expression? 3211
09-Oct-89
```

---

## ERN\$

The ERN\$ function returns the name of the main program, subprogram, or DEF function that was executing when the last error occurred.

---

### Format

*str-var* = ERN\$

---

### Syntax Rules

None.

---

### Remarks

1. If the ERN\$ function executes before an error occurs or after an error is handled, ERN\$ returns a null string.
2. If you call a subprogram or function compiled with /NOSETUP or containing an OPTION INACTIVE=SETUP statement, the ERN\$ function will not have a valid value if an exception occurs in the called procedure.

# ERN\$

---

## Example

```
10 DECLARE LONG int_exp
   !This module's name is DATE
   WHEN ERROR IN
   INPUT "Enter a number";int_exp
   USE
       PRINT "Error in module ";ERN$
       RETRY
   END WHEN
   PRINT Date$(int_exp)
   END
```

### Output

```
Enter a number? ABCD
Error in module DATE
Enter a number? 0
11-Oct-89
```

---

## **ERR**

The ERR function returns the error number of the current run-time error.

---

### **Format**

*int-var* = ERR

---

### **Syntax Rules**

The value of *int-var* returned by the ERR function is always a LONG integer.

---

### **Remarks**

If the ERR function is used before an error occurs or after an error is handled, the results are undefined.

# ERR

---

## Example

```
10 DECLARE LONG int_exp
   WHEN ERROR USE error_routine
20 INPUT "Enter an integer expression";int_exp
   PRINT DATE$(int_exp)
   END WHEN
   HANDLER error_routine:
       PRINT "Error number";ERR
       IF ERR = 50 THEN PRINT "DATA FORMAT ERROR"
       ELSE PRINT "UNEXPECTED ERROR"
       END IF
       RETRY
   END HANDLER
   END
```

### Output

```
Enter an integer expression? ABCD
Error number 50
DATA FORMAT ERROR
Enter an integer expression? 0
06-Oct-89
```

---

## ERT\$

The ERT\$ function returns explanatory text associated with an error number.

---

### Format

*str-var* = ERT\$ (*int-exp*)

---

### Syntax Rules

*Int-exp* is a VAX BASIC error number. The error number must be in the range 0 through 255.

---

### Remarks

1. The ERT\$ function can be used at any time to return the text associated with a specified error number.
2. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to an integer of the default size.

# ERT\$

---

## Example

```
10 DECLARE LONG int_exp
   WHEN ERROR USE error_routine
20 INPUT "Enter an integer expression";int_exp
   PRINT DATE$(int_exp)
   END WHEN
   HANDLER error_routine
       PRINT "Error number";ERR
       PRINT ERT$(ERR)
       RETRY
   END HANDLER
END
```

### Output

```
Enter an integer expression? ABCD
Error number 50
%Data format error
Enter an integer expression? 89
15-Oct-89
```



---

## EXIT

The EXIT statement lets you exit from a main program, a SUB, FUNCTION, or PICTURE subprogram, a multi-line DEF, a statement block, or a handler.

---

### Format

**EXIT** *block*

*block:* { **DEF**[ *exp* ]  
**FUNCTION**[ *exp* ]  
**SUB**  
**HANDLER**  
**PICTURE**  
**PROGRAM**[ *int-exp* ]  
*label* }

---

### Syntax Rules

1. The DEF, FUNCTION, SUB, HANDLER, and PROGRAM keywords specify the type of subprogram, multi-line DEF, or handler from which VAX BASIC is to exit.
2. If you specify an optional expression with the EXIT DEF statement or with the EXIT FUNCTION statement, the expression becomes the function result and supercedes any function assignment. It also overrides any expression specified on the END DEF or END FUNCTION statement. Note that the expression must be compatible with the FUNCTION or DEF data type.
3. *Label* specifies a statement label for an IF, SELECT, FOR, WHILE, or UNTIL statement block.

# EXIT

---

## Remarks

1. An **EXIT SUB**, **EXIT FUNCTION**, **EXIT PROGRAM**, **EXIT DEF**, or **EXIT PICTURE** statement is equivalent to an unconditional branch to an equivalent **END** statement. Control then passes to the statement that invoked the **DEF** or to the statement following the statement that called the subprogram.
2. The **EXIT HANDLER** statement causes **VAX BASIC** to transfer control to a specified area.
  - If the current **WHEN** block is nested, control transfers to the handler associated with the next outer protected region.
  - If an **ON ERROR** statement is in effect and the current **WHEN** block is not nested, control transfers to the target of the **ON ERROR** statement.
  - If neither of the previous conditions is true, an **EXIT HANDLER** statement transfers control to the calling program or **DCL**. This action is the equivalent of the **ON ERROR GO BACK** statement.
3. The **EXIT PROGRAM** statement causes **VAX BASIC** to exit from a main program module.
  - An optional integer expression on an **EXIT PROGRAM** statement specifies the exit status of the program that is reported to **DCL**.
  - The expression specified by an **EXIT PROGRAM** statement overrides any integer expression specified by an **END PROGRAM** statement.
  - **VAX BASIC** allows you to specify an **EXIT PROGRAM** statement without a matching **PROGRAM** statement.
4. The **EXIT label** statement is equivalent to an unconditional branch to the first statement following the end of the **IF**, **SELECT**, **FOR**, **WHILE**, or **UNTIL** statement labeled by the specified label.
5. An **EXIT FUNCTION**, **EXIT SUB** or **EXIT PROGRAM** statement cannot be used within a multi-line **DEF** function.
6. When the **EXIT FUNCTION**, **EXIT SUB** or **EXIT PROGRAM** statement executes, **VAX BASIC** releases all storage allocated to local variables and returns control to the calling program.

---

## Example

```
DEF emp.bonus(A)
IF A > 10
THEN
    PRINT "OUT OF RANGE"
    EXIT DEF 0
ELSE
    emp.bonus = A * 4
END IF
END DEF
INPUT A
PRINT emp.bonus(A)
END
```

### Output

```
? 11
OUT OF RANGE
0
```

# EXP

---

## EXP

The **EXP** function returns the value of the mathematical constant  $e$  raised to a specified power.

---

### Format

*real-var* = **EXP** (*real-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. The **EXP** function returns the value of  $e$  raised to the power of *real-exp*.
2. **VAX BASIC** expects the argument of the **EXP** function to be a real expression. When the argument is a real expression, **VAX BASIC** returns a value of the same floating-point size. When the argument is not a real expression, **VAX BASIC** converts the argument to the default floating-point size and returns a value of the default floating-point size.
3. When the default **REAL** size is **SINGLE** or **DOUBLE**, **EXP** allows arguments between  $-88$  and  $88$ . If the default **REAL** size is **GFLOAT**, **EXP** allows arguments between  $-709$  and  $709$ . If the default **REAL** size is **HFLOAT**, the arguments can be in the range  $-11356$  to  $11355$ . When the argument exceeds the upper limit of a range, **VAX BASIC** signals an error. When the argument exceeds the lower limit of a range, the **EXP** function returns a zero and **VAX BASIC** does not signal an error.

---

**Example**

```
DECLARE SINGLE num_val  
num_val = EXP(4.6)  
PRINT num_val
```

**Output**

99.4843

# EXTERNAL

---

## EXTERNAL

The **EXTERNAL** statement declares constants, variables, functions, and subroutines external to your program. You can describe parameters for external functions and subroutines.

---

### Format

#### External Constants

**EXTERNAL** *data-type* **CONSTANT** *const-name*,...

#### External Variables

**EXTERNAL** *data-type* *unsubs-var*,...

#### External Functions

**EXTERNAL** *data-type* **FUNCTION** { *func-name* [ *pass-mech* ]  
[ ( *external-param* ,... ) ] },...

#### External Subroutines

**EXTERNAL SUB** { *sub-name* [ *pass-mech* ] [ ( *external-param* ,... ) ] },...

*pass-mech*: { **BY VALUE**  
**BY REF**  
**BY DESC** }

*external-param*: [ **OPTIONAL** ] [ *param-data-type* ] [ **DIM**( [ , ] ... ) ]  
[ = *int-const* ] [ *pass-mech* ]

#### External Pictures

**EXTERNAL PICTURE** *pic-name* [ ( *param-list* ) ]

---

## Syntax Rules

1. For external constants, *data-type* can be BYTE, WORD, LONG, SINGLE, INTEGER (any size), or REAL (if the default size is SINGLE).
2. For external variables, the data type can be any valid numeric data type.
3. For external functions and subroutines, the data type can be BYTE, WORD, LONG, SINGLE, DOUBLE, GFLOAT, HFLOAT, DECIMAL, STRING, INTEGER, REAL, RFA, or a data type defined with the RECORD statement. See Table 1–2 in this manual for more information on data type size, range and precision.
4. The name of an external constant, variable, function, or subroutine can be from 1 through 31 characters.
5. For all external routine declarations, the name must be a valid VAX BASIC identifier and must not be the same as any other SUB, FUNCTION, PICTURE, or PROGRAM name.
6. *Param-data-type* specifies the data type of a parameter. If you do not specify a data type, parameters are of the default data type and size.
7. *Param-list* is identical to *external-param* except that no OPTIONAL parameter is allowed.
8. Parameters in the *param-list* must agree in number and data type with the parameters in the invocation. *Param-data-type* includes ANY, BYTE, WORD, LONG, INTEGER, SINGLE, DOUBLE, GFLOAT, HFLOAT, READ, a user-defined RECORD type, STRING, or RFA.

For more information on external pictures, see the *Programming with VAX BASIC Graphics*.

### 9. External Functions and Subroutines

- The data type that precedes the keyword FUNCTION defines the data type of the function result.
- *Pass-mech* specifies how parameters are to be passed to the function or subroutine.
  - A *pass-mech* clause outside the parentheses applies to all parameters.

# EXTERNAL

- A *pass-mech* clause inside the parentheses overrides the previous *pass-mech* and applies only to the specific parameter.
- *External-param* defines the form of the arguments passed to the external function or subprogram. Empty parentheses indicate that the subprogram expects zero parameters. Missing parentheses indicate that the EXTERNAL statement does not define parameters.

## 10. Using ANY as a VAX BASIC Data Type

- The ANY data type should only be used for calling non-BASIC procedures. Therefore, the ANY data type is illegal in a PICTURE declaration.
- If you specify ANY, VAX BASIC does not perform data type checking or conversions. If no passing mechanism is specified, VAX BASIC uses the default passing mechanism for the data type passed in a given invocation.
- When you specify a data type, all following parameters that are not specifically declared default to the last specified data type. Similarly, when you specify ANY, all following unspecified parameters default to the data type ANY until a new declaration is provided. For example:

```
EXTERNAL SUB allocate (LONG,ANY, )
```

## 11. Passing Optional Parameters

- The OPTIONAL keyword should be used only for calling non-BASIC procedures.
- If you specify the keyword OPTIONAL, VAX BASIC treats all following parameters as optional. In the following example, the last three parameters are optional.

```
EXTERNAL SUB queue (STRING, OPTIONAL STRING, LONG, ANY)
```

- When a procedure is called, the argument pointer (@AP) contains the number of actual parameters specified.
- VAX BASIC still performs type checking and conversion on optional parameters.



- If you want to omit an optional parameter that appears in the middle of a parameter list, VAX BASIC requires you to insert a comma placeholder. However, if you want to omit an optional parameter that appears at the end of a parameter list, you can omit that parameter without inserting any placeholder.
- You can specify the keyword `OPTIONAL` only once in any one parameter list.

## 12. Declaring Array Dimensions

The `DIM` keyword indicates that the parameter is an array. Commas specify array dimensions. The number of dimensions is equal to the number of commas plus 1. For example:

```
EXTERNAL STRING FUNCTION new (DOUBLE, STRING DIM(,), DIM( ))
```

This statement declares a function named *new* that has three parameters. The first is a double-precision floating-point value, the second is a two-dimensional string array, and the third is a one-dimensional string array. The function returns a string result.

---

## Remarks

1. The `EXTERNAL` statement must precede any program reference to the constant, variable, function, subroutine or picture declared in the statement.
2. The `EXTERNAL` statement is not executable.
3. A name declared in an `EXTERNAL CONSTANT` statement can be used in any nondeclarative statement as if it were a constant.
4. A name declared in an `EXTERNAL FUNCTION` statement can be used as a function invocation in an expression. In addition, you can invoke a function with the `CALL` statement unless the function data type is `DECIMAL`, `HFLOAT`, or `STRING`.
5. A name declared in an `EXTERNAL SUB` statement can be used in a `CALL` statement.

# EXTERNAL

6. The optional *pass-mech* clauses in the EXTERNAL FUNCTION and EXTERNAL SUB statements tell VAX BASIC how to pass arguments to a non-BASIC function or subprogram. Table 4-1 describes VAX BASIC parameter-passing mechanisms.
  - BY VALUE specifies that VAX BASIC passes the argument's 32-bit value.
  - BY REF specifies that VAX BASIC passes the argument's address. This is the default for all arguments except strings and entire arrays. If you know the size of string parameters and the dimensions of array parameters, you can improve run-time performance by passing strings and arrays by reference.
  - BY DESC specifies that VAX BASIC passes the address of a VAX BASIC descriptor. For information about the format of a VAX BASIC descriptor for strings and arrays, see Appendix C in this manual.
7. If you do not specify the data type ANY or declare parameters as optional, the arguments passed to external functions and subroutines should match the external parameters declared in the EXTERNAL FUNCTION or EXTERNAL SUB statement in number, type, and passing mechanism. VAX BASIC forces arguments to be compatible with declared parameters. If they are not compatible, VAX BASIC signals an error.

---

## Examples

### Example 1

```
!External Constant  
EXTERNAL LONG CONSTANT SSS_NORMAL
```

### Example 2

```
!External Variable  
EXTERNAL WORD SYSNUM
```

# EXTERNAL

## Example 3

```
!External Function  
EXTERNAL DOUBLE FUNCTION USR$2(WORD, LONG, ANY)
```

## Example 4

```
!External Subroutine  
EXTERNAL SUB calc BY DESC (STRING DIM(,), BYTE BY REF)
```

## FIELD

---

## FIELD

The **FIELD** statement dynamically associates string variables with all or parts of a record buffer. **FIELD** statements do not move data. Instead, they permit direct access through string variables to sections of a specified record buffer.

### NOTE

The **FIELD** statement is supported only for compatibility with **BASIC-PLUS**. Because data defined in the **FIELD** statement can be accessed only as string data, you must use the **CVTxx** functions to process numeric data; therefore, you must convert string data to numeric after you move it from the record buffer. Then, after processing, you must convert numeric data back to string data before transferring it to the record buffer. **DIGITAL** recommends that you use **VAX BASIC**'s dynamic mapping feature or multiple maps instead of the **FIELD** statement and **CVTxx** functions.

---

### Format

**FIELD** *#chnl-exp, int-exp AS str-var[ , int-exp AS str-var ]...*

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#). A file must be open on the specified channel or **VAX BASIC** signals an error.
2. *Int-exp* specifies the number of characters in *str-var*. However, a subsequent *int-exp* cannot depend on the return string from a previous *int-exp*. For example, the following statement is illegal because the second *int-exp* depends on the return string **A\$**:

```
FIELD #1%, 1% AS A$, ASCII(A$) AS B$
```

---

## Remarks

1. A FIELD statement is executable. You can change a buffer description at any time by executing another FIELD statement. For example:

```
FIELD #1%, 40% AS whole_field$  
FIELD #1%, 10% AS A$, 10% AS B$, 10% AS C$, 10% AS D$
```

The first FIELD statement associates the first 40 characters of a buffer with the variable *whole\_field\$*. The second FIELD statement associates the first 10 characters of the same buffer with *A\$*, the second 10 characters with *B\$*, and so on. Later program statements can refer to any of the variables named in the FIELD statements to access specific portions of the buffer.

2. You cannot define virtual array strings as string variables in a FIELD statement.
3. A variable named in a FIELD statement cannot be used in a COMMON or MAP statement, as a parameter in a CALL or SUB statement, or in a MOVE statement.
4. Using the FIELD statement on a virtual file that contains a virtual array causes VAX BASIC to signal "Illegal or illogical access" (ERR=136).
5. If you name an array in a FIELD statement, you cannot use MAT statements of the format

```
MAT array-name1 = array-name2 MAT array-name1 = NUL$
```

where *array-name1* is named in the FIELD statement. An attempt to do so causes VAX BASIC to signal a compile-time error.

# FIELD

---

## Example

```
FIELD #8%, 2% AS U$, 2% AS CL$, 4% AS X$, 4% AS Y$
LSET U$ = CVT%$(U%)
LSET CL$ = CVT%$(CL%)
LSET X$ = CVT$(X)
LSET Y$ = CVT$(Y)
U% = CVT%$(U$)
CL% = CVT%$(CL$)
X = CVT$(X$)
Y = CVT$(Y$)
```

# FIND

The **FIND** statement locates a specified record in a disk file and makes it the current record for a **GET**, **UPDATE**, or **DELETE** operation. **FIND** statements are valid on RMS sequential, relative, and indexed files.

## Format

**FIND** #*chnl-exp* [ , *position-clause* ][ , *lock-clause*]

*position-clause*: { **RFA** *rfa-exp*  
**RECORD** *num-exp*  
**KEY#** *key-clause* }

*lock-clause*: { **ALLOW** *allow-clause* [ , **WAIT** [ *int-exp* ] ]  
**WAIT** [ *int-exp* ]  
**REGARDLESS** }

*allow-clause*: { **NONE**  
**READ**  
**MODIFY** }

*key-clause*: *int-exp1* *rel-op* *key-exp*

*rel-op*: { **EQ**  
**GE**  
**NXEQ**  
**GT**  
**NX** }

# FIND

*key-exp*:  $\left\{ \begin{array}{l} \textit{int-exp2} \\ \textit{str-exp} \\ \textit{decimal-exp} \\ \textit{quadword-exp} \end{array} \right\}$

---

## Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. If you specify a *lock-clause*, it must follow the *position-clause*. If the *lock-clause* precedes the *position-clause*, VAX BASIC signals an error.
3. If you specify the REGARDLESS *lock-clause*, you cannot specify another *lock-clause* in the same FIND statement.

---

## Remarks

### 1. Position-clause

- *Position-clause* specifies the position of a record in a file. VAX BASIC signals an error if you specify a *position-clause* and the channel is not associated with a disk file. If you do not specify a *position-clause*, FIND locates records sequentially. Sequential record access is valid on all files.
- The RFA *position-clause* allows you to randomly locate records by specifying the record file address (RFA) of a record. You specify the disk address of a record, and RMS locates the record at that address. All file organizations can be accessed by RFA.

*Rfa-exp* in the RFA *position-clause* is a variable of the RFA data type that specifies the record's file address. Note that an RFA expression can only be a variable of the RFA data type or the GETRFA function. Use the GETRFA function to find the RFA of a record.



# FIND

- The **RECORD *position-clause*** allows you to randomly locate records in relative and sequential fixed files by specifying the record number.
  - *Num-exp* in the **RECORD *position-clause*** specifies the number of the record you want to locate. It must be between 1 and the number of the record with the highest number in the file.
  - When you specify a **RECORD clause**, *chnl-exp* must be a channel associated with an open relative or sequential fixed file.
- The **KEY *position-clause*** allows you to randomly locate records in indexed files by specifying a key of reference, a relational test, and a key value.
- An RFA value is valid only for the life of a specific version of a file. If a new version of a file is created, the RFA values may change.
- Attempting to access a record with an invalid RFA value results in a run-time error.

## 2. Lock-clause

- **Lock-clause** allows you to control how a record is locked to other access streams, to override lock checking when accessing shared files that may contain locked records, or to specify what action to take in the case of a locked record.
- The type of lock you impose on a record remains in effect until you explicitly unlock it with a **FREE** or **UNLOCK** statement, until you close the file, or until you perform a **GET**, **FIND**, **UPDATE** or **DELETE** on the same channel (unless you specified **UNLOCK EXPLICIT**).
- The **REGARDLESS *lock-clause*** specifies that the **FIND** statement can override lock checking and locate a record locked by another program.
- When you specify a **REGARDLESS *lock-clause***, **VAX BASIC** does not impose a lock on the retrieved record.
- The **ALLOW *lock-clause*** lets you control how a record is locked to other users and access streams. The file associated with the specified channel must have been opened with the **UNLOCK EXPLICIT** clause or **VAX BASIC** signals the error “Illegal record locking clause”.

# FIND

- The **ALLOW** *allow-clause* can be one of the following:
  - **ALLOW NONE** denies access to the record. This means that other access streams cannot retrieve the record unless they bypass lock checking with the **GET REGARDLESS** clause.
  - **ALLOW READ** provides read access to the record. This means that other access streams can retrieve the record but cannot use the **DELETE** or **UPDATE** statements on the record.
  - **ALLOW MODIFY** provides read and write to the record. This means that other access streams can use the **GET**, **FIND**, **DELETE**, and **UPDATE** statements on the record.
- If you do not open a file with the **ACCESS READ** clause or specify an *allow-clause*, locking is imposed as follows:
  - If the file associated with the specified channel was opened with **UNLOCK EXPLICIT**, **VAX BASIC** imposes the **ALLOW NONE** lock on the retrieved record and the next **GET** or **FIND** operation does not unlock the previously locked record.
  - If the file associated with the specified channel was not opened with **UNLOCK EXPLICIT**, **VAX BASIC** locks the retrieved record and unlocks the previously locked record.
- The **WAIT** *lock-clause* accepts an optional *int-exp*. *Int-exp* represents a timeout value in seconds. *Int-exp* must be from 0 through 255 or **VAX BASIC** signals a warning message.
  - **WAIT** followed by a timeout value causes **RMS** to wait for a locked record for a given period of time.
  - **WAIT** followed by no timeout value indicates that **RMS** should wait indefinitely for the record to become available.
  - If you specify a timeout value and the record does not become available within that period, **VAX BASIC** signals the the run-time error “Keyboard wait exhausted” (**ERR=15**). **VMSSTATUS** and **RMSSTATUS** then return **RMS\$\_TMO**. For more information on the **RMSSTATUS** and **VMSSTATUS** functions, see this chapter and the *VAX BASIC User Manual*.
  - If you attempt to wait for a record that another user has locked, and consequently that user attempts to wait for the record you have locked, a deadlock condition occurs. When a deadlock condition persists for a period of time (as defined

by the SYSGEN parameter DEADLOCK\_WAIT), RMS signals the error "RMS\$\_DEADLOCK" and VAX BASIC signals the error "Detected deadlock error while waiting for GET or FIND" (ERR=193).

- If you specify a WAIT clause followed by a timeout value that is less than the SYSGEN parameter DEADLOCK\_WAIT, VAX BASIC signals the error "Keyboard wait exhausted" (ERR=15) even though a deadlock condition may exist.

### 3. Key-clause

- In a *key-clause*, *int-exp1* is the target key of reference. It must be an integer in the range of zero through the highest-numbered key for the file. The primary key is #0, the first alternate key is #1, the second alternate key is #2, and so on. *Int-exp1* must be preceded by a number sign (#) or VAX BASIC signals an error.
- When you specify a *key-clause*, the specified channel must be a channel associated with an open indexed file.

### 4. Rel-op

- *Rel-op* is a relational operator that specifies how *key-exp* is to be compared with *int-exp1* in the *key-clause*.
  - EQ means "equal to"
  - NXEQ means "next or equal to"
  - GE means "greater than or next" (a synonym for NXEQ)
  - NX means "next"
  - GT means "greater than" (a synonym for NX)
- A successful random FIND operation by key locates the first record whose key satisfies the *key-clause* comparison:
  - With an exact key match (EQ), a successful FIND locates the first record in the file that equals the key value specified in *key-exp*. However, if the characters specified by a *str-exp* key expression are less than the key length, characters specified by *str-exp* are matched approximately rather than exactly. For example, if you specify "ABC" and the key length is six characters, VAX BASIC locates the first record that begins with ABC. If you specify "ABCABC", VAX BASIC locates only a record with the key "ABCABC". If no match is possible, VAX BASIC signals the error "Record not found" (ERR=155).

# FIND

- If you specify a next or equal to record key match (NXEQ), a successful FIND locates the next record that equals the key length specified in *int-exp* or *str-exp*. If no exact match exists, VAX BASIC locates the next record in the key sort order. If the keys are in ascending order, the next record will have a greater key value. If the keys are in descending order, the next record will have a lesser key value.
- If you specify a greater than or equal to key match (GE), the behavior is identical to that of next or equal to (NXEQ). (Likewise, the behavior of GT is identical to NX.) However, the use of GE in a descending key file may be confusing, since GE will retrieve the next record in the key sort order but the next record will have a lesser key value. For this reason, DIGITAL recommends that you use NXEQ in new program development, especially if you are using descending key files.
- If you specify a next key match (NX), a successful FIND locates the first record that follows the relational operator in the sort order. If no such record exists, VAX BASIC signals the error "Record not found" (ERR=155).

## 5. Key-exp

- *Int-exp2* specifies an integer value to be compared with the key value of a record.
  - *Str-exp* specifies a string value to be compared with the key value of a record. *Str-exp* can contain fewer characters than the key of the record you want to locate, but cannot be a null string.
  - *Decimal-exp* in the key clause specifies a packed decimal value to be compared with the key value of a record.
  - *Quadword-exp* in the key clause specifies a a record or group which is exactly 8 bytes long that is to be compared with the key value of a record.
6. The file on the specified channel must have been opened with ACCESS MODIFY, ACCESS READ, or SCRATCH before your program can execute a FIND operation.
  7. FIND does not transfer any data to the record buffer. To access the contents of a record, use the GET statement.

8. A successful sequential FIND operation updates both the current record pointers and next record pointers.
  - For sequential files, a successful FIND operation locates the next sequential record (the record pointed to by the next record pointer) in the file, changes the current record pointer to the record just found, and the next record pointer to the next sequential record. If the current record pointer points to the last record in a file, a sequential FIND operation causes VAX BASIC to signal “Record not found” (ERR=155).
  - For relative files, a successful FIND operation locates the record that exists with the next higher record number (or cell number), makes it the current record, and changes the next record pointer to the current record pointer plus 1.
  - For indexed files, a successful FIND operation locates the next existing logical record in the current key of reference, makes this the current record and changes the next record pointer to the current record pointer plus 1.
9. A successful random access FIND operation by RFA or by record changes the current record pointer to the record specified by *rfa-exp* or *int-exp*, but leaves the next record pointer unchanged.
10. A successful random access FIND operation by key changes the current record pointer to the first record whose key satisfies the *key-clause* comparison and leaves the next record pointer unchanged.
11. When a random access FIND operation by RFA, record, or key is not successful, VAX BASIC signals “Record not found” (ERR=155). The values of the current record pointer and next record pointer are undefined.
12. You should not use a FIND statement on a terminal-format or virtual array file.

# FIND

---

## Example

```
DECLARE LONG rec_num
MAP (cusrec) WORD cus_num
    STRING cus_nam=20, cus_add=20, cus_city=10, cus_zip=9
OPEN "CUS_ACCT.DAT" FOR INPUT AS #1,
    RELATIVE FIXED,
    ACCESS MODIFY
    MAP cusrec
INPUT "Which record number would you like to delete";rec_num
FIND #1, RECORD rec_num, WAIT
DELETE #1
CLOSE #1
END
```

---

## FIX

The **FIX** function truncates a floating-point value at the decimal point and returns the integer portion represented as a floating-point value.

---

### Format

*real-var* = **FIX** (*real-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. The **FIX** function returns the integer portion of a floating-point value, not an integer value.
2. **VAX BASIC** expects the argument of the **FIX** function to be a real expression. When the argument is a real expression, **VAX BASIC** returns a value of the same floating-point size. When the argument is not a real expression, **VAX BASIC** converts the argument to the default floating-point size and returns a value of the default floating-point size.
3. If *real-exp* is negative, **FIX** returns the negative integer portion. For example, **FIX**(-5.2) returns -5.

# FIX

---

## Example

```
DECLARE SINGLE result
result = FIX(-3.333)
PRINT FIX(24.566), result
```

### Output

```
24          -3
```



---

**FNEND**

The **FNEND** statement is a synonym for the **END DEF** statement. See the **END** statement for more information.

---

**Format**

**FNEND** [*exp*]

## **FNEXIT**

---

## **FNEXIT**

The **FNEXIT** statement is a synonym for the **EXIT DEF** statement. See the **EXIT** statement for more information.

---

### **Format**

**FNEXIT** [*exp*]

---

# FOR

The FOR statement repeatedly executes a block of statements, while incrementing a specified control variable for each execution of the statement block. FOR loops can be conditional or unconditional, and can modify other statements.

---

## Format

### Unconditional

**FOR** *num-unsubs-var* = *num-exp1* **TO** *num-exp2* [**STEP** *num-exp3*] [*statement*]...

**NEXT** *num-unsubs-var*

### Conditional

**FOR** *num-unsubs-var* = *num-exp1* [**STEP** *num-exp3*] { **UNTIL** }  
*cond-exp* { **WHILE** }  
 [*statement*]...

**NEXT** *num-unsubs-var*

### Unconditional Statement Modifier

*statement* **FOR** *num-unsubs-var* = *num-exp1* **TO** *num-exp2* [**STEP** *num-exp3*]

### Conditional Statement Modifier

*statement* **FOR** *num-unsubs-var* = *num-exp1* [**STEP** *num-exp3*]

{ **UNTIL** } *cond-exp*  
 { **WHILE** }

# FOR

---

## Syntax Rules

1. *Num-unsubs-var* must be a numeric, unsubscripted variable. *Num-unsubs-var* cannot be a record field.
2. *Num-unsubs-var* is the loop variable. It is incremented each time the loop executes.
3. In unconditional FOR loops, *num-exp1* is the initial value of the loop variable; *num-exp2* is the maximum value.
4. In conditional FOR loops, *num-exp1* is the initial value of the loop variable, while the *cond-exp* in the WHILE or UNTIL clause is the condition that controls loop iteration.
5. *Num-exp3* in the STEP clause is the value by which the loop variable is incremented after each execution of the loop.

---

## Remarks

1. There is a limit to the number of inner loops you can contain within a single outer loop. This number varies according to the complexity of the loops. If you exceed the limit, VAX BASIC signals an error message.
2. An inner loop must be entirely within an outer loop; the loops cannot overlap.
3. You cannot use the same loop variable in nested FOR loops. For example, if the outer loop uses FOR *I* = 1 TO 10, you cannot use the variable *I* as a loop variable in an inner loop.
4. The default for *num-exp3* is 1 if there is no STEP clause.
5. You can transfer control into a FOR loop only by returning from a function invocation, a subprogram call, a subroutine call, or an error handler that was invoked in the loop.
6. The starting, incrementing, and ending values of the loop do not change during loop execution.
7. The loop variable can be modified inside the FOR loop.

8. VAX BASIC converts *num-exp1*, *num-exp2*, and *num-exp3* to the data type of the loop variable before storing them.
9. When an unconditional FOR loop ends, the loop variable contains the value last used in the loop, not the value that caused loop termination.
10. During each iteration of a conditional loop, VAX BASIC tests the value of *cond-exp* before it executes the loop.
  - If you specify a WHILE clause and *cond-exp* is false (value zero), VAX BASIC exits from the loop. If the *cond-exp* is true (value nonzero), the loop executes again.
  - If you specify an UNTIL clause and *cond-exp* is true (value nonzero), VAX BASIC exits from the loop. If the *exp* is false (value zero), the loop executes again.
11. When FOR is used as a statement modifier, VAX BASIC executes the statement until the loop variable equals or exceeds *num-exp2* or until the WHILE or UNLESS condition is satisfied.
12. Each FOR statement must have a corresponding NEXT statement or VAX BASIC signals an error. (This is not the case if the FOR statement is used as a statement modifier.)

---

## Examples

### Example 1

```
!Unconditional
DECLARE LONG course_num, STRING course_name
FOR I = 3 TO 12 STEP 3
INPUT "Course number";course_num
INPUT "Course name";course_name
NEXT I
```

# FOR

## Output

```
Course number? 221
Course name? Botany
Course number? 231
Course name? Organic Chemistry
Course number? 237
Course name? Life Science II
Course number? 244
Course name? Programming in VAX BASIC
```

## Example 2

```
!Unconditional Statement Modifier
DECLARE INTEGER counter
PRINT "This is an unconditional statement modifier" FOR counter = 1 TO 3
END
```

## Output

```
This is an unconditional statement modifier
This is an unconditional statement modifier
This is an unconditional statement modifier
```

## Example 3

```
!Conditional Statement Modifier
DECLARE INTEGER counter, &
      STRING my_name
INPUT "Try and guess my name";my_name FOR counter = 1 UNTIL my_name = "VAX F
PRINT "You guessed it!"
```

## Output

```
Try and guess my name? VAX PASCAL
Try and guess my name? VAX SCAN
Try and guess my name? VAX BASIC
You guessed it!
```

---

## **FORMAT\$**

The **FORMAT\$** function converts an expression to a formatted string.

---

### **Format**

*str-var* = **FORMAT\$** (*exp*, *str-exp*)

---

### **Syntax Rules**

The rules for building a format string are the same as those for printing numbers with the **PRINT USING** statement. See the description of the **PRINT USING** statement for more information.

---

### **Remarks**

**DIGITAL** recommends that you use compile-time constant expressions for string expressions whenever possible. When you do this, the **VAX BASIC** compiler compiles the string at compilation time rather than at run time, thus improving the performance of your code.

# FORMAT\$

---

## Example

```
DECLARE STRING result,          &  
          INTEGER num_exp  
num_exp = 12345  
result = FORMAT$(num_exp,"##,###")  
PRINT result
```

### Output

```
12,345
```



---

**FREE**

The **FREE** statement unlocks all records and buckets associated with a specified channel.

---

**Format**

**FREE** *#chnl-exp*

---

**Syntax Rules**

*Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).

---

**Remarks**

1. The file specified by *chnl-exp* must be open.
2. You cannot use the **FREE** statement with files not on disk.
3. If there are no locked records or buckets on the specified channel, the **FREE** statement has no effect and **VAX BASIC** does not signal an error.
4. The **FREE** statement does not change record buffers or pointers. Moreover, the position of the record pointers is undefined.
5. After a **FREE** statement has executed, your program must execute a **GET** or **FIND** statement before a **PUT**, **UPDATE**, or **DELETE** statement can execute successfully.

# FREE

---

## Example

```
OPEN "CUST_ACCT.DAT" FOR INPUT AS #3
.
.
INPUT "Enter customer record number to retrieve";cust_rec_num
FREE #3
GET #3
```

In this example, CUST\_ACCT.DAT is opened for input. The **FREE** statement unlocks all records associated with the specified channel contained in the file. Once the **FREE** statement successfully executes, the user can then obtain a record with either a **FIND** or **GET** statement.

---

## FSP\$

The FSP\$ function returns a string describing an open file on a specified channel.

---

### Format

*str-var* = FSP\$ (*chnl-exp*)

---

### Syntax Rules

1. A file must be open on *chnl-exp*.
2. The FSP\$ function must come immediately after the OPEN statement for the file.

---

### Remarks

1. Use the FSP\$ function with files opened as ORGANIZATION UNDEFINED. Then use multiple MAP statements to interpret the returned data.
2. See the *VAX BASIC User Manual* and the *VMS Record Management Services Manual* for more information on FSP\$ values.

#### NOTE

VAX BASIC supports the FSP\$ function for compatibility with BASIC-PLUS-2. DIGITAL recommends that you use a USEROPEN routine to identify file characteristics.

# FSP\$

---

## Example

```
10 MAP (A) STRING A = 32
MAP (A) BYTE org, rat, WORD mrs, LONG alq, &
      WORD bks_bls, num_keys, LONG mrn
OPEN "STUDENT.DAT" FOR INPUT AS #1%, &
      ORGANIZATION UNDEFINED, &
      RECORDTYPE ANY, ACCESS READ
A = FSP$(1%)
PRINT "RMS organization = ";org
PRINT "RMS record attributes = ";rat
PRINT "RMS maximum record size = ";mrs
PRINT "RMS allocation quantity = ";alq
PRINT "RMS bucket size = ";bks_bls
PRINT "Number of keys = ";num_keys
PRINT "RMS maximum record number = ";mrn
```

## Output

```
RMS organization = 2
RMS record attributes = 2
RMS maximum record size = 5
RMS allocation quantity = 1
RMS bucket size = 0
Number of keys = 0
RMS maximum record number = 0
```

---

## FUNCTION

The **FUNCTION** statement marks the beginning of a **FUNCTION** subprogram and defines the subprogram's parameters.

---

### Format

**FUNCTION** *data-type func-name* [ *pass-mech* ] [( [ *formal-param* ],... ) ]  
[ *statement* ]...

{ **END FUNCTION** [ *exp* ] }  
{ **FUNCTIONEND** [ *exp* ] }

*pass-mech*: { **BY REF** }  
{ **BY DESC** }

*formal param*: [ *data-type* ] { *unsubs-var*  
*array-name* ( [ *int-const* ] ,... ) }  
[ = *int-const* ][ *pass-mech* ]

---

### Syntax Rules

1. *Func-name* names the **FUNCTION** subprogram.
2. *Func-name* can be from 1 through 31 characters. The first character must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs (\$), periods (.), or underscores (\_).

# FUNCTION

3. *Data-type* can be any VAX BASIC data type keyword or a data type defined in the RECORD statement. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.
4. The data type that precedes the *func-name* specifies the data type of the value returned by the function.
5. *Formal-param* specifies the number and type of parameters for the arguments the function expects to receive when invoked.
  - Empty parentheses indicate that the function has no parameters.
  - *Data-type* specifies the data type of a parameter. If you do not specify a data type, parameters are of the default data type and size. When you do specify a data type, all following parameters are of that data type until you specify a new data type.

If the data type is STRING and the passing mechanism is by reference (BY REF), the *=int-const* clause allows you to specify the length of the string.
  - Parameters defined in *formal-param* must agree in number and type with the arguments specified in the function invocation. VAX BASIC allows you to specify from 1 through 255 formal parameters.
6. *Pass-mech* specifies the parameter-passing mechanism by which the FUNCTION subprogram receives arguments when invoked. A *pass-mech* clause should be specified only when the FUNCTION subprogram is being called by a non-BASIC program or when the FUNCTION receives a string or array by reference.
7. A *pass-mech* clause outside the parentheses applies by default to all function parameters. A *pass-mech* clause in the *formal-param* list overrides the specified default and applies only to the immediately preceding parameter.
8. *Exp* specifies the function result which supersedes any function assignment. *Exp* must be compatible with the function's data type.

---

## Remarks

1. The **FUNCTION** statement must be the first statement in the **FUNCTION** subprogram.
2. Every **FUNCTION** statement must have a corresponding **END FUNCTION** or **FUNCTIONEND** statement.
3. Any **VAX BASIC** statement except **END**, **PICTURE**, **END PICTURE**, **PROGRAM**, **END PROGRAM**, **SUB**, **SUBEND**, **END SUB**, or **SUBEXIT** can appear in a **FUNCTION** subprogram.
4. **FUNCTION** subprograms must be declared with the **EXTERNAL** statement before your **VAX BASIC** program can invoke them.
5. **FUNCTION** subprograms receive parameters by reference or by descriptor.
  - **BY REF** specifies that the function receives the argument's address.
  - **BY DESC** specifies that the function receives the address of a **VAX BASIC** descriptor. For information about the format of a **VAX BASIC** descriptor for strings and arrays, see the *VAX BASIC User Manual*; for information on other types of descriptors, see the *VAX Architecture Handbook*.
6. By default, **FUNCTION** subprograms receive numeric unsubscripted variables by reference, and all other parameters by descriptor. You can override these defaults with a **BY** clause:
  - If you specify a string length with the *=int-const* clause, you must also specify **BY REF**. If you specify **BY REF** and do not specify a string length, **VAX BASIC** uses the default string length of 16.
  - If you specify array bounds, you must also specify **BY REF**.
7. All variables and data, except virtual arrays, **COMMON** areas, **MAP** areas, and **EXTERNAL** variables, in a **FUNCTION** subprogram, are local to the subprogram.

# FUNCTION

8. VAX BASIC initializes local numeric variables to zero and local string variables to the null string each time the FUNCTION subprogram is invoked.
9. If an exception is not handled within the FUNCTION subprogram, control is transferred back to the main program that invoked the function.

---

## Example

```
FUNCTION REAL sphere_volume (REAL R)
IF R < 0 THEN EXIT FUNCTION
sphere_volume = 4/3 * PI *R **3.
END FUNCTION
```



---

## FUNCTIONEND

The FUNCTIONEND statement is a synonym for the END FUNCTION statement. See the END statement for more information.

---

### Format

**FUNCTIONEND** [*exp*]

## FUNCTIONEXIT

---

# FUNCTIONEXIT

The **FUNCTIONEXIT** statement is a synonym for the **EXIT FUNCTION** statement. See the **EXIT** statement for more information.

---

### Format

**FUNCTIONEXIT** [*exp*]

---

## GET

The GET statement moves a record from a file to a record buffer and makes the data available for processing. GET statements are valid on sequential, relative, and indexed files.

---

### Format

**GET** *#chnl-exp* [ , *position-clause* ] [ , *lock-clause* ]

*position-clause*: { **RFA** *rfa-exp*  
**RECORD** *num-exp*  
**KEY#** *key-clause* }

*lock-clause*: { **ALLOW** *allow-clause* [ , **WAIT** [ *int-exp* ] ]  
**WAIT** [ *int-exp* ]  
**REGARDLESS** }

*allow-clause*: { **NONE**  
**READ**  
**MODIFY** }

*key-clause*: *int-exp1* *rel-op* *key-exp*

*rel-op*: { **EQ**  
**GE**  
**NXEQ**  
**GT**  
**NX** }

# GET

*key-exp*       $\left\{ \begin{array}{l} \textit{int-exp2} \\ \textit{str-exp} \\ \textit{decimal-exp} \\ \textit{quadword-exp} \end{array} \right\}$

---

## Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. If you specify a *lock-clause*, it must follow the *position-clause*. If the *lock-clause* precedes the *position-clause* VAX BASIC signals an error.
3. If you specify the REGARDLESS *lock-clause*, you cannot specify another *lock-clause* in the same GET statement.

---

## Remarks

1. **Position-clause**
  - *Position-clause* specifies the position of a record in a file. VAX BASIC signals an error if you specify a *position-clause* and *chnl-exp* is not associated with a disk file. If you do not specify a *position-clause*, GET retrieves records sequentially. Sequential record access is valid on all files.
  - The RFA *position-clause* allows you to randomly retrieve records by specifying the record file address (RFA); you specify the disk address of a record, and RMS retrieves the record at that address. All file organizations can be accessed by RFA.

*Rfa-exp* in the RFA *position-clause* is an expression of the RFA data type that specifies the record's file address. An RFA expression must be a variable of the RFA data type or the GETRFA function. Use the GETRFA function to obtain the RFA of a record.

- The **RECORD *position-clause*** allows you to randomly retrieve records in relative and sequential fixed files by specifying the record number.
  - *Num-exp* in the **RECORD *position-clause*** specifies the number of the record you want to retrieve. It must be between 1 and the number of the record with the highest number in the file.
  - When you specify a **RECORD clause**, *chnl-exp* must be a channel associated with an open relative or sequential fixed file.
- The **KEY *position-clause*** allows you to randomly retrieve records in indexed files by specifying a key of reference, a relational test, or a key value.
- An RFA value is valid only for the life of a specific version of a file. If a new version of a file is created, the RFA values may change.
- Attempting to access a record with an invalid RFA value results in a run-time error.

## 2. Lock-clause

- **Lock-clause** allows you to control how a record is locked to other access streams, to override lock checking when accessing shared files that may contain locked records, or to specify what action to take in the case of a locked record.
- The type of lock you impose on a record remains in effect until you explicitly unlock it with a **FREE** or **UNLOCK** statement, until you close the file, or until you perform a **GET**, **FIND**, **UPDATE** or **DELETE** on the same channel (unless you specified **UNLOCK EXPLICIT**).
- The **REGARDLESS *lock-clause*** specifies that the **GET** statement can override lock checking and read a record locked by another program.
- When you specify a **REGARDLESS *lock-clause***, **VAX BASIC** does not impose a lock on the retrieved record.
- If you specify an **ALLOW *lock-clause*** the file associated with *chnl-exp* must have been opened with the **UNLOCK EXPLICIT** clause or **VAX BASIC** signals the error “Illegal record locking clause”.
- The **ALLOW *allow-clause*** can be one of the following:
  - **ALLOW NONE** denies access to the record. This means that other access streams cannot retrieve the record unless they bypass lock checking with the **REGARDLESS** clause.

## GET

- ALLOW READ provides read access to the record. This means that other access streams can retrieve the record, but cannot DELETE or UPDATE the record.
- ALLOW MODIFY provides both read and write access to the record. This means that other access streams can GET, FIND, DELETE, or UPDATE the record.
- If you do not open a file with ACCESS READ or specify an ALLOW *lock-clause*, locking is imposed as follows:
  - If the file associated with *chnl-exp* was opened with UNLOCK EXPLICIT, VAX BASIC imposes the ALLOW NONE lock on the retrieved record and the next GET or FIND statement does not unlock the previously locked record.
  - If the file associated with *chnl-exp* was not opened with UNLOCK EXPLICIT, VAX BASIC locks the retrieved record and unlocks the previously locked record.
- The WAIT *lock-clause* accepts an optional *int-exp*. *Int-exp* represents a timeout value in seconds. *Int-exp* must be from 0 through 255 or VAX BASIC issues a warning message.
  - WAIT followed by a timeout value causes RMS to wait for a locked record for a given period of time.
  - WAIT followed by no timeout value indicates that RMS should wait indefinitely for the record to become available.
  - If you specify a timeout value and the record does not become available within that period, VAX BASIC signals the the run-time error "Keyboard wait exhausted" (ERR=15). VMSSTATUS and RMSSTATUS then return RMS\$\_TMO. For more information on the RMSSTATUS and VMSSTATUS functions, see this chapter and the *VAX BASIC User Manual*.
  - If you attempt to wait for a record that another user has locked, and consequently that user attempts to wait for the record you have locked, a deadlock condition occurs. When a deadlock condition persists for a period of time (as defined by the SYSGEN parameter DEADLOCK\_WAIT), RMS signals the error "RMS\$\_DEADLOCK" and VAX BASIC signals the error "Detected deadlock error while waiting for GET or FIND" (ERR=193).

- If you specify a WAIT clause followed by a timeout value that is less than the SYSGEN parameter DEADLOCK\_WAIT, then VAX BASIC signals the error “Keyboard wait exhausted” (ERR=15) even though a deadlock condition may exist.
- If you specify a WAIT clause on a GET operation to a unit device, the timeout value indicates how long to wait for the input to complete. This is equivalent to the WAIT statement.

### 3. Key-clause

- In a *key-clause*, *int-exp1* is the target key of reference. It must be a integer value in the range of zero through the highest-numbered key for the file. The primary key is #0, the first alternate key is #1, the second alternate key is #2, and so on. *Int-exp1* must be preceded by a number sign (#) or VAX BASIC signals an error.
- When you specify a key clause, *chnl-exp* must be a channel associated with an open indexed file.

### 4. Rel-op

- *Rel-op* specifies how *key-exp* is to be compared with *int-exp1* in the *key-clause*.
  - EQ means “equal to”
  - NXEQ means “next or equal to”
  - GE means “greater than or equal to” (a synonym for NXEQ)
  - NX means “next”
  - GT means “greater than” (a synonym for NX)
- With an exact key match (EQ), a successful GET operation retrieves the first record in the file that equals the key value specified in *key-exp*. If the key expression is a *str-exp* whose length is less than the key length, characters specified by the *str-exp* are matched approximately rather than exactly. That is, if you specify a string expression “ABC” and the key length is six characters, VAX BASIC matches the first record that begins with ABC. If you specify “ABCABC”, VAX BASIC matches only a record with the key “ABCABC”. If no match is possible, VAX BASIC signals the error “Record not found” (ERR=155).

# GET

- If you specify a next or equal to key match (NXEQ), a successful GET operation retrieves the first record that equals the key value specified in *key-exp*. If no exact match exists, VAX BASIC retrieves the next record in the key sort order. If the keys are in ascending order, the next record will have a greater key value. If the keys are in descending order, the next record will have a lesser key value.
- If you specify a greater than key match (GT), a successful GET operation retrieves the first record with a value greater than *key-exp*. If no such record exists, VAX BASIC signals the error “Record not found” (ERR=155).
- If you specify a next key match (NX), a successful GET operation retrieves the first record that follows the key expression in the key sort order. If no such record exists, VAX BASIC signals the error “Record not found” (ERR=155).
- If you specify a greater than or equal to key match (GE), the behavior is identical to that of next or equal to (NXEQ). Likewise, the behavior of GT is identical to NX. However, the use of GE in a descending key file may be confusing, because GE will retrieve the next record in the key sort order but the next record will have a lesser key value. For this reason, DIGITAL recommends that you use NXEQ in new program development, especially if you are using descending key files.

## 5. Key-exp

- *Int-exp2* in the key clause specifies an integer value to be compared with the key value of a record.
  - *Str-exp* in the key clause specifies a string value to be compared with the key value of a record. The string expression can contain fewer characters than the key of the record you want to retrieve but it cannot be a null string.
  - *Decimal-exp* in the key clause specifies a packed decimal value to be compared with the key value of a record.
  - *Quadword-exp* in the key clause specifies a RECORD or GROUP exactly 8 bytes long to be compared with the key value of a record.
6. The file specified by *chnl-exp* must be opened with ACCESS READ or ACCESS MODIFY or SCRATCH before your program can execute a GET statement. The default ACCESS clause is MODIFY.



# GET

7. If the last I/O operation was a successful FIND operation, a sequential GET operation retrieves the current record located by the FIND operation and sets the next record pointer to the record logically succeeding the pointer.
8. If the last I/O operation was not a FIND operation, a sequential GET operation retrieves the next record and sets the record logically succeeding the record pointer to the current record.
  - For sequential files, a sequential GET operation retrieves the next record in the file.
  - For relative files, a sequential GET operation retrieves the record with the next higher cell number.
  - For indexed files, a sequential GET operation retrieves the next record in the current key of reference.
9. A successful random GET operation by RFA or by record retrieves the record specified by *rfa-exp* or *int-exp*.
10. A successful random GET operation by key retrieves the first record whose key satisfies the *key-clause* comparison.
11. A successful random GET operation by RFA, record, or key sets the value of the current record pointer to the record just read. The next record pointer is set to the next logical record.
12. An unsuccessful GET operation leaves the record pointers and the record buffer in an undefined state.
13. If the retrieved record is smaller than the receiving buffer, VAX BASIC fills the remaining buffer space with nulls.
14. If the retrieved record is larger than the receiving buffer, VAX BASIC truncates the record and signals an error.
15. A successful GET operation sets the value of the RECOUNT variable to the number of bytes transferred from the file to the record buffer.
16. You should not use a GET statement on a terminal-format or virtual array file.

# GET

---

## Example

```
DECLARE LONG rec-num
MAP (CUSREC) WORD cus_num
    STRING cus_nam = 20, cus_add = 20, cus_city = 10, cus_zip = 9 &
OPEN "CUS_ACCT.DAT" FOR INPUT AS #1 &
    RELATIVE FIXED, ACCESS MODIFY, &
    MAP CUSREC
INPUT "Which record number would you like to view";rec_num
GET #1, RECORD REC_NUM, REGARDLESS
PRINT "The customer's number is ";CUS_NUM
PRINT "The customer's name is ";cus_nam
PRINT "The customer's address is ";cus_add
PRINT "The customer's city is ";cus_city
PRINT "The customer's zip code is ";cus_zip
CLOSE #1
END
```

---

## GETRFA

The GETRFA function returns the record's file address (RFA) of the last record accessed in an RMS file open on a specified channel.

---

### Format

*rfa-var* = **GETRFA** (*chnl-exp*)

---

### Syntax Rules

1. *Rfa-var* is a variable of the RFA data type.
2. *Chnl-exp* is the channel number of an open RMS file. You cannot include a number sign in the channel expression.
3. You must access a record in the file with a GET, FIND, or PUT statement before using the GETRFA function, or VAX BASIC signals "No current record" (ERR=131).

---

### Remarks

1. There must be a file open on the specified *chnl-exp* or VAX BASIC signals an error.
2. You can use the GETRFA function with RMS sequential, relative, indexed, and block I/O files.
3. The RFA value returned by the GETRFA function can be used only for assignments to and comparisons with other variables of the RFA data type. Comparisons are limited to equal to (=) and not equal to (<>) relational operations.
4. RFA values cannot be printed or used for any arithmetic operations.

# GETRFA

---

## Example

```
DECLARE RFA R_ARRAY(1 TO 100)
      .
      .
FOR I% = 1% TO 100%
    PUT #1
      R_ARRAY(I%) = GETRFA(1)
NEXT I%
```

---

## GOSUB

The GOSUB statement transfers control to a specified line number or label and stores the location of the GOSUB statement for eventual return from the subroutine.

---

### Format

**{ GO SUB  
GOSUB }** *target*

---

### Syntax Rules

1. *Target* must refer to an existing line number or label in the same program unit as the GOSUB statement or VAX BASIC signals an error.
2. *Target* cannot be inside a block structure such as a FOR...NEXT, WHILE, or UNTIL loop or a multi-line function definition unless the GOSUB statement is also within that block or function definition.

---

### Remarks

1. You can use the GOSUB statement from within protected regions of a WHEN block. GOSUB statements can also contain protected regions themselves.
2. If you fail to handle an exception that occurs while a statement contained in the body of a subroutine is executing, the exception is handled by the default error handler. The exception is not handled by any WHEN block surrounding the statement that invoked the subroutine.

# GOSUB

---

## Example

```
GOSUB subroutine_1
.
.
.
subroutine_1:
.
.
.
RETURN
```

---

## GOTO

The GOTO statement transfers control to a specified line number or label.

---

### Format

$\left\{ \begin{array}{l} \text{GO TO} \\ \text{GOTO} \end{array} \right\} \textit{target}$

---

### Syntax Rules

1. *Target* must refer to an existing line number or label in the same program unit as the GOTO statement or VAX BASIC signals an error.
2. *Target* cannot be inside a block structure such as a FOR...NEXT, WHILE, or UNTIL loop or a multi-line function definition unless the GOTO statement is also inside that loop or function definition.

---

### Remarks

1. You can specify the GOTO statement inside a WHEN block if the target is in the same protected region, an outer level protected region, or in a nonprotected region.
2. You cannot specify the GOTO statement inside a WHEN block if the target already resides in another protected region that does not contain the innermost current protected region.

# GOTO

---

## Example

```
IF answer = 0
    THEN GOTO done
END IF
.
.
done:
    EXIT PROGRAM
```



---

## HANDLER

The handler statement marks the beginning of a detached handler.

---

### Format

**HANDLER** *handler-name*

---

### Syntax Rules

*Handler-name* must be a valid VAX BASIC identifier and must not be the same as any label, DEF, DEF\*, SUB, FUNCTION or PICTURE name.

---

### Remarks

1. A detached handler must be delimited by a HANDLER statement and an END HANDLER statement.
2. A detached handler can be used only with VAX BASIC's exception-handling mechanism. If you attempt to branch into a detached handler for example with the GOTO statement, VAX BASIC signals a compile-time error.
3. To exit from a detached handler, you must use either END HANDLER, EXIT HANDLER, RETRY or CONTINUE. See these statements for more information.
4. Within a handler, VAX BASIC allows you to specify user-defined function references and procedure invocations as well as VAX BASIC statements.

## HANDLER

5. The following statements are illegal inside a handler:
- EXIT PROGRAM, FUNCTION, SUB, or PICTURE
  - GOTO to a target outside the handler
  - GOSUB to a target outside the handler
  - ON ERROR
  - RESUME

---

### Example

```
WHEN ERROR USE err_handler
.
.
.
END WHEN
HANDLER err_handler
  IF ERR = 50 THEN PRINT "Insufficient data"
    RETRY
  ELSE EXIT HANDLER
END IF
END HANDLER
```

---

## IF

The IF statement evaluates a conditional expression and transfers program control depending on the resulting value.

---

### Format

#### Conditional

```
IF cond-exp THEN statement... [ELSE statement...]  
END IF
```

#### Statement Modifier

```
statement IF cond-exp
```

---

## Syntax Rules

### 1. Conditional

- *Cond-exp* can be any valid conditional expression.
- All statements between the THEN keyword and the next ELSE, line number, or END IF are part of the THEN clause. All statements between the keyword ELSE and the next line number or END IF are part of the ELSE clause.
- VAX BASIC assumes a GOTO statement when the keyword ELSE is followed by a line number. When the target of a GOTO statement is a label, the keyword GOTO is required. The use of this syntax is not recommended for new program development.
- The END IF statement terminates the most recent unterminated IF statement.
- A new line number terminates all unterminated IF statements.

# IF

## 2. Statement Modifier

- IF can modify any executable statement except a block statement such as FOR, WHILE, UNTIL, or SELECT.
- *Cond-exp* can be any valid conditional expression.

---

## Remarks

### 1. Conditional

- VAX BASIC evaluates the conditional expression for truth or falsity. If true (nonzero), VAX BASIC executes the THEN clause. If false (zero), VAX BASIC skips the THEN clause and executes the ELSE clause, if present.
- The keyword NEXT cannot be in a THEN or ELSE clause unless the FOR or WHILE statement associated with the keyword NEXT is also part of the THEN or ELSE clause.
- If a THEN or ELSE clause contains a block statement such as a FOR, SELECT, UNTIL, or WHILE, then a corresponding block termination statement such as a NEXT or END, must appear in the same THEN or ELSE clause.
- IF statements can be nested to 12 levels.
- Any executable statement is valid in the THEN or ELSE clause, including another IF statement. You can include any number of statements in either clause.
- Execution continues at the statement following the END IF or ELSE clause. If the statement does not contain an ELSE clause, execution continues at the next statement after the THEN clause.

### 2. Statement Modifier

- VAX BASIC executes the statement only if the conditional expression is true (nonzero).

---

## Example

```
IF Update_flag = True
THEN
    Weekly_salary = New_rate * 40.0
    UPDATE #1
    IF Dept <> New_dept
    THEN
        GET #1, KEY #1 EQ New_dept
        Dept_employees = Dept_employees + 1
        UPDATE #1
    END IF
    PRINT "Update complete"
ELSE
    PRINT "Skipping update for this employee"
END IF
```

## INKEY\$

---

## INKEY\$

The INKEY\$ function reads a single keystroke from a terminal opened on a specified channel and returns the typed character.

---

### Format

*string-var* = INKEY\$ (*chnl-exp* [, WAIT [ *int-exp* ] ])

---

### Syntax Rules

1. *Chnl-exp* must be the channel number of a terminal.
  2. *Int-exp* represents the timeout value in seconds and must be from 0 through 255. Values beyond this range cause VAX BASIC to signal a compile-time or run-time error.
- 

### Remarks

1. Before using the INKEY\$ function, specify the DCL command SET TERMINAL/HOSTSYNC. This command controls whether the system can synchronize the flow of input from the terminal. If you specify SET TERMINAL/HOSTSYNC, the system generates a CTRL/S or a CTRL/Q to enable or disable the reception of input. This prevents the typeahead buffer from overflowing. If you do not use this command and the typeahead buffer overflows, VAX BASIC signals the error "Data overflow" (ERR=289).
2. Before using the INKEY\$ function on a VT200-series terminal, set your terminal to VT200 mode with 7 bit controls.

3. Before using the INKEY\$ function, either your terminal or VMS, but not both, must enable screen wrapping. To enable terminal screen wrapping, use the Set-Up key on your terminal's keyboard to set the terminal to Auto Wrap. Then disable VMS screen wrapping by entering the DCL SET TERMINAL /NOWRAP command. To enable VMS screen wrapping, enter the DCL SET TERMINAL /WRAP command. Then disable terminal screen wrapping by using the Set-Up key to set the terminal to No Auto Wrap.
4. The INKEY\$ function behaves as if the terminal were in APPLICATION\_KEYPAD mode.
5. If the channel is not open, VAX BASIC signals the error "I/O, channel not open" (ERR=9). If a file or a device other than a terminal is open on the channel, VAX BASIC signals the error "Illegal operation" (ERR=141).
6. The optional WAIT clause specifies a timeout interval during which the command will await terminal input. If you specify WAIT *int-exp*, the timeout period will be the specified number of seconds. If you specify a WAIT clause followed by no timeout value, VAX BASIC waits indefinitely for terminal input.
7. VAX BASIC always examines the typeahead buffer first and retrieves the next keystroke in the buffer if the buffer is not empty. If the typeahead buffer is empty and an optional WAIT clause was specified, VAX BASIC waits for a keystroke to be typed for the specified timeout interval (indefinitely if WAIT was specified with no timeout interval). If the typeahead buffer is empty, and the waiting period is either not specified or expired, VAX BASIC returns the error message "Keyboard wait exhausted" (ERR=15).
8. The escape character (ASCII code 27) is not valid as INKEY\$ input. If you enter an escape character, normal program execution resumes when the INKEY\$ times out. Without a specified timeout value, the program execution cannot resume without error.
9. VAX BASIC returns the error message "Keyboard wait exhausted" (ERR=15) when any key is pressed after the escape character if no timeout is specified or if the specified timeout has not yet occurred.
10. INKEY\$ turns off all line editing. As a result, control of all line-editing characters and the arrow keys is passed back to the user.

## INKEY\$

11. Non-editing characters normally intercepted by the VMS terminal driver are not returned. These include the CTRL/C, CTRL/Y, CTRL/S, CTRL/Y/O characters (unless CTRL/C trapping is enabled). They are handled by the device driver just as in normal input.
12. All ASCII characters are returned in a 1-byte string.
13. All keystrokes that result in an escape sequence are translated to mnemonic strings based on the following key names:
  - PF1–PF4
  - E1–E6
  - F7–F20
  - LEFT
  - RIGHT
  - UP
  - DOWN
  - KP0 to KP9
  - KP–
  - KP,
  - KP.
  - ENTER



---

## Example

```
PROGRAM Inkey_demo
  DECLARE STRING KEYSTROKE
Inkey_Loop:
  WHILE 1%
    KEYSTROKE = INKEY$(0%,WAIT)

    SELECT KEYSTROKE
      CASE '26'C
        PRINT "CTRL/Z to exit"
        EXIT Inkey_Loop
      CASE CR,LF,VT,FF
        PRINT "Line terminator"
      CASE "PF1" TO "PF4"
        PRINT "P function key"
      CASE "E1" TO "E6", "F7" TO "F9", "F10" TO "F20"
        PRINT "VT200 function key"
      CASE "KP0" TO "KP9"
        PRINT "Application keypad key"
      CASE < SP
        PRINT "Control character"
      CASE '127'C
        PRINT "<DEL>"
      CASE ELSE
        PRINT 'Character is "', KEYSTROKE, "'"
    END SELECT
  NEXT
END PROGRAM
```

# INPUT

---

## INPUT

The INPUT statement assigns values from your terminal or from a terminal-format file to program variables.

---

### Format

```
INPUT [#chnl-exp, ][ str-const1 { ' ; } ] var1 [ { ' ; } [ str-const2 { ' ; } ]  
var2 ]..
```

---

### Syntax Rules

1. You must supply an argument to the INPUT statement. Otherwise, VAX BASIC signals an error message.
2. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
3. You can include more than one string constant in an INPUT statement. *Str-const1* is issued for *str-var1*, *str-const2* for *str-var2*, and so on.
4. *Str-var1* and *str-var2* cannot be a DEF function names unless the INPUT statement is inside the multi-line DEF that defines the function.
5. The separator (comma or semicolon) that directly follows *str-var1* and *str-var2* has no formatting effect. VAX BASIC always advances to a new line when you terminate input with a carriage return.
6. The separator that directly follows *str-const1* and *str-const2*, determines where the question mark prompt (if requested) is displayed and where the cursor is positioned for input.

A comma causes VAX BASIC to skip to the next print zone and display the question mark unless a SET NO PROMPT statement has been executed, as follows.

# INPUT

```
DECLARE STRING your_name  
INPUT "What is your name",your_name
```

## Output

```
What is your name      ?
```

A semicolon causes VAX BASIC to display the question mark next to *str-const* unless a SET NO PROMPT statement has been executed. For example:

```
DECLARE STRING your_name  
INPUT "What is your name";your_name
```

## Output

```
What is your name?
```

7. VAX BASIC always advances to a new line when you terminate input with a carriage return.

---

## Remarks

1. If you do not specify a channel, the default *chnl-exp* is #0 (the controlling terminal). If a *chnl-exp* is specified, a file must be open on that channel with ACCESS READ or MODIFY before the INPUT statement can execute.
2. If input comes from a terminal, VAX BASIC displays the contents of *str-const1*, if present. If the terminal is open on channel #0, VAX BASIC also displays a question mark (?).
3. You can disable the question mark prompt by using the SET NO PROMPT statement. See the SET PROMPT statement for more information.
4. When VAX BASIC receives a line terminator or a complete record, it checks each data element for correct data type and range limits, then assigns the values to the corresponding variables.
5. If you specify a string variable to receive the input text, and the user enters an unquoted string in response to the prompt, VAX BASIC ignores the string's leading and trailing spaces and tabs. An unquoted string cannot contain any commas.

# INPUT

6. If there is not enough data in the current record or line to satisfy the variable list, VAX BASIC takes one of the following actions:
  - If the input device is a terminal and you have not specified SET NO PROMPT, VAX BASIC repeats the question mark but not the *str-const*, on a new line until sufficient data is entered.
  - If the input device is not a terminal, VAX BASIC signals “Not enough data in record” (ERR=59).
7. If there are more data items than variables in the INPUT response, VAX BASIC ignores the excess.
8. If there is an error while data is being converted or assigned (for example, string data being assigned to a numeric variable), VAX BASIC takes one of the following actions:
  - If there is no error handler in effect and the input device is a terminal, VAX BASIC signals a warning, reexecutes the INPUT statement, and displays *str-const* and the input prompt.
  - If there is an error handler in effect and the input device is not a terminal, VAX BASIC signals “Illegal number” (ERR=52) or “Data format error” (ERR=50).
9. When a RETRY, CONTINUE or RESUME statement transfers control to an INPUT statement, the INPUT statement retrieves a new record or line regardless of any data left in the previous record or line.
10. After a successful INPUT statement, the RECOUNT variable contains the number of characters transferred from the file or terminal to the record buffer.
11. If you terminate input text with CTRL/Z, VAX BASIC assigns the value to the variable and signals “End of file on device” (ERR=11) when the next terminal input statement executes. If you are in the BASIC environment and there is no subsequent INPUT, INPUT LINE, or LINPUT statement in the program, the CTRL/Z is passed to VAX BASIC as a signal to exit the BASIC environment. VAX BASIC signals “Unsaved changes have been made, CTRL/Z or EXIT to exit” if you have made changes to your program or are running a program that has never been saved. If you have not made changes, VAX BASIC exits from the BASIC environment and does not signal an error.

---

**Example**

```
DECLARE STRING var_1, &  
          INTEGER var_2  
INPUT "The first variable";var_1, "The second variable";var_2
```

**Output**

```
The first variable? name  
The second variable? 4
```

## INPUT LINE

---

## INPUT LINE

The INPUT LINE statement assigns a string value (including the line terminator in some cases) from a terminal or terminal-format file to a string variable.

---

### Format

```
INPUT LINE [#chnl-exp, ][ str-const1 { ' ; } ] str-var1  
          [ statement ]...[ { ' ; } [ str-const2 { ' ; } ] str-var2 ]...
```

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Str-var1* or *str-var2* cannot be a DEF function name unless the INPUT LINE statement is inside the multi-line DEF that defines the function.
3. You can include more than 1 string constant in an INPUT LINE statement. *Str-const1* is issued for *str-var1*, *str-const2* for *str-var2*, and so on.
4. The separator (comma or semicolon) that directly follows *str-var1* and *str-var2* has no formatting effect. VAX BASIC always advances to a new line when you terminate input with a carriage return.
5. The separator that directly follows *str-const1* and *str-const2* determines where the question mark (if requested) is displayed and where the cursor is positioned for input. Specifically:
  - A comma causes VAX BASIC to skip to the next print zone and display the question mark unless a SET NO PROMPT statement has been executed. For example:

# INPUT LINE

```
DECLARE STRING your_name
INPUT LINE "Name",your_name
```

## Output

Name                    ?

- A semicolon causes VAX BASIC to display the question mark next to *str-const* unless a SET NO PROMPT statement has been executed.  
For example:

```
DECLARE STRING your_name
INPUT LINE "Name";your_name
```

## Output

Name?

6. VAX BASIC always advances to a new line when you terminate input with a carriage return.

---

## Remarks

1. The default *chnl-exp* is #0 (the controlling terminal). If a channel is specified, a file must be open on that channel with ACCESS READ before the INPUT LINE statement can execute.
2. VAX BASIC signals an error if the INPUT LINE statement has no argument.
3. If input comes from a terminal, VAX BASIC displays the contents of *str-const1*, if present. If the terminal is open on channel #0, VAX BASIC also displays a question mark (?).
4. You can disable the question mark prompt by using the SET NO PROMPT statement. See the SET PROMPT statement for more information.
5. The INPUT LINE statement assigns all input characters to string variables. In addition, the INPUT LINE statement places the following line terminator characters in the assigned string if they are part of the string value.

# INPUT LINE

Hex code	ASCII char	Character name
0A	LF	Line Feed
0B	VT	Vertical Tab
0C	FF	Form Feed
0D	CR	Carriage Return
0D0A	CRLF	Carriage Return/Line Feed
1B	ESC	Escape

Any other line terminator, such as CRTL/D and CTRL/F when line editing is turned off, is not included in the assigned string.

6. When a `RETRY`, `CONTINUE` or `RESUME` statement transfers control to an `INPUT LINE` statement, the `INPUT LINE` statement retrieves a new record or line regardless of any data left in the previous record or line.
7. After a successful `INPUT LINE` statement, the `RECOUNT` variable contains the number of characters transferred from the file or terminal to the record buffer.
8. If you terminate input text with `CTRL/Z`, `VAX BASIC` assigns the value to the variable and signals "End of file on device" (`ERR=11`) when the next terminal input statement executes. If you are in the `BASIC` environment and there is no next `INPUT`, `INPUT LINE`, or `LINPUT` statement in the program, the `CTRL/Z` is passed to `VAX BASIC` as a signal to exit the `BASIC` environment. `VAX BASIC` signals "Unsaved changes have been made, `CTRL/Z` or `EXIT` to exit" if you have made changes to your program. If you have not made changes, `VAX BASIC` exits from the `BASIC` environment and does not signal an error.

---

## Example

```
DECLARE STRING Z,N,record_string
INPUT LINE "Type two words", Z$, 'Type your name';N$
INPUT LINE #4%, record_string$
```



---

## INSTR

The INSTR function searches for a substring within a string. It returns the position of the substring's starting character.

---

### Format

*int-var* = INSTR (*int-exp*, *str-exp1*, *str-exp2*)

---

### Syntax Rules

1. *Int-exp* specifies the character position in the main string at which VAX BASIC starts the search.
2. *Str-exp1* specifies the main string.
3. *Str-exp2* specifies the substring.

---

### Remarks

1. The INSTR function searches *str-exp1*, the main string, for the first occurrence of a substring, *str-exp2*, and returns the position of the substring's first character.
2. INSTR returns the character position in the main string at which VAX BASIC finds the substring, except in the following situations:
  - If only the substring is null, and if *int-exp* is less than or equal to zero, INSTR returns a value of 1.
  - If only the substring is null, and if *int-exp* is equal to or greater than 1 and less than or equal to the length of the main string, INSTR returns the value of *int-exp*.
  - If only the substring is null, and if *int-exp* is greater than the length of the main string, INSTR returns the main string's length plus 1.

# INSTR

- If the substring is not null, and if *int-exp* is greater than the length of the main string, INSTR returns a value of zero.
  - If only the main string is null, INSTR returns a value of zero.
  - If both the main string and the substring are null, INSTR returns a 1.
3. If VAX BASIC cannot find the substring, INSTR returns a value of zero.
  4. If *int-exp* does not equal 1, VAX BASIC still counts from the beginning of the main string to calculate the starting position of the substring. That is, VAX BASIC counts character positions starting at position 1, regardless of where you specify the start of the search. For example, if you specify 10 as the start of the search and VAX BASIC finds the substring at position 15, INSTR returns the value 15.
  5. If *int-exp* is less than 1, VAX BASIC assumes a starting position of 1.
  6. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to an integer of the default size.

---

## Example

```
DECLARE STRING alpha,    &  
          INTEGER result  
alpha = "ABCDEF"  
result = INSTR(1, alpha, "DEF")  
PRINT result
```

### Output

4

---

# INT

The INT function returns the floating-point value of the largest whole number less than or equal to a specified expression.

---

## Format

*real-var* = INT (*real-exp*)

---

## Syntax Rules

VAX BASIC expects the argument of the INT function to be a real expression. When the argument is a real expression, VAX BASIC returns a value of the same floating-point size. When the argument is not a real expression, VAX BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

---

## Remarks

If *real-exp* is negative, VAX BASIC returns the largest whole number less than or equal to *real-exp*. For example, INT(-5.3) is -6.

# INT

---

## Examples

### Example 1

```
DECLARE SINGLE any_num, result
any_num = 6.667
result = INT(any_num)
PRINT result
```

### Output

6

### Example 2

```
!This example contrasts the INT and FIX functions
DECLARE SINGLE test_num
test_num = -32.7
PRINT "INT OF -32.7 IS: "; INT(test_num)
PRINT "FIX OF -32.7 IS: "; FIX(test_num)
```

### Output

```
INT OF -32.7 IS: -33
FIX OF -32.7 IS: -32
```

---

## INTEGER

The **INTEGER** function converts a numeric expression or numeric string to a specified or default **INTEGER** data type.

---

### Format

$$int-var = \mathbf{INTEGER} (exp \left[ \begin{array}{l} , \mathbf{BYTE} \\ , \mathbf{WORD} \\ , \mathbf{LONG} \end{array} \right] )$$

---

### Syntax Rules

*Exp* can be either numeric or string. A string expression can contain the ASCII digits 0 through 9, a plus sign (+), or a minus sign (-).

---

### Remarks

1. **VAX BASIC** evaluates *exp*, then converts it to the specified **INTEGER** size. If you do not specify a size, **VAX BASIC** uses the default **INTEGER** size.
2. If *exp* is a string, **VAX BASIC** ignores leading and trailing spaces and tabs.
3. The **INTEGER** function returns a value of zero when a string argument contains only spaces and tabs, or when it is null.
4. The **INTEGER** function truncates the decimal portion of **REAL** and **DECIMAL** numbers.

# INTEGER

---

## Example

```
INPUT "Enter a floating-point number";F_P  
PRINT INTEGER(F_P, WORD)
```

### Output

```
Enter a floating-point number? 76.99  
76
```

---

## ITERATE

The ITERATE statement allows you to explicitly reexecute a loop.

---

### Format

**ITERATE** [*label*]

---

### Syntax Rules

1. Label is the label of the first statement of a FOR...NEXT, WHILE, or UNTIL loop.
2. Label must conform to the rules for naming variables.

---

### Remarks

1. ITERATE is equivalent to an unconditional branch to the current loop's NEXT statement. If you supply a label, ITERATE transfers control to the NEXT statement in the specified loop. If you do not supply a label, ITERATE transfers control to the current loop's NEXT statement.
2. The ITERATE statement can be used only within a FOR...NEXT, WHILE, or UNTIL loop.

# ITERATE

---

## Example

```
WHEN ERROR IN
Date_loop: WHILE 1% = 1%
    GET #1
    ITERATE Date_loop IF Day$ <> Today$
    ITERATE Date_loop IF Month$ <> This_month$
    ITERATE Date_loop IF Year$ <> This_year$
    PRINT Item$
NEXT
USE
IF ERR = 11
THEN
    CONTINUE DONE
ELSE
    EXIT HANDLER
END IF
END WHEN
Done: END
```



---

## **KILL**

The **KILL** statement deletes a disk file, removes the file's directory entry, and releases the file's storage space.

---

### **Format**

**KILL** *file-spec*

---

### **Syntax Rules**

*File-spec* can be a quoted string constant, a string variable, or a string expression. It cannot be an unquoted string constant.

---

### **Remarks**

1. The **KILL** statement marks a file for deletion but does not delete the file until all users have closed it.
2. If you do not specify a complete file specification, VAX BASIC uses the default device and directory. If you do not specify a file version, VAX BASIC deletes the highest version of the file.
3. The file must exist, or VAX BASIC signals an error.
4. You can delete a file in another directory if you have access to that directory and privilege to delete the file.

---

### **Example**

```
KILL "TEMP.DAT"
```

# LBOUND

---

## LBOUND

The LBOUND function returns the lower bounds of a compile-time or run-time dimensioned array.

---

### Format

*num-var* = **LBOUND** (*array-name* [, *num-exp* ])

---

### Syntax Rules

1. *Array-name* must specify an array that has been either explicitly or implicitly declared.
2. *Num-exp* specifies the number of the dimension for which you have requested the lower bound.

---

### Remarks

1. If you do not specify a dimension, VAX BASIC automatically returns the lower bounds of the first dimension.
2. If you specify a numeric expression that is less than or equal to zero, VAX BASIC signals an error.
3. If you specify a numeric expression that exceeds the number of dimensions, VAX BASIC signals an error.

---

## Example

```
DECLARE INTEGER CONSTANT B = 5
DIM A(B)
account_num = 1
FOR dim_num = LBOUND (A) TO 5
    A(dim_num) = account_num
    account_num = account_num + 1
    PRINT A(dim_num)
NEXT dim_num
```

### Output

```
1
2
3
4
5
6
```

## LEFT\$

---

## LEFT\$

The LEFT\$ function extracts a specified substring from a string's left side, leaving the main string unchanged.

---

### Format

*str-var* = LEFT[\$] (*str-exp*, *int-exp*)

---

### Syntax Rules

1. *Int-exp* specifies the number of characters to be extracted from the left side of *str-exp*.
2. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to an integer of the default size.

---

### Remarks

1. The LEFT\$ function extracts a substring from the left of the specified *str-exp* and stores it in *str-var*.
2. If *int-exp* is less than 1, LEFT\$ returns a null string.
3. If *int-exp* is greater than the length of *str-exp*, LEFT\$ returns the entire string.

---

**Example**

```
DECLARE STRING sub_string, main_string
main_string = "1234567"
sub_string = LEFT$(main_string, 4)
PRINT sub_string
```

**Output**

1234

# LEN

---

## LEN

The LEN function returns an integer value equal to the number of characters in a specified string.

---

### Format

*int-var* = **LEN** (*str-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. If *str-exp* is null, LEN returns a value of zero.
2. The length of *str-exp* includes leading, trailing, and embedded blanks. Tabs in *str-exp* are treated as a single space.
3. The value returned by the LEN function is a LONG integer.

---

## Example

```
DECLARE STRING alpha, &  
        INTEGER length  
alpha = "ABCDEFGH"  
length = LEN(alpha)  
PRINT length
```

### Output

7

# LET

---

## LET

The LET statement assigns a value to one or more variables.

---

### Format

[LET] *var,... = exp*

---

### Syntax Rules

1. *Var* cannot be a DEF or FUNCTION name unless the LET statement occurs inside that DEF block or in that FUNCTION subprogram.
2. The keyword LET is optional.

---

### Remarks

1. You cannot assign string data to a numeric variable or unquoted numeric data to a string variable.
2. The value assigned to a numeric variable is converted to the variable's data type. For example, if you assign a floating-point value to an integer variable, VAX BASIC truncates the value to an integer.
3. For dynamic strings, the destination string's length equals the source string's length.
4. When you assign a value to a fixed-length string variable (a variable declared in a COMMON, MAP, or RECORD statement), the value is left-justified and padded with spaces or truncated to match the length of the string variable.



---

## Example

```
DECLARE STRING alpha, &  
          INTEGER length  
LET alpha = "ABCDEFGH"  
LET length = LEN(alpha)  
PRINT length
```

### Output

7

# LINPUT

---

## LINPUT

The LINPUT statement assigns a string value, without line terminators, from a terminal or terminal-format file to a string variable.

---

### Format

**LINPUT** [*#chnl-exp*, ] [ *str-const1* { ' ; } ] *str-var1* [ { ' ; } ] [ *str-const2* { ' ; } ]  
*str-var2* ]...

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Str-var1* and *str-var2* cannot be DEF function names unless the LINPUT statement is inside the multi-line DEF that defines the function.
3. You can include more than one string constant in a LINPUT statement. *Str-const1* is issued for *str-var1*, *str-const2* for *str-var2*, and so on.
4. The separator (comma or semicolon) that directly follows *str-var1* and *str-var2* has no formatting effect. VAX BASIC always advances to a new line when you terminate input with a carriage return.
5. The separator character that directly follows *str-const1* and *str-const2* determines where the question mark (if requested) is displayed and where the cursor is positioned for input.
  - A comma causes VAX BASIC to skip to the next print zone to display the question mark unless a SET NO PROMPT statement has been executed. For example:

```
DECLARE STRING your_name  
LINPUT "Name",your_name
```

## Output

Name                    ?

- A semicolon causes VAX BASIC to display the question mark next to *str-const* unless a SET NO PROMPT statement has been executed. For example:

```
DECLARE STRING your_name
LINPUT "What is your name";your_name
```

## Output

What is your name?

6. VAX BASIC always advances to a new line when you terminate input with a carriage return.

---

## Remarks

1. The default *chnl-exp* is #0 (the controlling terminal). If you specify a channel, the file associated with that channel must have been opened with ACCESS READ or MODIFY.
2. VAX BASIC signals an error if the LINPUT statement has no argument.
3. If input comes from a terminal, VAX BASIC displays the contents of *str-const1*, if present. If the terminal is open on channel #0, VAX BASIC also displays a question mark (?).
4. You can disable the question mark prompt by using the SET NO PROMPT statement. See the SET PROMPT statement for more information.
5. The LINPUT statement assigns all characters, except any line terminator's to *str-var1* and *str-var2*. Single and double quotation marks, commas, tabs, leading and trailing spaces, or other special characters in the string are part of the data.
6. If the RETRY, CONTINUE or RESUME statement transfers control to a LINPUT statement, the LINPUT statement retrieves a new record regardless of any data left in the previous record.

# LINPUT

7. After a successful LINPUT statement, the RECOUNT variable contains the number of bytes transferred from the file or terminal to the record buffer.
8. If you terminate input text with CTRL/Z, VAX BASIC assigns the value to the variable and signals "End of file on device" (ERR=11) when the next terminal input statement executes. If you are in the BASIC environment and there is no next INPUT, INPUT LINE, or LINPUT statement in the program, the CTRL/Z is passed to VAX BASIC as a signal to exit the BASIC environment.

---

## Example

```
DECLARE STRING last_name
LINPUT "ENTER YOUR LAST NAME";Last_name
LINPUT #2%, Last_name
```

---

## LOC

The LOC function returns a longword integer specifying the virtual address of a simple or subscripted variable, or the address of an external function. For dynamic strings, the LOC function returns the address of the descriptor rather than the address of the data.

---

### Format

$$int-var = \text{LOC} \left( \left\{ \begin{array}{l} var \\ ext-routine \end{array} \right\} \right)$$

---

### Syntax Rules

1. *Var* can be any local or external, simple or subscripted variable.
2. *Var* cannot be a virtual array element.
3. *Ext-routine* can be the name of an external function.

---

### Remarks

1. The LOC function always returns a LONG value.
2. The LOC function is useful for passing the address of an external function as a parameter to a procedure. When passing a routine address as a parameter you should usually pass the address by value. For example, VMS system services expect to receive AST procedure entry masks by reference; therefore, the address of the entry mask should be in the argument list on the stack.

# LOC

---

## Example

```
DECLARE INTEGER A, B  
A = 12  
B = LOC(A)  
PRINT B
```

### Output

```
2146799372
```

---

## LOG

The LOG function returns the natural logarithm (base  $e$ ) of a specified number. The LOG function is the inverse of the EXP function.

---

### Format

*real-var* = LOG (*real-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. *Real-exp* must be greater than zero. An attempt to find the logarithm of zero or a negative number causes VAX BASIC to signal "Illegal argument in LOG" (ERR=53).
2. The LOG function uses the mathematical constant  $e$  as a base. VAX BASIC approximates  $e$  to be 2.718281828459045 (double precision).
3. The LOG function returns the exponent to which  $e$  must be raised to equal *real-exp*.
4. VAX BASIC expects the argument of the LOG function to be a real expression. When the argument is a real expression, VAX BASIC returns a value of the same floating-point size. When the argument is not a real expression, VAX BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

# LOG

---

## Example

```
DECLARE SINGLE exponent  
exponent = LOG(98.6)  
PRINT exponent
```

### **Output**

```
4.59107
```



---

## LOG10

The LOG10 function returns the common logarithm (base 10) of a specified number.

---

### Format

*real-var* = LOG10 (*real-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. *Real-exp* must be larger than zero. An attempt to find the logarithm of zero or a negative number causes VAX BASIC to signal "Illegal argument in LOG" (ERR=53).
2. The LOG10 function returns the exponent to which 10 must be raised to equal *real-exp*.
3. VAX BASIC expects the argument of the LOG10 function to be a real expression. When the argument is a real expression, VAX BASIC returns a value of the same floating-point size. When the argument is not a real expression, VAX BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

# LOG10

---

## Example

```
DECLARE SINGLE exp_base_10  
exp_base_10 = LOG10(250)  
PRINT exp_base_10
```

### Output

2.39794

---

## LSET

The LSET statement assigns left-justified data to a string variable. LSET does not change the length of the destination string variable.

---

### Format

**LSET** *str-var*,... = *str-exp*

---

### Syntax Rules

1. *Str-var* is the destination string. *Str-exp* is the string value assigned to *str-var*.
2. *Str-var* cannot be a DEF function or function name unless the LSET statement is inside the multi-line DEF or function that defines the function.

---

### Remarks

1. The LSET statement treats all strings as fixed length. LSET neither changes the length of the destination string nor creates new storage. Rather, it overwrites the current storage of *str-var*.
2. If the destination string is longer than *str-exp*, LSET left-justifies *str-exp* and pads it with spaces on the right. If smaller, LSET truncates characters from the right of *str-exp* to match the length of *str-var*.

# LSET

---

## Example

```
DECLARE STRING alpha  
alpha = "ABCDE"  
LSET alpha = "FGHIJKLMN"  
PRINT alpha
```

### Output

```
FGHIJ
```

---

## MAG

The MAG function returns the absolute value of a specified expression. The returned value has the same data type as the expression.

---

### Format

*var* = **MAG** (*exp*)

---

### Syntax Rules

None.

---

### Remarks

1. The returned value is always greater than or equal to zero. The absolute value of 0 is zero. The absolute value of a positive number equals that number. The absolute value of a negative number equals that number multiplied by  $-1$ .
2. The MAG function is similar to the ABS function in that it returns the absolute value of a number. The ABS function, however, takes a floating-point argument and returns a floating-point value. The MAG function takes an argument of any numeric data type and returns a value of the same data type as the argument. DIGITAL recommends the use of the MAG function rather than the ABS and ABS% functions, because the MAG function returns a value using the data type of the argument.

# MAG

---

## Example

```
DECLARE SINGLE A  
A = -34.6  
PRINT MAG(A)
```

### **Output**

34.6

---

## MAGTAPE

The MAGTAPE function permits your program to control unformatted magnetic tape files.

### NOTE

The MAGTAPE function is supported only for compatibility with BASIC-PLUS-2. DIGITAL recommends that you do not use the MAGTAPE function for new program development.

---

### Format

*int-var1* = **MAGTAPE** (*func-code*, *int-var*, *chnl-exp*)

---

### Syntax Rules

1. *Func-code* specifies the code for the MAGTAPE function you want to perform. VAX BASIC supports only function code 3, rewind tape. Table 4-4 explains how to perform other MAGTAPE functions with VAX BASIC.
2. *Int-var* is an integer parameter for function codes 4, 5, and 6. However, because VAX BASIC supports only function code 3, *int-var* is not used and always equals zero.
3. *Chnl-exp* is a numeric expression that specifies a channel number associated with the magnetic tape file.

# MAGTAPE

**Table 4-4: MAGTAPE Functionality in VAX BASIC**

<b>Code</b>	<b>Function</b>	<b>VAX BASIC Action</b>
2	Write EOF	Close channel with the CLOSE statement
3	Rewind tape	Use the RESTORE # statement, the REWIND clause on an OPEN statement, or the MAGTAPE function
4	Skip records	Perform GET operations, ignore data until reaching desired record
5	Backspace	Rewind tape, perform GET operations, ignore data until reaching desired record
6	Set density or set parity	Use the DCL commands MOUNT/DENSITY and MOUNT/FOREIGN or the \$MOUNT system service
7	Get status	Use the RMSSTATUS function

---

## Remarks

For more information on the MAGTAPE function, see Appendix A in this manual.

---

## Example

```
I = MAGTAPE (3%,0%,2%)
```



---

## MAP

The MAP statement defines a named area of statically allocated storage called a PSECT, declares data fields in the record, and associates them with program variables.

---

### Format

**MAP** (*map-name*) { [*data-type*] *map-item* },...

*map-item*:

{	<i>num-unsubs-var</i>	}
	<i>num-array-name</i> ( [ <i>int-const1 TO</i> ] <i>int-const2</i> ,... )	
	<i>record-var</i>	
	<i>str-unsubs-var</i> [= <i>int-const</i> ]	
	<i>str-array-name</i> ( [ <i>int-const1 TO</i> ] <i>int-const2</i> ,... ) [= <i>int-const</i> ]	
	<b>FILL</b> [ ( <i>int-const</i> ) ] [= <i>int-const</i> ]	
	<b>FILL%</b> [ ( <i>int-const</i> ) ]	
	<b>FILL\$</b> [ ( <i>int-const</i> ) ] [= <i>int-const</i> ]	

---

### Syntax Rules

1. *Map-name* is global to the program and image. It cannot appear elsewhere in the program unit as a variable name.
2. *Map-name* can be from 1 through 31 characters. The first character of the name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs (\$), periods (.), or underscores (\_).
3. *Data-type* can be any VAX BASIC data type keyword or a data type defined by a RECORD statement. Data type keywords, size, range, and precision are listed in Table 1-2.

# MAP

4. When you specify a data type, all following *map-items*, including FILL items, are of that data type until you specify a new data type.
5. If you specify a dollar sign (\$) or percent sign (%) suffix character, the variable must be a string or integer data type.
6. If you do not specify a data type, a *map-item* without a suffix character (%) or (\$) takes the current default data type and size.
7. *Map-item* declares the name and format of the data to be stored.
  - *Num-unsubs-var* and *num-array-name* specify a numeric variable or a numeric array.
  - *Record-var* specifies a record instance.
  - *Str-unsubs-var* and *str-array-name* specify a fixed-length string variable or array. You can specify the number of bytes to be reserved for the variable with the *=int-const* clause. The default string length is 16.
  - The FILL, FILL%, and FILL\$ keywords allow you to reserve parts of the record buffer within or between data elements and to define the format of the storage. *Int-const* specifies the number of FILL items to be reserved. The *=int-const* clause allows you to specify the number of bytes to be reserved for string FILL items. Table 4-2 describes FILL item format and storage allocation.
  - In the applicable formats of FILL, (*int-const*) represents a repeat count, not an array subscript. FILL (*n*), for example, represents *n* elements, not *n + 1*.
8. Variable names, array names and FILL items following a data type other than STRING cannot end with a dollar sign. Variable names, array names and FILL items following a data type other than BYTE, WORD, LONG or INTEGER, cannot end with a percent sign.
9. Variables and arrays declared in a MAP statement cannot be declared elsewhere in the program by any other declarative statements.
10. When you declare an array, VAX BASIC allows you to specify both lower and upper bounds. Upper bounds are required; lower bounds are optional.
  - *Int-const2* specifies the upper bounds of the array and, when accompanied by *int-const1*, must be preceded by the keyword TO.
  - *Int-const1* must be less than or equal to *int-const2*.

- If you do not specify *int-const1*, VAX BASIC uses zero as the default lower bound.
- *Int-const1* and *int-const2* can be any combination of negative and positive values.

---

## Remarks

1. VAX BASIC does not execute MAP statements. The MAP statement allocates static storage and defines data at compilation time.
2. A program can have multiple maps with the same name. The allocation for each map overlays the others. Thus, data is accessible in many ways. The actual size of the data area is the size of the largest map. When you link your program, the size of the map area is the size of the largest map with that name.
3. *Map-items* with the same name can appear in different MAP statements with the same map name only if they match exactly in attributes such as data type, position, and so forth. If the attributes are not the same, VAX BASIC signals an error. For example:

```
MAP (ABC) LONG A, B
MAP (ABC) LONG A, C ! This MAP statement is valid
MAP (ABC) LONG B, A ! This MAP statement produces an error
MAP (ABC) WORD A, B ! This MAP statement produces an error
```

The third MAP statement causes VAX BASIC to signal the error “variable <name> not aligned in multiple references in MAP <name>”, while the fourth MAP statement generates the error “attributes of overlaid variable <name> don’t match”.

4. The MAP statement should precede any reference to variables declared in it.
5. Storage space for *map-items* is allocated in order of occurrence in the MAP statement.
6. A MAP area can be accessed by more than one program module, as long as you define the *map-name* in each module that references the MAP.

# MAP

7. A **COMMON** area and a **MAP** area with the same name specify the same storage area and are not allowed in the same program module. However, a **COMMON** in one module can reference the storage declared by a **MAP** or **COMMON** in another module.
8. Variables in a **MAP** statement are not initialized by **VAX BASIC**.
9. A map named in an **OPEN** statement's **MAP** clause is associated with that file. The file's records and record fields are defined by that map. The size of the map determines the record size for file I/O, unless the **OPEN** statement includes a **RECORDSIZE** clause.

---

## Example

```
MAP (BUF1) BYTE AGE, STRING emp_name = 20      &  
    SINGLE emp_num  
  
MAP (BUF1) BYTE FILL, STRING last_name (11) = 12, &  
    FILL = 8, SINGLE FILL
```

---

## MAP DYNAMIC

The MAP DYNAMIC statement names the variables and arrays whose size and position in a storage area can change at run time. The MAP DYNAMIC statement is used in conjunction with the REMAP statement. The REMAP statement defines or redefines the position in the storage area of variables named in the MAP DYNAMIC statement.

---

### Format

**MAP DYNAMIC** (*map-dyn-name*){[ *data-type* ]*map-item* },...

*map-dyn-name*: { *map-name*  
                  *static-str-var* }

*map-item*: { *num-unsubs-var*  
              *num-array-name* ( [ *int-const1* **TO** ] *int-const2* ,...)  
              *record-var*  
              *str-unsubs-var*  
              *str-array-name* ( [ *int-const1* **TO** ] *int-const2* ,... ) }

---

### Syntax Rules

1. *Map-dyn-name* can either be a map name or a static string variable.
  - *Map-name* is the storage area named in a MAP statement.
  - If you specify a map name, then a MAP statement with the same name must precede both the MAP DYNAMIC statement and the REMAP statement.
  - When you specify a static string variable, the string must be declared before you can specify a MAP DYNAMIC statement or a REMAP statement.

## MAP DYNAMIC

- *Static-str-var* must specify a static string variable or a string parameter variable.
  - If you specify a *static-str-var*, the following restrictions apply:
    - *Static-str-var* cannot be a string constant.
    - *Static-str-var* cannot be the same as any previously declared *map-item* in a MAP DYNAMIC statement.
    - *Static-str-var* cannot be a subscripted variable.
    - *Static-str-var* cannot be a record component.
    - *Static-str-var* cannot be a parameter declared in a DEF or DEF\* function.
2. *Map-item* declares the name and data type of the items to be stored in the storage area. All variable pointers point to the beginning of the storage area until the program executes a REMAP statement.
    - *Num-unsubs-var* and *num-array-name* specify a numeric variable or a numeric array.
    - *Record-var* specifies a record instance.
    - *Str-unsubs-var* and *str-array-name* specify a string variable or array. You cannot specify the number of bytes to be reserved for the variable in the MAP DYNAMIC statement. All string items have a fixed length of zero until the program executes a REMAP statement.
  3. When you specify an array name, VAX BASIC allows you to specify both lower and upper bounds. The upper bound is required; the lower bound is optional.
    - *Int-const1* specifies the lower bounds of the array.
    - *Int-const2* specifies the upper bounds of the array and, when accompanied by *int-const1*, must be preceded by the keyword TO.
    - *Int-const1* must be less than or equal to *int-const2*.
    - If you do not specify *int-const1*, VAX BASIC uses zero as the default lower bound.
    - *Int-const1* and *int-const2* can be either negative or positive values.
  4. *Data-type* can be any VAX BASIC data type keyword or a data type defined with a RECORD statement. Data type keywords, size, range, and precision are listed in Table 1–2 in this manual.

## MAP DYNAMIC

5. When you specify a data type, all following *map-items* are of that data type until you specify a new data type.
6. If you do not specify any data type, *map-items* take the current default data type and size.
7. *Map-items* must be separated with commas.
8. If you specify a dollar sign or percent sign suffix, the variable must be a `STRING` data type or one of the integer data types.

---

### Remarks

1. All variables and arrays declared in a `MAP DYNAMIC` statement cannot be declared elsewhere in the program by any other declarative statements.
2. The `MAP DYNAMIC` statement does not affect the amount of storage allocated to the map buffer declared in a previous `MAP` statement or the storage allocated to a static string. Until your program executes a `REMAP` statement, all variable and array element pointers point to the beginning of the `MAP` buffer or static string.
3. `VAX BASIC` does not execute `MAP DYNAMIC` statements. The `MAP DYNAMIC` statement names the variables whose size and position in the `MAP` or static string buffer can change and defines their data type.
4. Before you can specify a map name in a `MAP DYNAMIC` statement, there must be a `MAP` statement in the program unit with the same map name. Otherwise, `VAX BASIC` signals the error "Insufficient space for `MAP DYNAMIC` variables in `MAP <name>`". Similarly, before you can specify a static string variable in the `MAP DYNAMIC` statement, the string variable must be declared. Otherwise, `VAX BASIC` signals the same error message.
5. A static string variable must be either a variable declared in a `MAP` or `COMMON` statement or a parameter declared in a `SUB`, `FUNCTION`, or `PICTURE`. It cannot be a parameter declared in a `DEF` or `DEF*` function.
6. If a static string variable is the same as a map name, `VAX BASIC` uses the map name if the name appears in a `MAP DYNAMIC` statement.

# MAP DYNAMIC

7. The MAP DYNAMIC statement must lexically precede the REMAP statement or VAX BASIC signals the error "MAP variable <name> referenced before declaration."

---

## Example

```
100      MAP (MY.BUF) STRING DUMMY = 512
        MAP DYNAMIC (MY.BUF) STRING LAST, FIRST, MIDDLE,      &
        BYTE AGE, STRING EMPLOYER,                             &
        STRING CHARACTERISTICS
```



---

## **MAR**

The **MAR** function returns the current margin width of a specified channel.

---

### **Format**

*int-var* = **MAR**[%] (*chnl-exp*)

---

### **Syntax Rules**

The file associated with *chnl-exp* must be open.

---

### **Remarks**

1. If *chnl-exp* specifies a terminal and you have not set a margin width with the **MARGIN** statement, the **MAR** function returns a value of zero. If you have set a margin width, the **MAR** function returns that number.
2. The value returned by the **MAR** function is a **LONG** integer.

---

### **Example**

```
DECLARE INTEGER width
MARGIN #0, 80
width = MAR(0)
PRINT width
```

#### **Output**

80

# MARGIN

---

## MARGIN

The MARGIN statement specifies the margin width for a terminal or for records in a terminal-format file.

---

### Format

**MARGIN** [*#chnl-exp*, ] *int-exp*

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Int-exp* specifies the margin width.

---

### Remarks

1. If you do not specify a channel, VAX BASIC sets the margin on the controlling terminal.
2. The file associated with *chnl-exp* must be an open terminal-format file or terminal.
3. VAX BASIC signals the error "Illegal operation" (ERR=141) if the file associated with *chnl-exp* is not a terminal-format file.
4. If *chnl-exp* does not correspond to a terminal, and if *int-exp* is zero, VAX BASIC sets the right margin to the size specified by the RECORDSIZE clause in the OPEN statement, if the clause is present. If no RECORDSIZE clause is present, VAX BASIC sets the margin to 72 (or, in the case of channel 0, to the width of SYS\$OUTPUT).

# MARGIN

5. If *chnl-exp* is not present or if it corresponds to a terminal, and if *int-exp* is zero, VAX BASIC sets the right margin to the size specified by the RECORDSIZE clause in the OPEN statement, if the clause is present. If no RECORDSIZE clause is present, VAX BASIC sets the margin to 72.
6. VAX BASIC prints as much of a specified record as the margin setting allows on one line before going to a new line. Numeric fields are never split across lines.
7. If you specify a margin larger than the channel's record size, VAX BASIC signals an error. The default record size for a terminal or terminal format file is 132.
8. The MARGIN statement applies to the specified channel only while the channel is open. If you close the channel and then reopen it, VAX BASIC uses the default margin.

---

## Example

```
OPEN "EMP.DAT" FOR OUTPUT AS #1
MARGIN #1, 132
.
.
.
```

# MAT

---

## MAT

The MAT statement lets you implicitly create and manipulate one- and two-dimensional arrays. You can use the MAT statement to assign values to array elements, or to redimension a previously dimensioned array. You can also perform matrix arithmetic operations such as multiplication, addition, and subtraction, and other matrix operations such as transposing and inverting matrices.

---

### Format

#### Numeric Initialization

$$\text{MAT } num\text{-array} = \left\{ \begin{array}{l} \text{CON} \\ \text{IDN} \\ \text{ZER} \end{array} \right\} [ ( int\text{-exp1 } [, int\text{-exp2} ] ) ]$$

#### String Initialization

$$\text{MAT } str\text{-array} = \text{NUL\$} [ ( int\text{-exp1 } [, int\text{-exp2} ] ) ]$$

#### Array Arithmetic

$$\text{MAT } num\text{-array1} = num\text{-array2} \left[ \left\{ \begin{array}{l} + \\ - \\ * \end{array} \right\} num\text{-array3} \right]$$
$$\text{MAT } num\text{-array1} = num\text{-array2} * num\text{-array3} [ * num\text{-array4} ], \dots$$

#### Scalar Multiplication

$$\text{MAT } num\text{-array4} = ( num\text{-exp} ) * num\text{-array5}$$

#### Inversion and Transposition

$$\text{MAT } num\text{-array6} = \left\{ \begin{array}{l} \text{TRN} \\ \text{INV} \end{array} \right\} ( num\text{-array7} )$$

---

## Syntax Rules

1. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the new dimensions of an existing array.
2. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.
3. If you do not specify bounds, VAX BASIC creates the array and dimensions it to (0 TO 10) or (10 TO 10, 0 TO 10).
4. If you specify bounds, VAX BASIC creates the array with the specified bounds. If the bounds exceed (0 TO 10) or (10 TO 10, 0 TO 10), VAX BASIC signals “Redimensioned array” (ERR=105).
5. The lower bounds must be zero.

---

## Remarks

1. To perform MAT operations on arrays larger than (10,10), create the input and output arrays with the DIM statement.
2. When the array exists, the following rules apply:
  - If you specify upper bounds, VAX BASIC redimensions the array to the specified size. However, MAT operations cannot increase the total number of array elements.
  - All arrays specified with the MAT statement must have lower bounds of zero. If you supply a nonzero value, VAX BASIC signals either a compile-time or a run-time error.
  - If you do not specify bounds, VAX BASIC does not redimension the array.
  - An array passed to a subprogram and redimensioned with a MAT statement remains redimensioned when control returns to the calling program, with two exceptions:
    - When the array is within a record and is passed by descriptor
    - When the array is passed by reference

# MAT

3. You cannot use the MAT statement on arrays of more than two dimensions.
4. You cannot use the MAT statement on arrays of data type DECIMAL or on arrays named in a RECORD statement.
5. **Initialization**
  - CON sets all elements of *num-array* to 1, except those in row and column zero.
  - IDN creates an identity matrix from *num-array*. The number of rows and columns in *num-array* must be identical. IDN sets all elements to zero except those on the diagonal from *num-array*(1,1) to *num-array*(n,n), which are set to 1.
  - ZER sets all array elements to zero, except those in row and column zero.
  - NUL\$ sets all elements of a string array to the null string, except those in row and column zero.
6. **Array Arithmetic**
  - The equal sign (=) assigns the results of the specified operation to the elements in *num-array1*.
  - If *num-array3* is not specified, VAX BASIC assigns the values of *num-array2*'s elements to the corresponding elements of *num-array1*. *Num-array1* must have at least as many rows and columns as *num-array2*.
  - Use the plus sign (+) to add the elements of two arrays. *Num-array2* and *num-array3* must have identical bounds.
  - Use the minus sign (−) to subtract the elements of two arrays. *Num-array2* and *num-array3* must have identical bounds.
  - Use the asterisk (\*) to perform matrix multiplication on the elements of *num-array2* and *num-array3* and to assign the results to *num-array1*. This operation gives the dot product of *num-array2* and *num-array3*. All three arrays must be two-dimensional, and the number of columns in *num-array2* must equal the number of rows in *num-array3*. VAX BASIC redimensions *num-array1* to have the same number of rows as *num-array2* and the same number of columns as *num-array3*.

- With matrix multiplication, you can specify more than two numeric arrays. However, each array must be two-dimensional. Moreover, in each dimension, the lower bound of each array must be zero and the upper bound must be 4. You can use the graphics transformation functions, which will automatically create arrays with these dimensions. See the DRAW statement in *Programming with VAX BASIC Graphics* for more information.

#### 7. Scalar Multiplication

- VAX BASIC multiplies each element of *num-array5* by *num-exp* and stores the results in the corresponding elements of *num-array4*.

#### 8. Inversion and Transposition

- TRN transposes *num-array7* and assigns the results to *num-array6*. If *num-array7* has *m* rows and *n* columns, *num-array6* will have *n* rows and *m* columns. Both arrays must be two-dimensional.
  - You cannot transpose a matrix to itself: MAT A = TRN(A) is invalid.
  - INV inverts *num-array7* and assigns the results to *num-array6*. *Num-array7* must be a two-dimensional array that can be reduced to the identity matrix with elementary row operations. The row and column dimensions must be identical.
9. You cannot increase the number of array elements or change the number of dimensions in an array when you redimension with the MAT statement. For example, you can redimension an array with dimensions (5,4) to (4,5) or (3,2), but you cannot redimension that array to (5,5) or to (10). The total number of array elements includes those in row and column zero.
  10. If an array is named in both a DIM statement and a MAT statement, the DIM statement must lexically precede the MAT statement.
  11. MAT statements do not operate on elements in the zero element (one-dimensional arrays) or in the zero row or column (two-dimensional arrays). MAT statements use these elements to store results of intermediate calculations. Therefore, you should not depend on values in row and column zero if your program uses MAT statements.

# MAT

---

## Examples

### Example 1

```
!Numeric Initialization  
MAT CONVERT = zer(10,10)
```

### Example 2

```
!Initialization  
MAT na_me$ = NUL$(5,5)
```

### Example 3

```
!Array Arithmetic  
MAT new_int = old_int - rslt_int
```

### Example 4

```
!Scalar Multiplication  
MAT Z40 = (4.24) * Z
```

### Example 5

```
!Inversion and Transposition  
MAT Q% = INV (Z)
```



---

## MAT INPUT

The MAT INPUT statement assigns values from a terminal or terminal-format file to array elements.

---

### Format

**MAT INPUT** [*#chnl-exp*, ] { *array* [ ( *int-exp1* [, *int-exp2* ) ] },...

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. The file associated with *chnl-exp* must be an open terminal-format file or terminal. If *chnl-exp* is not specified, VAX BASIC takes data from the controlling terminal.
3. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.
4. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.

---

### Remarks

1. You cannot use the MAT INPUT statement on arrays of more than two dimensions.
2. You cannot use the MAT INPUT statement on arrays of data type DECIMAL or on arrays named in a RECORD statement.
3. All arrays specified with the MAT INPUT statement must have lower bounds of zero.

# MAT INPUT

4. If you do not specify bounds, VAX BASIC creates the array and dimensions it to (10,10).
5. If you do specify upper bounds, VAX BASIC creates the array with the specified bounds. If the bounds exceed (10) or (10,10), VAX BASIC signals "Redimensioned array" (ERR=105).
6. To use the MAT INPUT statement with arrays larger than (10,10), create the input and output arrays with the DIM statement.
7. When the array exists, the following rules apply:
  - If you specify bounds, VAX BASIC redimensions the array to the specified size. However, MAT INPUT cannot increase the total number of array elements.
  - If you do not specify bounds, VAX BASIC does not redimension the array.
8. The MAT INPUT statement prompts with a question mark on terminals open on channel #0 only unless a SET NO PROMPT statement has been executed. See the description of the SET PROMPT statement for more information.
9. Use commas to separate data elements and a line terminator to end the input of data. Use an ampersand (&) before the line terminator to continue data over more than one line.
10. The MAT INPUT statement assigns values by row. For example, it assigns values to all elements in row 1 before beginning row 2.
11. The MAT INPUT statement assigns the row number of the last data element transferred into the array to the system variable NUM.
12. The MAT INPUT statement assigns the column number of the last data element transferred into the array to the system variable NUM2.
13. If there are fewer elements in the input data than there are array elements, VAX BASIC does not change the remaining array elements.
14. If there are more data elements in the input stream than there are array elements, VAX BASIC ignores the excess.
15. Row zero and column zero are not changed.
16. For information about graphics input, see the MAT LOCATE and the MAT GET statements in *Programming with VAX BASIC Graphics*.

---

## Example

```
MAT INPUT XYZ(5,5)
MAT PRINT XYZ;
```

### Output

```
? 1,2,3,4,5
 1 2 3 4 5
 0 0 0 0 0
 0 0 0 0 0
 0 0 0 0 0
 0 0 0 0 0
```

# MAT LINPUT

---

## MAT LINPUT

The MAT LINPUT statement receives string data from a terminal or terminal-format file and assigns it to string array elements.

---

### Format

**MAT LINPUT** [*#chnl-exp*, ] { *str-array* [ ( *int-exp1* [, *int-exp2* ] ) ] }, ...

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file or terminal. It must be immediately preceded by a number sign (#).
2. The file associated with *chnl-exp* must be an open terminal-format file or terminal. If a channel is not specified, VAX BASIC takes data from the controlling terminal.
3. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.
4. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.

---

### Remarks

1. You cannot use the MAT LINPUT statement on arrays of more than two dimensions.
2. You cannot use the MAT LINPUT statement on arrays of data type other than STRING or on arrays named in a RECORD statement.
3. If you do not specify bounds, VAX BASIC creates the array and dimensions it to (10,10).

## MAT LINPUT

4. If you do specify upper bounds, VAX BASIC creates the array with the specified bounds. If the bounds exceed (10) or (10,10), VAX BASIC signals "Redimensioned array" (ERR=105).
5. All arrays specified with the MAT LINPUT statement must have lower bounds of zero.
6. To use MAT LINPUT with arrays larger than (10,10), create the input and output arrays with the DIM statement.
7. When the array exists, the following rules apply:
  - If you specify bounds, VAX BASIC redimensions the array to the specified size. However, MAT LINPUT cannot increase the total number of array elements.
  - If you do not specify bounds, VAX BASIC does not redimension the array.
8. For terminals open on channel zero only, the MAT LINPUT statement prompts with a question mark (unless a SET NO PROMPT statement has been executed) for each string array element, starting with element (1,1). VAX BASIC assigns values to all elements of row 1 before beginning row 2.
9. The MAT LINPUT statement assigns the row number of the last data element transferred into the array to the system variable NUM.
10. The MAT LINPUT statement assigns the column number of the last data element transferred into the array to the system variable NUM2.
11. Typing only a line terminator in response to the question mark prompt causes VAX BASIC to assign a null string to that string array element.
12. MAT LINPUT does not change row and column zero.

# MAT LINPUT

---

## Example

```
DIM cus_rec$(3,3)
MAT LINPUT cus_rec$(2,2)
PRINT cus_rec$(1,1)
PRINT cus_rec$(1,2)
PRINT cus_rec$(2,1)
PRINT cus_rec$(2,2)
```

### Output

```
? Babcock
? Santani
? Lloyd
? Kelly
Babcock
Santani
Lloyd
Kelly
```

---

## MAT PRINT

The MAT PRINT statement prints the contents of a one- or two-dimensional array on your terminal or assigns the value of each array element to a record in a terminal-format file.

---

### Format

**MAT PRINT** [#*chnl-exp*, ] { array [ ( *int-exp1* [, *int-exp2* ] ) ] [ ; ] }...

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file or terminal. It must be immediately preceded by a number sign (#).
2. The file associated with *chnl-exp* must be an open terminal-format file or terminal. If you do not specify a channel, VAX BASIC takes data from the controlling terminal.
3. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.
4. The separator (comma or semicolon) determines the output format for the array:
  - If you use a comma, VAX BASIC prints each array element in a new print zone and starts each row on a new line.
  - If you use a semicolon, VAX BASIC separates each array element with a space and starts each row on a new line.
  - If you do not use a separator character, VAX BASIC prints each array element on its own line.

# MAT PRINT

---

## Remarks

1. You cannot use the MAT PRINT statement on arrays of more than two dimensions.
2. You cannot use the MAT PRINT statement on arrays of data type DECIMAL or on arrays named in a RECORD statement.
3. When you use the MAT PRINT statement to print more than one array, each array name except the last must be followed with either a comma or a semicolon. VAX BASIC prints a blank line between arrays.
4. If the array does not exist, the following rules apply:
5. All arrays specified with the MAT PRINT statement must have lower bounds of zero.
  - If you do not specify bounds, VAX BASIC creates the array and dimensions it to (10,10).
  - If you specify upper bounds, VAX BASIC creates the array with the specified bounds. If the bounds exceed (10) or (10,10), VAX BASIC prints the elements (10) or (10,10), and signals "Subscript out of range" (ERR=55).
6. When the array exists, the following rules apply:
  - If the specified bounds are smaller than the maximum bounds of a dimensioned array, VAX BASIC prints a subset of the array, but does not redimension the array. For example, if you use the DIM statement to dimension A(20,20), and then MAT PRINT A(2,2), VAX BASIC prints elements (1,1), (1,2), (2,1), and (2,2) only; array A(20,20) does not change.
  - If you do not specify bounds, VAX BASIC prints the entire array.
7. The MAT PRINT statement does not print elements in row or column zero.
8. The MAT PRINT statement cannot redimension an array.



---

## Example

```
DIM cus_rec$(3,3)
MAT LINPUT cus_rec$(2,2)
MAT PRINT cus_rec$(2,2)
```

### Output

```
? Babcock
? Santani
? Lloyd
? Kelly
Babcock
Santani
Lloyd
Kelly
```

## MAT READ

---

## MAT READ

The MAT READ statement assigns values from DATA statements to array elements.

---

### Format

**MAT READ** { *array* [ ( *int-exp1* [, *int-exp2* ] ) ] },...

---

### Syntax Rules

1. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.
  2. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.
- 

### Remarks

1. If you do not specify bounds, VAX BASIC creates the array and dimensions it to (10) or (10,10).
2. If you specify bounds, VAX BASIC creates the array with the specified bounds. If the bounds exceed (10) or (10,10), VAX BASIC signals "Redimensioned array" (ERR=105).
3. To read arrays larger than (10,10), create the array with the DIM statement.
4. All arrays specified with the MAT statement must have lower bounds of zero.

## MAT READ

5. When the array exists, the following rules apply:
  - If you specify upper bounds, VAX BASIC redimensions the array to the specified size. However, MAT READ cannot increase the total number of array elements.
  - If you do not specify bounds, VAX BASIC does not redimension the array.
6. All the DATA statements must be in the same program unit as the MAT READ statement.
7. The MAT READ statement assigns data items by row. For example, it assigns data items to all elements in row 1 before beginning row 2.
8. The MAT READ statement does not read elements into row or column zero.
9. The MAT READ statement assigns the row number of the last data element transferred into the array to the system variable, NUM.
10. The MAT READ statement assigns the column number of the last data element transferred into the array to the system variable, NUM2.
11. If you use MAT READ for an existing array without specifying bounds, VAX BASIC does not redimension the array. If you use MAT READ for an existing array and specify bounds, VAX BASIC redimensions the array.
12. You cannot use the MAT READ statement on arrays of more than two dimensions.
13. You cannot use the MAT READ statement on arrays of data type DECIMAL or on arrays named in a RECORD statement.

# MAT READ

---

## Example

```
MAT READ A(3,3)
MAT READ B(3,3)
PRINT
PRINT "Matrix A"
PRINT
MAT PRINT A;
PRINT
PRINT "Matrix B"
PRINT
MAT PRINT B;
DATA 1,2,3,4,5,6
```

### Output

Matrix A

```
1 2 3
4 5 6
0 0 0
```

Matrix B

```
0 0 0
0 0 0
0 0 0
```

---

## **MAX**

The **MAX** function compares the values of two or more numeric expressions and returns the highest value.

---

### **Format**

*num-var* = **MAX** ( *num-exp1*, *num-exp2* [ , *num-exp3* ,...])

---

### **Syntax Rules**

VAX BASIC allows you to specify up to eight numeric expressions.

---

### **Remarks**

1. If you specify values with different data types, VAX BASIC performs data type conversions to maintain precision.
2. VAX BASIC returns a function result whose data type is compatible with the values you supply.

# MAX

---

## Example

```
DECLARE REAL John_grade, &
           Bob_grade, &
           Joe_grade, &
           highest_grade
INPUT "John's grade";John_grade
INPUT "Bob's grade";Bob_grade
INPUT "Joe's grade";Joe_grade
highest_grade = MAX(John_grade, Bob_grade, Joe_grade)
PRINT "The highest grade is";highest_grade
```

### Output

```
John's grade? 90
Bob's grade? 95
Joe's grade? 79
The highest grade is 95
```

---

## MID\$

MID\$ can be used either as a statement or as a function. The MID\$ statement performs substring insertion into a string. The MID\$ function extracts a specified substring from a string expression.

---

### Format

#### MID\$ statement

**MID[\$]** (*str-var*, *int-exp1* [, *int-exp2* ]) = *str-exp*

#### MID\$ function

*str-var* = **MID[\$]** (*str-exp*, *int-exp1*, *int-exp2*)

---

### Syntax Rules

1. *Int-exp1* specifies the position of the substring's first character.
2. *Int-exp2* specifies the length of the substring.

---

### Remarks

1. If *int-exp1* is less than 1, VAX BASIC assumes a starting character position of 1.
2. If *int-exp2* is less than or equal to zero, VAX BASIC assumes a length of zero.
3. If you specify a floating-point expression for *int-exp1* or *int-exp2*, VAX BASIC truncates it to a LONG integer.

# MID\$

## 4. MID\$ statement

- The MID\$ statement replaces a specified portion of *str-var* with *str-exp*.
- If *int-exp1* is greater than the length of *str-var*, *str-var* remains unchanged.
- The length of *str-var* does not change regardless of the value of *int-exp2*.
- If the optional *int-exp2* is not specified, VAX BASIC assumes *int-exp2* to be the length of *str-exp* minimized by the length of *str-var* minus *int-exp1*. For example:

```
A$ = "ABCDEFGH"  
MID$ (A$, 3) = "123456789"  
PRINT A$
```

### Output

```
"AB12345"
```

- If *int-exp2* is less than or equal to zero, *str-var* remains unchanged.
- If *int-exp2* is greater than the length of *str-var*, VAX BASIC assumes *int-exp2* to be equal to the length of *str-var*.
- *Int-exp2* is always minimized against the length of *str-var* minus *int-exp1*.

## 5. MID\$ function

- The MID\$ function extracts a substring from *str-exp* and stores it in *str-var*.
- If *int-exp1* is greater than the length of *str-exp*, MID\$ returns a null string.
- If *int-exp2* is greater than the length of *str-exp*, VAX BASIC returns the string that begins at *int-exp1* and includes all characters remaining in *str-exp*.
- If *int-exp2* is less than or equal to zero, MID\$ returns a null string.



---

## Examples

### Example 1

```
!MID$ Function
DECLARE STRING old_string, new_string
old_string = "ABCD"
new_string = MID$(old_string,1,3)
PRINT new_string
```

### Output

ABC

### Example 2

```
!MID$ Statement
DECLARE STRING old_string, replace_string
old_string = "ABCD"
replace_string = "123"
PRINT old_string
MID$(old_string,1,3) = replace_string
PRINT old_string
```

### Output

ABCD  
123D

# MIN

---

## MIN

The MIN function compares the values of two or more numeric expressions and returns the smallest value.

---

### Format

*num-var* = **MIN** ( *num-exp1*, *num-exp2* [ , *num-exp3* ,... ] )

---

### Syntax Rules

VAX BASIC allows you to specify up to eight numeric expressions.

---

### Remarks

1. If you specify values with different data types, VAX BASIC performs data type conversions to maintain precision.
2. VAX BASIC returns a function result whose data type is compatible with the values you supply.

---

## Example

```
DECLARE REAL John_grade, &
           Bob_grade, &
           Joe_grade, &
           lowest_grade
INPUT "John's grade";John_grade
INPUT "Bob's grade";Bob_grade
INPUT "Joe's grade";Joe_grade
lowest_grade = MIN(John_grade, Bob_grade, Joe_grade)
PRINT "The lowest grade is";lowest_grade
```

### Output

```
John's grade? 95
Bob's grade? 100
Joe's grade? 84
The lowest grade is 84
```

# MOD

---

## MOD

The MOD function divides a numeric value by another numeric value and returns the remainder.

---

### Format

*num-var* = **MOD** ( *num-exp1*, *num-exp2* )

---

### Syntax Rules

Num-exp1 is divided by num-exp2.

---

### Remarks

1. If you specify values with different data types, VAX BASIC performs data type conversions to maintain precision.
2. VAX BASIC returns a function result whose data type is compatible with the values you supply.
3. The function result is either a positive or negative value, depending on the value of the first numeric expression. For example, if the first numeric expression is negative, then the function result will also be negative.

---

**Example**

```
DECLARE REAL A,B  
A = 500  
B = MOD(A,70)  
PRINT "The remainder equals";B
```

**Output**

The remainder equals 10

# MOVE

---

## MOVE

The MOVE statement transfers data between a record buffer and a list of variables.

---

### Format

**MOVE** { **TO**  
**FROM** } #*chnl-exp*, *move-item*,...

*move-item*: { *num-var*  
*num-array* ( [,]... )  
*str-var* [ = *int-exp* ]  
*str-array* ( [,]... ) [ = *int-exp* ]  
[ *data-type* ] **FILL** [ ( *int-exp* ) ] [ = *int-const* ]  
**FILL%** [ ( *int-exp* ) ]  
**FILL\$** [ ( *int-exp* ) ] [ = *int-exp* ] }

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Move-item* specifies the variable or array to which or from which data is to be moved.
3. Parentheses indicate the number of dimensions in a numeric array. The number of dimensions is equal to the number of commas plus 1. Empty parentheses indicate a one-dimensional array, one comma indicates a two-dimensional array, and so on.

# MOVE

4. *Str-var* and *str-array* specify a fixed length string variable or array. Parentheses indicate the number of dimensions in a string array. The number of dimensions is equal to the number of commas plus 1. You can specify the number of bytes to be reserved for the variable or array elements with the *=int-exp* clause. The default string length for a MOVE FROM statement is 16. For a MOVE TO statement, the default is the string's length.
5. The FILL, FILL%, and FILL\$ keywords allow you to transfer fill items of a specific data type. Table 4-2 shows FILL item formats, representations, and storage requirements.
  - If you specify a data type before the FILL keyword, the fill is of that data type. If you do not specify a data type, the fill is of the default data type. *Data-type* can be any VAX BASIC data type keyword or a data type defined by a RECORD statement. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.
  - FILL items following a data type other than STRING cannot end with a dollar sign. FILL items following a data type other than BYTE, WORD, LONG or INTEGER, cannot end with a percent sign.
  - *Int-exp* specifies the number of FILL items to be moved.
  - FILL% indicates integer fill. FILL\$ indicates string fill. The *=int-exp* clause specifies the number of bytes to be moved for string FILL items.
  - In the applicable formats of FILL, (*int-exp*) represents a repeat count, not an array subscript. FILL (*n*), for example, represents *n* elements, not *n* + 1.
6. You cannot use an expression or function reference as a *move-item*.

---

## Remarks

1. Before a MOVE FROM statement can execute, the file associated with *chnl-exp* must be open and there must be a record in the record buffer.

# MOVE

2. A MOVE statement neither transfers data to or from external devices, nor invokes the VAX Record Management Services (RMS). Instead, it transfers data between user areas. Thus, a record should first be fetched with the GET statement before you use a MOVE FROM statement, and a MOVE TO statement should be followed by a PUT or UPDATE statement that writes the record to a file.
3. MOVE FROM transfers data from the record buffer to the *move-item*.
4. MOVE TO transfers data from the *move-item* to the record buffer.
5. The MOVE statement does not affect the record buffer's size. If a MOVE statement partially fills a buffer, the rest of the buffer is unchanged. If there is more data in the variable list than in the buffer, VAX BASIC signals "MOVE overflows buffer" (ERR=161).
6. Each MOVE statement to or from a channel transfers data starting at the beginning of the buffer. For example:

```
MOVE FROM #1%, I%, A$ = I%
```

In this example, VAX BASIC assigns the first value in the record buffer to *I%*; the value of *I%* is then used to determine the length of *A\$*.

7. If a MOVE statement operates on an entire array, the following conditions apply:
  - VAX BASIC transfers elements of row and column zero (in contrast to the MAT statements).
  - The storage size of the array elements and the size of the array determine the amount of data moved. A MOVE statement that transfers data from the buffer to a longword integer array transfers the first four bytes of data into the first element (for example, (0,0)), the next four bytes of data into element (0,1), and so on.
8. If the MOVE TO statement specifies an explicit string length, the following restrictions apply:
  - If the string is equal to or longer than the explicit string length, VAX BASIC moves only the specified number of characters into the buffer.
  - If the string is shorter than the explicit string length, VAX BASIC moves the entire string and pads it with spaces to the specified length.
9. VAX BASIC does not check the validity of data during the MOVE operation.



---

## Example

```
MOVE FROM #4%, RUNS%, HITS%, ERRORS%, RBI%, BAT_AVERAGE  
MOVE TO #9%, FILL$ = 10%, A$ = 10%, B$ = 30%, C$ = 2%
```

## NAME...AS

---

## NAME...AS

The NAME...AS statement renames the specified file.

---

### Format

**NAME** *file-spec1* **AS** *file-spec2*

---

### Syntax Rules

1. *File-spec1* and *file-spec2* must be string expressions.
2. There is no default file type in *file-spec1* or *file-spec2*. If the file to be renamed has a file type, *file-spec1* must include both the file name and the file type.
3. If you specify only a file name, VAX BASIC searches for a file with no file type. If you do not specify a file type for *file-spec2*, VAX BASIC names the file, but does not assign a file type.
4. You can include a directory name but not a device name. If you specify a directory name with *file-spec2*, the file will be placed in the specified directory. If you do not specify a directory name the default is the current directory.
5. File version numbers are optional. VAX BASIC renames the highest version of *file-spec1* if you do not specify a version number.

---

## Remarks

1. If the file specified by *file-spec1* does not exist, VAX BASIC signals “Can’t find file or account” (ERR=5).
2. If you use the NAME...AS statement on an open file, VAX BASIC does not rename the file until it is closed.
3. You cannot use the NAME...AS statement to move a file between devices. You can only change the directory, name, type, or version number.

---

## Example

```
$ Directory USER$$DISK:[BASIC_PROG]
Directory USER$$DISK:[BASIC_PROG]

FIRST_PROG.BAS;1
Total of 1 file.
$ BASIC

VAX BASIC V3.4
Ready

NAME "FIRST_PROG.BAS" AS "SECOND_PROG.BAS"
Ready

EXIT

$ Directory USER$$DISK:[BASIC_PROG]
Directory USER$$DISK:[BASIC_PROG]

SECOND_PROG.BAS;1
Total of 1 file.
```

# NEXT

---

## NEXT

The NEXT statement marks the end of a FOR, UNTIL, or WHILE loop.

---

### Format

**NEXT** [ *num-unsubs-var* ]

---

### Syntax Rules

1. *Num-unsubs-var* is required in a FOR...NEXT loop and must correspond to the *num-unsubs-var* specified in the FOR statement.
2. *Num-unsubs-var* is not allowed in an UNTIL or WHILE loop.
3. *Num-unsubs-var* must be a numeric, unsubscripted variable.

---

### Remarks

Each NEXT statement must have a corresponding FOR, UNTIL, or WHILE statement or VAX BASIC signals an error.

---

## Example

```
PROGRAM calculating_pay
DECLARE INTEGER no_hours, &
          SINGLE weekly_pay, minimum_wage
minimum_wage = 3.65
no_hours = 40
WHILE no_hours > 0
  INPUT "Enter the number of hours you intend to work this week";no_hours
  weekly_pay = no_hours * minimum_wage
  PRINT "If you worked";no_hours;"hours, your pay would be";weekly_pay
NEXT
END PROGRAM
```

### Output

```
Enter the number of hours you intend to work this week? 35
If you worked 35 hours, your pay would be 127.75
Enter the number of hours you intend to work this week? 23
If you worked 23 hours, your pay would be 83.95
Enter the number of hours you intend to work this week? 0
If you worked 0 hours your pay would be 0
```

# NOECHO

---

## NOECHO

The NOECHO function disables echoing of input on a terminal.

---

### Format

*int-var* = NOECHO (*chnl-exp*)

---

### Syntax Rules

*Chnl-exp* must specify a terminal.

---

### Remarks

1. If you specify NOECHO, VAX BASIC accepts characters typed on the terminal as input, but the characters do not echo on the terminal.
2. The NOECHO function is the complement of the ECHO function; NOECHO disables the effect of ECHO and vice versa.
3. NOECHO always returns a value of zero.

---

## Example

```
DECLARE INTEGER Y,          &
          STRING pass_word
Y = NOECHO(0)
INPUT "Enter your password";pass_word
IF pass_word = "DARLENE" THEN PRINT "Confirmed"
Y = ECHO(0)
```

### Output

```
Enter your password?
Confirmed
```

# NOMARGIN

---

## NOMARGIN

The **NOMARGIN** statement removes the right margin limit set with the **MARGIN** statement for a terminal or a terminal-format file.

---

### Format

**NOMARGIN** [*#chnl-exp*]

---

### Syntax Rules

*Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).

---

### Remarks

1. When you specify **NOMARGIN**, the right margin is set to 132.
2. *Chnl-exp*, if specified, must be an open terminal-format file or a terminal.
3. If you do not specify a channel, VAX BASIC sets the margin on the controlling terminal to 132.
4. The **NOMARGIN** statement applies to the specified channel only while the channel is open. If you close the channel and then reopen it, VAX BASIC uses the default margin of 72.



---

## Example

```
OPEN "EMP.DAT" FOR OUTPUT AS #1
NOMARGIN #1
.
.
.
```

# NUM

---

## NUM

The NUM function returns the row number of the last data element transferred into an array by a MAT I/O statement.

---

### Format

*int-var* = NUM

---

### Syntax Rules

None.

---

### Remarks

1. NUM returns a value of zero if it is invoked before VAX BASIC has executed any MAT I/O statements.
2. For a two-dimensional array, NUM returns an integer specifying the row number of the last data element transferred into the array. For a one-dimensional array, NUM returns the number of elements entered.
3. The value returned by the NUM function is an integer of the default size.

---

## Example

```
OPEN "STU_ACCT" FOR INPUT AS #2
DIM stu_rec$(3,3)
MAT INPUT #2, stu_rec$
PRINT "Row count =";NUM
PRINT "Column number =";NUM2
```

### Output

```
Row count = 1
Column number = 1
```

## NUM2

---

## NUM2

The NUM2 function returns the column number of the last data element transferred into an array by a MAT I/O statement.

---

### Format

*int-var* = NUM2

---

### Syntax Rules

None.

---

### Remarks

1. NUM2 returns a value of zero if it is invoked before VAX BASIC has executed any MAT I/O statements or if the last array element transferred was in a one-dimensional list.
2. The NUM2 function returns an integer specifying the column number of the last data element transferred into an array.
3. The value returned by the NUM2 function is an integer of the default size.

---

## Example

```
OPEN "STU_ACCT" FOR INPUT AS #2
DIM stu_rec$(3,3)
MAT INPUT #2, stu_rec$
PRINT "Row count =";NUM
PRINT "Column number =";NUM2
```

### Output

```
Row count = 1
Column number = 1
```

# NUM\$

---

## NUM\$

The NUM\$ function evaluates a numeric expression and returns a string of characters in PRINT statement format, with leading and trailing spaces.

---

### Format

*str-var* = NUM\$ (*num-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. If *num-exp* is positive, the first character in the string expression is a space. If *num-exp* is negative, the first character is a minus sign (–).
2. The NUM\$ function does not include trailing zeros in the returned string. If all digits to the right of the decimal point are zeros, NUM\$ omits the decimal point as well.
3. When *num-exp* is a floating-point variable and has an integer portion of six decimal digits or less (for example, 1234.567), VAX BASIC rounds the number to six digits (1234.57). If *num-exp* has seven decimal digits or more, VAX BASIC rounds the number to six digits and prints it in E format.
4. When *num-exp* is between 0.1 and 1 and contains more than six digits, VAX BASIC rounds it to six digits. When *num-exp* is smaller than 0.1, VAX BASIC rounds it to six digits and prints it in E format.
5. If *num-exp* is a longword integer, the returned string can have up to 10 digits.

6. If *num-exp* is a DECIMAL value, the returned string can have up to 31 digits.
7. The last character in the returned string is a space.

---

## Example

```
DECLARE STRING number  
number = NUM$(34.5500/31.8)  
PRINT number
```

### Output

```
1.08648
```

## NUM1\$

---

## NUM1\$

The NUM1\$ function changes a numeric expression to a numeric character string without leading and trailing spaces and without rounding.

---

### Format

*str-var* = NUM1\$ (*num-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. The NUM1\$ function returns a string consisting of numeric characters and a decimal point that corresponds to the value of *num-exp*. Leading and trailing spaces are not included in the returned string.
2. The NUM1\$ function returns a maximum of
  - 3 digits for BYTE integers
  - 5 digits for SINGLE floating-point numbers and WORD integers
  - 10 digits for LONG integers
  - 16 digits for DOUBLE floating-point numbers
  - 15 digits for GFLOAT floating-point numbers
  - 33 digits for HFLOAT floating-point numbers
  - 31 digits for DECIMAL numbers
3. The NUM1\$ function does not support E-format notation.



---

**Example**

```
DECLARE STRING number  
number = NUM1$(PI/2)  
PRINT number
```

**Output**

1.5708

# ON ERROR GO BACK

---

## ON ERROR GO BACK

Under certain conditions, an ON ERROR GO BACK statement executed in a subprogram or DEF function transfers control to the calling program.

### NOTE

The ON ERROR GO BACK statement is supported for compatibility with other DIGITAL BASICs. For new program development, DIGITAL recommends that you use WHEN blocks.

---

### Format

**{ ONERROR  
ON ERROR } GO BACK**

---

### Syntax Rules

The ON ERROR GO BACK statement is illegal inside a protected region or within an attached or detached handler. Use the EXIT HANDLER statement instead.

---

### Remarks

1. If there is no error outstanding, execution of an ON ERROR GO BACK statement causes subsequent errors to return control to the calling program's error handler.
2. If there is an error outstanding, execution of an ON ERROR GO BACK statement immediately transfers control to the calling program's error handler.

## ON ERROR GO BACK

3. By default, DEF functions and subprograms re-signal errors to the calling program.
4. The ON ERROR GO BACK statement remains in effect until the program unit completes execution, until VAX BASIC executes another ON ERROR statement, or until VAX BASIC enters a protected region.
5. An ON ERROR GO BACK statement executed in the main program is equivalent to an ON ERROR GOTO 0 statement.
6. If a main program calls a subprogram named SUB1, and SUB1 calls the subprogram named SUB2, an ON ERROR GO BACK statement executed in SUB2 transfers control to SUB1's error handler when an error occurs in SUB2. If SUB1 also has executed an ON ERROR GO BACK statement, VAX BASIC transfers control to the main program's error handling routine.
7. For current program development, see the WHEN ERROR statement.
8. DIGITAL does not recommend that you mix ON ERROR statements with protected regions in the same program unit. For more information, see the *VAX BASIC User Manual*.

---

### Example

```
IF ERR = 11
  THEN
    RESUME err_hand
  ELSE
    ON ERROR GO BACK
END IF
```

## ON ERROR GOTO

---

# ON ERROR GOTO

The ON ERROR GOTO statement transfers program control to a specified line or label in the current program unit when an error occurs under certain conditions.

### NOTE

The ON ERROR GOTO statement is supported for compatibility with other DIGITAL BASICs. For new program development, DIGITAL recommends that you use WHEN blocks.

---

### Format

$$\left\{ \begin{array}{l} \text{ONERROR} \\ \text{ON ERROR} \end{array} \right\} \left\{ \begin{array}{l} \text{GO TO} \\ \text{GOTO} \end{array} \right\} \textit{target}$$

---

### Syntax Rules

1. You cannot specify an ON ERROR GOTO statement within a protected region or handler.
2. *Target* must be a valid VAX BASIC line number or label and must exist in the same program unit as the ON ERROR GOTO statement.
3. If an ON ERROR GOTO statement is in a DEF function, *target* must also be in that function definition.

---

## Remarks

1. VAX BASIC transfers program control to a specified line number or label under two conditions:
  - If an error does not occur within the protected region of WHEN block.
  - If an error occurs within the protected region of a WHEN block and was propagated by the handler associated with the WHEN block.
2. Execution of an ON ERROR GOTO statement causes subsequent errors to transfer control to the specified target.
3. The ON ERROR GOTO statement remains in effect until the program unit completes execution or until VAX BASIC executes another ON ERROR statement.
4. VAX BASIC does not allow recursive error handling. If a second error occurs during execution of an error-handling routine, control passes to the VAX BASIC error handler and the program stops executing.
5. For current program development, see the WHEN ERROR statement.
6. DIGITAL does not recommend that you mix ON ERROR statements with protected regions within the same program unit. For more information, see the *VAX BASIC User Manual*.

# ON ERROR GOTO

---

## Example

```
SUB LIST (STRING A)
DECLARE STRING B
ON ERROR GOTO err_block
OPEN A FOR INPUT AS FILE #1
Input_loop:
  LINPUT #1, B
  PRINT B
  .
  .
  .
  GOTO Input_loop
err_block:
  IF (ERR=11%)
  THEN
    CLOSE #1%
    RESUME done
  ELSE
    ON ERROR GOTO 0
  END IF
done:
END SUB
```

---

## ON ERROR GOTO 0

The ON ERROR GOTO 0 statement disables ON ERROR error handling and passes control to the VAX BASIC error handler when an error occurs.

### NOTE

The ON ERROR GOTO 0 statement is supported for compatibility with other DIGITAL BASICs. For new program development, DIGITAL recommends that you use WHEN blocks.

---

### Format

$$\left\{ \begin{array}{l} \text{ON ERROR} \\ \text{ONERROR} \end{array} \right\} \left\{ \begin{array}{l} \text{GO TO} \\ \text{GOTO} \end{array} \right\} 0$$

---

### Syntax Rules

VAX BASIC does not allow you to specify an ON ERROR GOTO 0 statement within an attached or detached handler or within a protected region.

---

### Remarks

1. If an error is outstanding, execution of an ON ERROR GOTO 0 statement immediately transfers control to the VAX BASIC error handler. The VAX BASIC error handler will report the error and exit the program.
2. If there is no error outstanding, execution of an ON ERROR GOTO 0 statement causes subsequent errors to transfer control to the VAX BASIC error handler.

## ON ERROR GOTO 0

3. When an `ON ERROR GOTO 0` statement is executed, control is transferred to the VAX BASIC error handler if an error did not occur within the protected region of a `WHEN` block.
4. If an error occurs within the protected region of a `WHEN` block and was propagated by the handler associated with the `WHEN` block, VAX BASIC transfers control to the specified line number or label contained in the subprogram or `DEF`.
5. For current program development, see the `WHEN ERROR` statement.
6. DIGITAL does not recommend that you mix `ON ERROR` statements with attached or detached handlers within the same program unit. For more information, see the *VAX BASIC User Manual*.

---

### Example

```
ON ERROR GOTO err_routine
FOR I = 1% TO 10%
    PRINT "Please type a number"
    INPUT A
NEXT I
err_routine:
IF ERR = 50
    THEN
        RESUME
    ELSE
        ON ERROR GOTO 0
END IF
```

### Output

```
Please type a number
? [CTRL/Z]

%BAS-F-ILLUSADEV, Illegal usage for device
-BAS-I-ON_CHAFIL, on channel 0 for file SYS$INPUT:[TUTTI]SYSINPUT.DAT;
                                     at user PC 00000632
-RMS-F-DEV, error in device name or inappropriate device type for operation
-BAS-I-FROLINMOD, from line 10 in module BADUSER
```



---

## ON...GOSUB

The ON...GOSUB statement transfers program control to one of several subroutines, depending on the value of a control expression.

---

### Format

**ON** *int-exp* **GOSUB** *target*, ... [ **OTHERWISE** *target* ]

---

### Syntax Rules

1. *Int-exp* determines which target VAX BASIC selects as the GOSUB argument. If *int-exp* equals 1, VAX BASIC selects the first target. If *int-exp* equals 2, VAX BASIC selects the second target, and so on.
2. *Target* must be a valid VAX BASIC line number or label and must exist in the current program unit.

---

### Remarks

1. Control cannot be transferred into a statement block (such as FOR...NEXT, UNTIL...NEXT, WHILE...NEXT, DEF...END DEF, SELECT...END SELECT, WHEN...END WHEN, or HANDLER...END HANDLER).
2. If there is an OTHERWISE clause, and if *int-exp* is less than 1 or greater than the number of targets in the list, VAX BASIC selects the target of the OTHERWISE clause.
3. If there is no OTHERWISE clause, and if *int-exp* is less than 1 or greater than the number of targets in the list, VAX BASIC signals "ON statement out of range" (ERR=58).
4. If a target specifies a nonexecutable statement, VAX BASIC transfers control to the first executable statement that lexically follows the target.

## ON...GOSUB

5. You can only use the ON...GOSUB statement inside a handler if all the targets are contained within the handler.
6. If you fail to handle an exception that occurs while an ON...GOSUB statement in the body of a subroutine is executing, the exception is handled by the default error handler. The exception is not handled by any WHEN block surrounding the ON...GOSUB statement that invoked the subroutine.
7. You can specify the ON...GOSUB statement inside a WHEN block if the ON...GOSUB target is in the same protected region, an outer protected region, or in a non-protected region.
8. You cannot specify an ON...GOSUB statement inside a WHEN block if the ON...GOSUB target already resides in another protected region that does not contain the most current protected region.
9. The target cannot be more than 32767 bytes away from the ON...GOSUB statement.

---

### Example

```
100  INPUT "Please enter 1, 2 or 3"; A%
      ON A% GOSUB 1000, 2000, 3000 OTHERWISE err_routine
      GOTO done

1000  PRINT "That was a 1"
      RETURN
2000  PRINT "That was a 2"
      RETURN
3000  PRINT "That was a 3"
      RETURN

err_routine:
      PRINT "Out of range:"
      RETURN
done:
      END PROGRAM
```

---

## ON...GOTO

The ON...GOTO statement transfers program control to one of several lines or targets, depending on the value of a control expression.

---

### Format

**ON** *int-exp* { **GO TO**  
**GOTO** } *target* ,... [ **OTHERWISE** *target* ]

---

### Syntax Rules

1. *Int-exp* determines which target VAX BASIC selects as the GOTO argument. If *int-exp* equals 1, VAX BASIC selects the first target. If *int-exp* equals 2, VAX BASIC selects the second target, and so on.
2. *Target* must be a valid VAX BASIC line number or a label and must exist in the current program unit.

---

### Remarks

1. Control cannot be transferred into a statement block (such as FOR...NEXT, UNTIL...NEXT, WHILE...NEXT, DEF...END DEF, SELECT...END SELECT, WHEN...END WHEN, or HANDLER...END HANDLER).
2. If there is an OTHERWISE clause, and if *int-exp* is less than one or greater than the number of targets in the list, VAX BASIC transfers control to the target of the OTHERWISE clause.
3. If there is no OTHERWISE clause, and if *int-exp* is less than 1 or greater than the number of line numbers in the list, VAX BASIC signals "ON statement out of range" (ERR=58).

## ON...GOTO

4. If a target specifies a nonexecutable statement, VAX BASIC transfers control to the first executable statement that lexically follows the target.
5. You can only use the ON...GOTO statement inside a handler if all the targets are contained within the handler.
6. You can specify the ON...GOTO statement inside a WHEN block if the ON...GOTO target is in the same protected region, an outer protected region, or in a non-protected region.
7. You cannot specify an ON...GOTO statement inside a WHEN block if the ON...GOTO target already resides in another protected region that does not contain the most current protected region.

---

### Example

```
ON INDEX% GOTO 700,800,900 OTHERWISE finish
.
.
.
finish:
    END PROGRAM
```

---

## OPEN

The OPEN statement opens a file for processing. It transfers user-specified file characteristics to VAX Record Management Services (RMS) and verifies the results.

---

### Format

```
OPEN file-spec1 [ FOR INPUT
                  FOR OUTPUT ] AS [ FILE ] [#] chnl-exp
                  [, open-clause ]...
```

*open-clause*:

```
[ ACCESS { APPEND
           READ
           WRITE
           MODIFY
           SCRATCH } ]
```

```
[ ALLOW { NONE
          READ
          WRITE
          MODIFY } ]
```

```
[ BUFFER int-exp4 ]
```

```
[ CONTIGUOUS ]
```

```
[ DEFAULTNAME file-spec2 ]
```

```
[ EXTENDSIZE int-exp5 ]
```

```
[ FILESIZE int-exp2 ]
```

# OPEN

[ **MAP** *map-name* ]

[ [ **ORGANIZATION** { **INDEXED**  
**RELATIVE**  
**SEQUENTIAL**  
**UNDEFINED**  
**VIRTUAL** } [ **STREAM**  
**VARIABLE**  
**FIXED** ] ] ]

[ **RECORDSIZE** *int-exp1* ]

[ **RECORDTYPE** { **LIST**  
**FORTRAN**  
**NONE**  
**ANY** } ]

[ **TEMPORARY** ]

[ **UNLOCK EXPLICIT** ]

[ **USEROPEN** *func-name* ]

[ **WINDOWSIZE** *int-exp3* ]

## Sequential Files Only

[ **BLOCKSIZE** *int-exp8* ]

[ **NOREWIND** ]

[ **NOSPAN** ]

[ **SPAN** ]

## Relative and Indexed Files Only

[ **BUCKETSIZE** *int-exp9* ]

## Indexed Files Only

```
[ ALTERNATE [ KEY ] key-clause [ DUPLICATES ] [ CHANGES ]
  [ ASCENDING
  DESCENDING ] ]
```

```
[ CONNECT chnl-exp2 ]
```

```
[ PRIMARY [ KEY ] key-clause [ DUPLICATES ] [ ASCENDING
  DESCENDING ] ]
```

*key-clause*: { *int-unsubs-var*  
*decimal-unsubs-var*  
*str-unsubs-var*  
(*str-unsubs-var1* ,... *str-unsubs-var8* )  
*quad-record-group* }

---

## Syntax Rules

1. *File-spec1* specifies the file to be opened and associated with *chnl-exp*. It can be any valid string expression and must be a valid VMS file specification. VAX BASIC passes these values to RMS without editing, alteration, or validity checks.

VAX BASIC does not supply any default file specifications, unless you include the **DEFAULTNAME clause** in the OPEN statement.

2. The **FOR clause** determines how VAX BASIC opens a file.
  - If you open a file with FOR INPUT, the file must exist or VAX BASIC signals an error.
  - If you open a file with FOR OUTPUT, VAX BASIC creates the file if it does not exist. If the file does exist, VAX BASIC creates a new version of the file.

# OPEN

- If you do not use FOR INPUT or FOR OUTPUT to open an indexed file, you must specify a primary key in the event the file does not exist.
  - If you do not specify either FOR INPUT or FOR OUTPUT, VAX BASIC tries to open an existing file. If there is no such file, VAX BASIC creates one.
3. *Chnl-exp* is a numeric expression that specifies a channel number to be associated with the file being opened. It can be preceded by an optional number sign (#) and must be in the range of 1 through 119. Note that channels 100 through 119 are usually reserved for allocation by the RTL routines, LIB\$GET\_LUN and LIB\$FREE\_LUN.
  4. A statement that accesses a file cannot execute until you open that file and associate it with a channel.

---

## Remarks

1. The OPEN statement does not retrieve records.
2. Channel #0, the terminal, is always open. If you try to open channel zero, VAX BASIC signals the error "Illegal I/O channel" (ERR=46).
3. If a program opens a file on a channel already associated with an open file, VAX BASIC closes the previously opened file and opens the new one.
4. The **ACCESS clause** determines how the program can use the file.
  - ACCESS READ allows only FIND, GET, or other input statements on the file. The OPEN statement cannot create a file if the ACCESS READ clause is specified.
  - ACCESS WRITE allows only PUT, UPDATE, or other output statements on the file.
  - ACCESS MODIFY allows any I/O statement except SCRATCH on the file. ACCESS MODIFY is the default.
  - ACCESS SCRATCH allows any I/O statement valid for a sequential or terminal-format file.



- **ACCESS APPEND** is the same as **ACCESS WRITE** for sequential files, except that VAX BASIC positions the file pointer after the last record when it opens the file. You cannot use **ACCESS APPEND** on relative or indexed files.
5. The **ALLOW clause** can be used in the **OPEN** statement to specify file sharing of relative, indexed, sequential, and virtual files.
    - **ALLOW NONE** lets no other users access the file. This is the default if any access other than **READ** is specified. Note that you must have write access to the file in order to specify **ALLOW NONE**.
    - **ALLOW READ** lets other users have read access to the file.
    - **ALLOW WRITE** lets other users have write access to the file.
    - **ALLOW MODIFY** lets other users have unlimited access to the file.
  6. The **BLOCKSIZE clause** specifies the physical block size of magnetic tape files. The **BLOCKSIZE** clause can be used for magnetic tape files only.
    - The value of *int-exp8* is the number of records in a block. Therefore, the block size in bytes is the product of the **RECORDSIZE** and the **BLOCKSIZE** value.
    - The default blocksize is one record.
  7. The **BUCKETSIZE clause** applies only to relative and indexed files. It specifies the size of an RMS bucket in terms of the number of records one bucket should hold.
    - The value of *int-exp9* is the number of records in a bucket.
    - The default is one record.
  8. The **BUFFER clause** can be used with all file organizations except **UNDEFINED**.
    - For **RELATIVE** and **INDEXED** files, *int-exp4* specifies the number of device or file buffers RMS uses for file processing.
    - For **SEQUENTIAL** files, *int-exp4* specifies the size of the buffer; for example, **BUFFER 8** for a **SEQUENTIAL** file sets the buffer size to eight 512-byte blocks.
    - **DIGITAL** recommends that you accept the system defaults or change the defaults with the **DCL SET RMS\_DEFAULT** command.

# OPEN

9. The **CONTIGUOUS clause** causes RMS to try to create the file as a contiguous-best-try sequence of disk blocks. The CONTIGUOUS clause does not affect existing files or nondisk files.

The CONTIGUOUS clause does not guarantee that the file will occupy contiguous disk space. If RMS can locate the file in a contiguous area, it will do so. However, if there is not enough free contiguous space for a file, RMS allocates the largest possible contiguous space and does not signal an error. See the *VMS Record Management Services Manual* for more information on contiguous disk allocation.

10. The **CONNECT clause** permits multiple record streams to be connected to the file.

- The CONNECT clause must specify an INDEXED file already opened on *chnl-exp2* with the primary OPEN statement.
- You cannot connect to a connected channel; you can connect only to the initially opened channel.
- You can connect more than one stream to an open channel.
- All clauses of the two files to be connected must be identical except MAP, CONNECT, and USEROPEN.
- Do not use the CONNECT clause when accessing files over DECnet or VAX BASIC will signal the error "Cannot open file" (ERR=162).

11. The **DEFAULTNAME clause** lets you supply a default file specification. If *file-spec1* is not a complete file specification, *file-spec2* in the DEFAULTNAME clause supplies the missing parts. For example:

```
10      INPUT 'FILE NAME';fnam$
20      OPEN fnam$ FOR INPUT AS FILE #1%,    &
          DEFAULTNAME "USER$$DISK:.DAT"
```

If you type "ABC" for the file name, VAX BASIC tries to open USER\$\$DISK:[ ]ABC.DAT.

12. The **EXTENDSIZE clause** lets you specify the increment by which RMS extends a file after its initial allocation is filled. The value of *int-exp5* is in 512-byte disk blocks. The EXTENDSIZE clause has no effect on an existing file.
13. The **FILESIZE clause** lets you pre-extend a new file to a specified size.
- The value of *int-exp2* is the initial allocation of disk blocks.
  - The FILESIZE clause has no effect on an existing file.

14. The **MAP clause** specifies that a previously declared map is associated with the file's record buffer. The MAP clause determines the record buffer's address and length unless overridden by the RECORDSIZE clause.
  - The size of the specified map must be as large or larger than the longest record length or maximum record size. For files with a fixed record size, the specified map must match exactly.
  - The size of the largest MAP with the same map name in the current program unit becomes the file's record size if the OPEN statement does not include a RECORDSIZE clause.
  - DIGITAL recommends that you do not use both the MAP and RECORDSIZE clauses in an OPEN statement. However, if you do use both the MAP and RECORDSIZE clauses in an OPEN statement, the following rules apply:
    - The RECORDSIZE clause overrides the record size set by the MAP clause.
    - The map must be as large or larger than the specified RECORDSIZE.
  - If there is no MAP clause, the record buffer space that VAX BASIC allocates is not directly accessible. Therefore, MOVE statements are needed to access data in the record buffer.
  - You must have a MAP clause when creating an indexed file; you cannot use KEY clauses without MAP statements because keys serve as offsets into the buffer.
  - The size of the specified map cannot exceed 32767 bytes.
15. The **NOREWIND clause** controls tape positioning on magnetic tape files. The NOREWIND clause can be used for magnetic tape files only.
  - If you specify NOREWIND, the OPEN statement does not position the tape at the beginning. Your program can search for records from the current position.
  - If you do not specify either ACCESS APPEND or NOREWIND, the OPEN statement positions the tape at its beginning and then searches for the file.

# OPEN

16. The **NOSPAN clause** specifies that sequential records cannot cross block boundaries.
  - **SPAN** specifies that records can cross block boundaries. **SPAN** is the default.
  - The **NOSPAN** clause does not affect nondisk files.
17. The **ORGANIZATION clause** specifies the file organization. When present, it must precede all other clauses. When you specify an **ORGANIZATION** clause, you must also specify one of the following organization options: **VIRTUAL**, **UNDEFINED**, **INDEXED**, **SEQUENTIAL** or **RELATIVE**. Specify **ORGANIZATION UNDEFINED** if you do not know the actual organization of the file. If you do not specify an **ORGANIZATION** clause **VAX BASIC** opens a terminal format file by default.
  - When you specify **ORGANIZATION VIRTUAL**, you create a sequentially fixed file with a record size of 512 (or a multiple of 512). You can then access the file with the **FIND**, **GET**, **PUT**, or **UPDATE** statements or through one or more virtual arrays. **VAX BASIC** allows you to overwrite existing records in a file not containing virtual arrays and opened as **ORGANIZATION VIRTUAL** by using the **PUT** statement with a **RECORD** clause. All other organizations require the **UPDATE** statement to change an existing record. **DIGITAL** recommends that you also use the
  - **UPDATE** statement to change existing records in **VIRTUAL** files that do not contain virtual arrays.
  - When you do not know the organization of a file, you can open a file for input and specify **ORGANIZATION UNDEFINED**. You can then use the **FSP\$** function or a **USEROPEN** routine to determine the attributes of the file. You will usually want to specify the **RECORDTYPE ANY** clause with the **ORGANIZATION UNDEFINED** clause. The combination of these two clauses should allow you to access any file sequentially.
  - When you specify **ORGANIZATION INDEXED**, you create an indexed file whose data records are sorted in ascending or descending order according to a *primary index key value*.
    - The index keys you specify determine the order in which records are stored.

- Index keys must be variables declared in a MAP statement associated with the OPEN statement for the file.
  - VAX BASIC allows you to specify an indexed file as either variable or fixed length.
  - When you specify ORGANIZATION SEQUENTIAL, you create a file that stores records in the order that they are written.
    - Sequential files can contain records of any valid VAX BASIC record format: fixed-length, variable-length, or stream.
    - If you open an existing file using stream as a record format option, the file must be one of the following stream record formats defined by RMS:
      - STREAM records can be delimited by any special character.
      - STREAM\_LF must be delimited by a line-feed character.
      - STREAM\_CR must be delimited by a carriage return.

If the file is not one of these stream formats, VAX BASIC signals the error “RECATNOT, record attributes not matched”.
  - When you specify ORGANIZATION RELATIVE, you create a file that contains a series of records that are numbered consecutively. VAX BASIC allows you to specify either fixed-length or variable-length records.
  - If you omit the ORGANIZATION clause entirely, a terminal-format file is opened.
    - Terminal-format files are implemented as RMS sequential variable files and store ASCII characters in variable-length records.
    - Carriage control is performed by the operating system; the record does not contain carriage returns or line feeds.
    - You use essentially the same syntax to access terminal-format files as when reading from or writing to the terminal (INPUT and PRINT).
18. The **PRIMARY KEY** clause lets you specify an indexed file’s key. You must specify a primary key when opening an indexed file. The

# OPEN

**ALTERNATE KEY** clause lets you specify up to 254 alternate keys. The **ALTERNATE KEY** clause is optional.

- **RMS** creates one index list for each primary and alternate key you specify. These indexes are part of the file and contain pointers to the records. Each key you specify corresponds to a sorted list of record pointers.
- You can specify each key as **ASCENDING** or **DESCENDING**; **ASCENDING** is the default. In an **ASCENDING** key, lower key values occur toward the beginning of the index. In a **DESCENDING** key, higher key values occur toward the beginning of the index.
- The keys you specify determine the order in which records in the file are stored. All keys must be variables declared in the file's corresponding **MAP** statement. The position of the key in the **MAP** statement determines its position in the record. The data type and size of the key are as declared in the **MAP** statement.
- A key can be an unsubscripted string, a **WORD**, **LONG**, or packed decimal variable, or a record or group which is exactly eight bytes long.
- You can also create a segmented index key for string keys by separating the string variable names with commas and enclosing them in parentheses. You can then reference a segment of the specified key by referencing one of the string variables instead of the entire key. A string key can have up to eight segments.
- The order of appearance of keys determines key numbers. The primary key, which must appear first, is key #0. The first alternate key is #1, and so on.
- **DUPLICATES** in the **PRIMARY** and **ALTERNATE** key clauses specifies that two or more records can have the same key value. If you do not specify **DUPLICATES**, the key value must be unique in all records.
- **CHANGES** in the **ALTERNATE KEY** clause specifies that you can change the value of an alternate key when updating records. If you do not specify **CHANGES** when creating the file, you cannot change the value of a key. You cannot specify **CHANGES** with the **PRIMARY KEY** clause.
- **KEY** clauses are optional for existing files. If you do specify a key, it must match a key in the file.

19. The **RECORDTYPE** clause specifies the file's record attributes.
- **LIST** specifies implied carriage control, <CR>. This is the default for all file organizations except **VIRTUAL**.
  - **FORTRAN** specifies a control character in the record's first byte.
  - **NONE** specifies no attributes. This is the default for **VIRTUAL** files.  
If you open a terminal-format file with **RECORDTYPE NONE**, you must explicitly insert carriage control characters into the records your program writes to the file.
  - **ANY** specifies a match with any file attributes when opening an existing file. If you create a new file, **ANY** is treated as **LIST** for all organizations except **VIRTUAL**. For **VIRTUAL**, it is treated as **NONE**.
20. The **RECORDSIZE** clause specifies the file's record size. Note that there are restrictions on the maximum record size allowed for various file and record formats. See the *VMS Record Management Services Manual* for more information.
- For fixed-length records, *int-exp1* specifies the size of all records.
  - For variable-length records, *int-exp1* specifies the size of the largest record.
  - **DIGITAL** recommends that you do not use both the **MAP** and **RECORDSIZE** clauses in an **OPEN** statement. However, if you do use both the **MAP** and **RECORDSIZE** clauses in an **OPEN** statement, the following rules apply:
    - The **RECORDSIZE** clause overrides the record size set by the **MAP** clause.
    - The map must be as large or larger than the specified **RECORDSIZE**.
  - If you specify a **MAP** clause but no **RECORDSIZE** clause, the record size is equal to the map size.
  - If there is no **MAP** clause, the **RECORDSIZE** clause determines the record size.
  - When creating a relative or indexed file, you must specify either a **MAP** or **RECORDSIZE** clause. Otherwise, **VAX BASIC** signals an error.
  - For fixed files, the record size must match exactly.

## OPEN

- If you do not specify a **RECORDSIZE** clause when opening an existing file, VAX BASIC retrieves the record size value from the file.
  - When you print to a terminal-format file, you must supply a record size if the margin is to exceed 72 characters. For example, if you want to print a 132-character line, specify **RECORDSIZE 132** or use the **MARGIN** and **NOMARGIN** statements.
  - When creating **SEQUENTIAL** files, VAX BASIC supplies a default record size of 132.
  - The record size is always 512 for **VIRTUAL** files, unless you specify a **RECORDSIZE**.
21. The **TEMPORARY** clause causes VAX BASIC to delete the output file as soon as the program closes it.
  22. The **UNLOCK EXPLICIT** clause allows you to retain locks on records until they are explicitly unlocked.
    - The type of lock you impose on a record with a **GET** or **FIND** statement remains in effect until you explicitly unlock the record or file with a **FREE** or **UNLOCK** statement or until you close the file.
    - If you specify **UNLOCK EXPLICIT**, and do not specify an **ALLOW** clause with a **GET** or **FIND** statement, VAX BASIC imposes the **ALLOW NONE** lock by default and the next **GET** or **FIND** operation does not unlock the previously locked record.
    - You must open a file with **UNLOCK EXPLICIT** before you can explicitly lock records with the **ALLOW** clause on **GET** and **FIND** statements. See the sections on **GET** and **FIND** in this manual and the *VAX BASIC User Manual* for more information on explicit record locking and unlocking.
  23. The **USEROPEN** clause lets you open a file with your own **FUNCTION** subprogram.
    - *Func-name* must be a separately compiled **FUNCTION** subprogram and must conform to **FUNCTION** statement rules for naming subprograms.
    - You do not need to declare the **useropen** routine as an external function.



# OPEN

- VAX BASIC calls the user program after it fills the FAB (File Access Block), the RAB (Record Access Block), and the XABs (Extended Attribute Blocks). The subprogram must issue the appropriate RMS calls, including \$OPEN and \$CONNECT, and return the RMS status as the value of the function. See the *VAX BASIC User Manual* for more information on the USEROPEN routine.

## NOTE

Future releases of the Run-Time Library may alter the use of some VAX RMS fields. Therefore, you may have to alter your USEROPEN procedures accordingly.

24. The **WINDOWSIZE clause** followed by *int-exp3* lets you specify the number of block retrieval pointers you want to maintain in memory for the file.

Retrieval pointers are associated with the file header and point to contiguous blocks on disk.

- By keeping retrieval pointers in memory you can reduce the I/O associated with locating a record, as the operating system does not have to access the file header for pointers as frequently.
- The number of retrieval pointers in memory at any one time is determined by the system default or by the WINDOWSIZE clause.
- The default number of retrieval pointers on VMS systems is 7.
- A value of zero specifies the default number of retrieval pointers. A value of -1 means to map the entire file, if possible. Values from -128 through -2 are reserved.

---

## Examples

### Example 1

```
OPEN "FILE.DAT" AS FILE #4
```

# OPEN

## Example 2

```
OPEN "INPUT.DAT" FOR INPUT AS FILE #4,           &
    ORGANIZATION SEQUENTIAL FIXED,             &
    RECORDSIZE 200,                             &
    MAP ABC,                                     &
    ALLOW MODIFY, ACCESS MODIFY

OPEN Newfile$ FOR OUTPUT AS FILE #3,           &
    INDEXED VARIABLE,                           &
    MAP Emp_name,                               &
    DEFAULTNAME "USER$$DISK:.DAT",             &
    PRIMARY KEY Last$ DUPLICATES,              &
    ALTERNATE KEY First$ CHANGES

MAP (SEGKEY) STRING last_name = 15,           &
    MI = 1, first_name = 15

OPEN "NAMES.IND" FOR OUTPUT AS FILE #1,        &
    ORGANIZATION INDEXED,                       &
    PRIMARY KEY (last_name, first_name, MI),    &
    MAP SEGKEY
```

## Example 3

```
MAP (OWNERKEYS) STRING owner_id = 6, dog_reg_no = 7, &
    last_name = 25, first_name = 20

OPEN "OWNERS.IND" FOR OUTPUT AS FILE #1,        &
    ORGANIZATION INDEXED,                       &
    PRIMARY KEY (owner_id),                     &
    ALTERNATE KEY (last_name) DUPLICATES CHANGES, &
    ALTERNATE (dog_reg_no) DESCENDING,         &
    MAP OWNERKEYS
```

The MAP statement describes the three string variables used as index keys in the file OWNERS.IND. The OPEN statement declares an indexed file with two alternate keys in addition to the primary key. The alternate key *dog\_reg\_no* is a DESCENDING key; the other keys are ASCENDING by default.

---

## OPTION

The **OPTION** statement allows you to set compilation qualifiers such as default data type, size, and scale factor. You can also set compilation conditions such as severity of run-time errors to handle, constant type checking, subscript checking, overflow checking, decimal rounding, and setup in a source program. The options you set affect only the program module in which the **OPTION** statement occurs.

---

### Format

**OPTION** *option-clause*,...

*option-clause*:

{	<b>ANGLE=</b> <i>angle-clause</i> <b>HANDLE=</b> <i>handle-clause</i> <b>CONSTANT TYPE =</b> <i>const-type-clause</i> <b>OLD VERSION =</b> CDD <b>TYPE=</b> <i>type-clause</i> <b>SIZE=</b> <i>size-clause</i> <b>SCALE=</b> <i>int-const</i>	}
	{ <b>ACTIVE</b> } = <i>active-clause</i> { <b>INACTIVE</b> }	

*angle-clause*:

{	<b>DEGREES</b> <b>RADIANS</b>	}
---	----------------------------------	---

*handle-clause*:

{	<b>BASIC</b> <b>SEVERE</b> <b>ERROR</b> <b>WARNING</b> <b>INFORMATIONAL</b>	}
---	---	---

## OPTION

*const-type-clause:* { **REAL**  
**INTEGER**  
**DECIMAL** }

*type-clause:* { **INTEGER**  
**REAL**  
**EXPLICIT**  
**DECIMAL** }

*size-clause:* { *size-item*  
(*size-item*,...) }

*size-item:* { **INTEGER** *int-clause*  
**REAL** *real-clause*  
**DECIMAL**(*d,s*) }

*int-clause:* { **BYTE**  
**WORD**  
**LONG** }

*real-clause:* { **SINGLE**  
**DOUBLE**  
**GFLOAT**  
**HFLOAT** }

*active-clause:* { *active-item*  
(*active-item*,...) }

*active-item:*

{  
 INTEGER OVERFLOW  
 DECIMAL OVERFLOW  
 SETUP  
 DECIMAL ROUNDING  
 SUBSCRIPT CHECKING  
 }

---

## Syntax Rules

None.

---

## Remarks

1. **Option-clause** specifies the compilation qualifiers to be in effect for the program module.
2. **Angle-clause** specifies whether angles are to be evaluated in radians or in degrees. If you do not specify an *angle-clause*, VAX BASIC uses radians as the default.
3. **Handle-clause** specifies the severity level of the errors which are to be handled by an error handler.
  - If you do not specify an OPTION HANDLE statement, VAX BASIC uses OPTION HANDLE = BASIC as the default. Only those errors that are trappable and that map onto a VAX BASIC ERR value will transfer control to the current error handler. See the *VAX BASIC User Manual* for a list of VAX BASIC run-time errors.
  - If you specify a severity level, all trappable and non-trappable errors of the specified severity or less transfer control to the current error handler. This includes non-BASIC errors. For example, OPTION HANDLE = ERROR implies ERROR, WARNING, and INFORMATIONAL errors but not SEVERE errors.

## OPTION

- If you specify `OPTION HANDLE = SEVERE`, you can handle fatal errors. However, in most cases, a fatal error indicates that the program environment is badly corrupted and you should not continue program execution.
- 4. **Const-type-clause** specifies the data type for all constants that do not end in a data type suffix or are not in explicit literal notation with a data type supplied.
- 5. **Type-clause** sets the default data type for variables that have not been explicitly declared and for constants if no constant type clause is specified. You can specify only one *type-clause* in a program module.
- 6. **Size-clause** sets the default data subtypes for floating-point, integer, and packed decimal data. **Size-item** specifies the data subtype you want to set. You can specify an `INTEGER`, `REAL` or `DECIMAL` *size-item*, or an combination. Multiple *size-items* in an `OPTION` statement must be enclosed in parentheses and separated by commas.
- 7. **SCALE** controls the scaling of double precision floating-point variables. *Int-const* specifies the power of 10 you want as the scaling factor. It must be an integer between 0 and 6 or `VAX BASIC` signals an error. See the description of the `SCALE` command in Chapter 2 of this manual for more information on scaling.
- 8. `OLD VERSION = CDD` is provided for compatibility with previous versions of `BASIC`. When bounds are specified in the `CDD` array, `VAX BASIC` changes the lower bound to zero and adjusts the upper bound of the array. By default, if you do not specify `OLD VERSION = CDD`, `VAX BASIC` compiles the program with the bounds specified in the `CDD` data definition.
- 9. **Active-clause** specifies the decimal rounding, integer and decimal overflow checking, setup, and subscript checking conditions you want in effect for the program module. *Active-item* specifies the conditions you want to set. Multiple *active-items* in an `OPTION` statement must be enclosed in parentheses and separated by commas.
- 10. You can have more than one option in an `OPTION` statement, or you can use multiple `OPTION` statements in a program module. However, each `OPTION` statement must lexically precede all other source code in the program module, with the exception of comment fields, `REM`, `PICTURE`, `PROGRAM`, `SUB`, `FUNCTION`, and `OPTION` statements.

11. `OPTION` statement specifications apply only to the program module in which the statement appears and affect all variables in the module, including `SUB` and `FUNCTION` parameters.
12. `VAX BASIC` signals an error in the case of conflicting options. For example, you cannot specify more than one *type-clause* or `SCALE` factor in the same program unit.
13. If you do not specify a *type-clause* or a *subtype-clause*, `VAX BASIC` uses the current environment default data types.
14. If you do not specify a scale factor, `VAX BASIC` uses the current environment default scale factor.
15. `ACTIVE` specifies the conditions that are to be in effect for a particular program module. `INACTIVE` specifies the conditions that are not to be in effect for a particular program module. If a condition does not appear in an *active-clause*, `VAX BASIC` uses the current environment default for the condition.

See the description of the `COMPILE` command in Chapter 2 of this manual and the *VAX BASIC User Manual* for more information on the `INTEGER_OVERFLOW`, `DECIMAL_OVERFLOW`, `SETUP`, `DECIMAL_ROUNDING`, and `SUBSCRIPT_CHECKING` compilation qualifiers. These qualifiers correspond to *active-clause* conditions (`INTEGER_OVERFLOW`, `DECIMAL_OVERFLOW`, `SETUP`, `DECIMAL_ROUNDING`, and `SUBSCRIPT_CHECKING`).

---

## Example

```

FUNCTION REAL DOUBLE monthly_payment,           &
      (DOUBLE interest_rate,                   &
       LONG no_of_payments,                   &
       DOUBLE principle)
OPTION TYPE = REAL,                             &
      SIZE = (REAL DOUBLE, INTEGER LONG),      &
      SCALE = 4

```

## PLACE\$

---

## PLACE\$

The PLACE\$ function explicitly changes the precision of a numeric string. PLACE\$ returns a numeric string, truncated or rounded, according to the value of an integer argument you supply.

---

### Format

*str-var* = PLACE\$ (*str-exp*, *int-exp*)

---

### Syntax Rules

1. *Str-exp* specifies the numeric string you want to process. It can contain an optional minus sign (-), ASCII digits, and an optional decimal point.
  2. *Int-exp* specifies the numeric precision of *str-exp*. Table 4-5 shows examples of rounding and truncation and the values of *int-exp* that produce them.
- 

### Remarks

1. The PLACE\$ function does not support E-format notation.
2. If *str-exp* has more than 60 characters, VAX BASIC signals the error "Illegal number" (ERR=52).
3. *Str-exp* is rounded or truncated, or both, according to the value of *int-exp*.
4. If *int-exp* is between -60 and 60, rounding and truncation occur as follows:
  - For positive integer expressions, rounding occurs to the right of the decimal place. For example, if *int-exp* is 1, rounding occurs one digit to the right of the decimal place (the number is rounded to the nearest tenth). If *int-exp* is 2, rounding occurs two digits to the



right of the decimal place (the number is rounded to the nearest hundredth), and so on.

- If *int-exp* is zero, VAX BASIC rounds to the nearest unit.
  - For negative integer expressions, rounding occurs to the left of the decimal point. If *int-exp* is  $-1$ , for example, VAX BASIC moves the decimal point one place to the left, then rounds to units. If *int-exp* is  $-2$ , rounding occurs two places to the left of the decimal point; VAX BASIC moves the decimal point two places to the left, then rounds to tens.
5. If *int-exp* is between 9940 and 10,060, truncation occurs as follows:
    - If *int-exp* is 10,000, VAX BASIC truncates the number at the decimal point.
    - If *int-exp* is greater than 10,000 (10,000 plus  $n$ ) VAX BASIC truncates the numeric string  $n$  places to the right of the decimal point. For example, if *int-exp* is 10,001 (10,000 plus 1), VAX BASIC truncates the number starting one place to the right of the decimal point. If *int-exp* is 10,002 (10,000 plus 2), VAX BASIC truncates the number starting two places to the right of the decimal point, and so on.
    - If *int-exp* is less than 10,000 (10,000 minus  $n$ ), VAX BASIC truncates the numeric string  $n$  places to the left of the decimal point. For example, if *int-exp* is 9999 (10,000 minus 1), VAX BASIC truncates the number starting one place to the left of the decimal point. If 9998 (10,000 minus 2), VAX BASIC truncates starting two places to the left of the decimal point, and so on.
  6. If *int-exp* is not between  $-60$  and  $60$  or 9940 and 10,060, VAX BASIC returns a value of zero.
  7. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to an integer of the default size.
  8. Table 4-5 shows examples of rounding and truncation and the values of *int-exp* that produce them. The number used is 123456.654321.

# PLACES

**Table 4–5: Rounding and Truncation of 123456.654321**

<b>Int-exp</b>	<b>Effect</b>	<b>Value Returned</b>
–5	Rounded to 100,000s and truncated	1
–4	Rounded to 10,000s and truncated	12
–3	Rounded to 1000s and truncated	123
–2	Rounded to 100s and truncated	1235
–1	Rounded to 10s and truncated	12346
0	Rounded to units and truncated	123457
1	Rounded to tenths and truncated	123456.7
2	Rounded to hundredths and truncated	123456.65
3	Rounded to thousandths and truncated	123456.654
4	Rounded to ten-thousandths and truncated	123456.6543
5	Rounded to hundred-thousandths and truncated	123456.65432
9,995	Truncated to 100,000s	1
9,996	Truncated to 10,000s	12
9,997	Truncated to 1000s	123
9,998	Truncated to 100s	1234
9,999	Truncated to 10s	12345
10,000	Truncated to units	123456
10,001	Truncated to tenths	12345.6
10,002	Truncated to hundredths	123456.65
10,003	Truncated to thousandths	123456.654
10,004	Truncated to ten-thousandths	123456.6543
10,005	Truncated to hundred-thousandths	123456.65432

---

## Example

```
DECLARE STRING str_exp, str_var
str_exp = "9999.9999"
str_var = PLACE$(str_exp,3)
PRINT str_var
```

### Output

```
10000
```

# POS

---

## POS

The POS function searches for a substring within a string and returns the substring's starting character position.

---

### Format

*int-var* = **POS** (*str-exp1*, *str-exp2*, *int-exp*)

---

### Syntax Rules

1. *Str-exp1* specifies the main string.
2. *Str-exp2* specifies the substring.
3. *Int-exp* specifies the character position in the main string at which VAX BASIC starts the search.

---

### Remarks

1. The POS function searches *str-exp1*, the main string, for the first occurrence of *str-exp2*, the substring, and returns the position of the substring's first character.
2. If *int-exp* is greater than the length of the main string, POS returns a value of zero.
3. POS always returns the character position in the main string at which VAX BASIC finds the substring, with the following exceptions:
  - If only the substring is null, and if *int-exp* is less than or equal to zero, POS returns a value of 1.
  - If only the substring is null, and if *int-exp* is equal to or greater than 1 and less than or equal to the length of the main string, POS returns the value of *int-exp*.

- If only the substring is null and if *int-exp* is greater than the length of the main string, POS returns the main string's length plus 1.
  - If only the main string is null, POS returns a value of zero.
  - If both the main string and the substring are null, POS returns 1.
4. If VAX BASIC cannot find the substring, POS returns a value of zero.
  5. If *int-exp* is less than 1, VAX BASIC assumes a starting position of 1.
  6. If *int-exp* does not equal 1, VAX BASIC still counts from the string's beginning to calculate the starting position of the substring. That is, VAX BASIC counts character positions starting at position 1, regardless of where you specify the start of the search. For example, if you specify 10 as the start of the search and VAX BASIC finds the substring at position 15, POS returns the value 15.
  7. If you know that the substring is not near the beginning of the string, specifying a starting position greater than 1 speeds program execution by reducing the number of characters VAX BASIC must search.
  8. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to an integer of the default size.

---

## Example

```
DECLARE STRING main_str, &  
             sub_str  
DECLARE INTEGER first_char  
main_str = "ABCDEFGH"  
sub_str = "DEFG"  
first_char = POS(main_str, sub_str, 1)  
PRINT first_char
```

### Output

4

# PRINT

---

## PRINT

The PRINT statement transfers program data to a terminal or a terminal-format file.

---

### Format

**PRINT** [*#chnl-exp*, ] [*output-list*]

*output-list*: [*exp*][ { ; } *exp* ]... [ ; ]

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#). If you do not specify a channel, VAX BASIC prints to the controlling terminal.
2. *Output-list* specifies the expressions to be printed and the print format to be used.
3. *Exp* can be any valid expression.
4. A separator character (comma or semicolon) must separate each *exp*. Separator characters control the print format as follows:
  - A comma (,) causes VAX BASIC to skip to the next print zone before printing the expression.
  - A semicolon (;) causes VAX BASIC to print the expression immediately after the previous expression.

---

**Remarks**

1. A terminal or terminal-format file must be open on the specified channel. (Your current terminal is always open on channel #0.)
2. A PRINT line has an integral number of print zones. Note, however, that the number of print zones in a line differs from terminal to terminal.
3. The right margin setting, if set by the MARGIN statement, controls the width of the PRINT line. The default right margin is 72.
4. The PRINT statement prints string constants and variables exactly as they appear, with no leading or trailing spaces.
5. VAX BASIC prints quoted string literals exactly as they appear. Therefore, you can print quotation marks, commas, and other characters by enclosing them in quotation marks.
6. A PRINT statement with no *output-list* prints a blank line.
7. An expression in the *output-list* can be followed by more than one separator character. That is, you can omit an expression and specify where the next expression is to be printed by the use of multiple separator characters. For example:

```
PRINT "Name",,"Address and ";"City"
```

**Output**

```
Name                Address and City
```

In this example, the double commas after “Name” cause VAX BASIC to skip two print zones before printing “Address and ”. The semicolon causes the next expression, “City”, to be printed immediately after the preceding expression. Multiple semicolons have the same effect as a single semicolon.

8. When printing numeric fields, VAX BASIC precedes each number with a space or minus sign (–) and follows it with a space.
9. VAX BASIC does not print trailing zeros to the right of the decimal point. If all digits to the right of the decimal point are zeros, VAX BASIC omits the decimal point as well.

# PRINT

10. For REAL numbers (SINGLE, DOUBLE, GFLOAT, and HFLOAT), VAX BASIC does not print more than six digits in explicit notation. If a number requires more than six digits, VAX BASIC uses E format and precedes positive exponents with a plus sign (+). VAX BASIC rounds a floating-point number with a magnitude between 0.1 and 1.0 to six digits. For magnitudes smaller than 0.1, VAX BASIC rounds the number to six digits and prints it in E format.
11. The PRINT statement can print up to
  - 3 digits of precision for BYTE integers
  - 5 digits of precision for WORD integers
  - 10 digits of precision for LONG integers
  - 31 digits of precision for DECIMAL numbers
  - The string length for STRING values

VAX BASIC prints both INTEGER and DECIMAL values according to the previous rules. However, for REAL values, VAX BASIC displays a maximum of 6 digits.
12. If there is a comma or semicolon following the last item in *output-list*, VAX BASIC does the following:
  - When printing to a terminal, VAX BASIC does not generate a line terminator after printing the last item. The next item printed with a PRINT statement is printed at the position specified by the separator character following the last item in the first PRINT statement.
  - When printing to a terminal-format file, VAX BASIC does not write out the record until a PRINT statement without trailing punctuation executes.
13. If no punctuation follows the last item in the *output-list* VAX BASIC does the following:
  - When printing to a terminal, VAX BASIC generates a line terminator after printing the last item.
  - When printing to a terminal-format file, VAX BASIC writes out the record after printing the last item.



## PRINT

14. If a string field does not fit on the current line VAX BASIC does the following:
  - When printing string elements to a terminal, VAX BASIC prints as much as will fit on the current line and prints the remainder on the next line.
  - When printing string elements to a terminal-format file, VAX BASIC prints the entire element on the next line.
15. If a numeric field is the first field in a line, and the numeric field spans more than one line, VAX BASIC prints part of the number on one line and the remainder on the next. Otherwise, numeric fields are never split across lines. If the entire field cannot be printed at the end of one line, the number is printed on the next line.
16. When a number's trailing space does not fit in the last print zone, the number is printed without the trailing space.

---

### Example

```
PRINT "name "; "age", "height "; "weight"
```

#### Output

```
name age      height weight
```

## PRINT USING

---

# PRINT USING

The PRINT USING statement generates output formatted according to a format string (either numeric or string) to a terminal or a terminal-format file.

---

### Format

**PRINT** [ *#chnl-exp* ] **USING** *str-exp* { ' ; ' } *output-list*

*output-list*: [ *exp* ] [ { ' ; ' } *exp* ]... [ ' ; ' ]

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#). If you do not specify a channel, VAX BASIC prints to the controlling terminal.
2. *Str-exp* is the format string. It must contain at least one valid format field and must be followed by a separator (comma or semicolon) and at least one expression.

#### NOTE

DIGITAL recommends that you use compile-time constant expressions for *str-exp* whenever possible. When you do this, the VAX BASIC compiler compiles the string at compilation time rather than at run time, thus improving the performance of your program.

3. *Output-list* specifies the expressions to be printed.
  - *Exp* can be any valid expression.
  - A comma or semicolon must separate each expression.
  - A comma or semicolon is optional after the last expression in the list.

---

## Remarks

1. The PRINT USING statement can print up to
  - Three digits of precision for BYTE integers
  - Five digits of precision for WORD integers
  - Six digits of precision for SINGLE floating-point numbers
  - Ten digits of precision for LONG integers
  - Sixteen digits of precision for DOUBLE floating-point numbers
  - Fifteen digits of precision for GFLOAT floating-point numbers
  - Thirty-three digits of precision for HFLOAT floating-point numbers
  - Thirty-one digits of precision for DECIMAL numbers
  - The string length for STRING values
2. A terminal or terminal-format file must be open on the specified channel or VAX BASIC signals an error.
3. The separator characters (comma or semicolon) in the PRINT USING statement do not control the print format as in the PRINT statement. The print format is controlled by the format string. Therefore, it does not matter whether you use a comma or semicolon.
4. **Formatting Numeric Output**
  - The number sign (#) reserves space for one sign or digit.
  - The comma (,) causes VAX BASIC to insert commas before every third significant digit to the left of the decimal point. In the format field, the comma must be to the left of the decimal point, and to the right of the rightmost dollar sign, asterisk, or number sign. A comma reserves space for a comma or digit.

## PRINT USING

- The period (.) inserts a decimal point. The number of reserved places on either side of the period determines where the decimal point appears in the output.
- The hyphen (–) reserves space for a sign and specifies trailing minus sign format. If present, it must be the last character in the format field. It causes VAX BASIC to print negative numbers with a minus sign after the last digit, and positive numbers with a trailing space. The hyphen (–) can be used as part of a dollar sign (\$\$) format field.
- The letters CD (Credit/Debit) enclosed in angle brackets (<CD>) print CR (Credit Record) after negative numbers or zero and DR (Debit Record) after positive numbers. If present, they must be the last characters in the format field. The Credit/Debit format can be used as part of a dollar sign (\$\$) format field.
- Four carets(^^^^) specify E-format notation for floating-point and DECIMAL numbers. They reserve four places for SINGLE, DOUBLE, GFLOAT, and DECIMAL values and five places for HFLOAT values. If present, they must be the last characters in the format field.
- Two dollar signs (\$\$) reserve space for a dollar sign and a digit and cause VAX BASIC to print a dollar sign immediately to the left of the most significant digit.
- Two asterisks (\*\*) reserve space for two digits and cause VAX BASIC to fill the left side of the numeric field with leading asterisks.
- A zero enclosed in angle brackets (<0>) prints leading zeros instead of leading spaces.
- A percent sign enclosed in angle brackets (<%>) prints all spaces in the field if the value of the print item is zero.

### NOTE

You cannot specify the dollar sign (\$\$), asterisk-fill (\*\*), and zero-fill (<0>) formats within the same print field. Similarly, VAX BASIC does not allow you to specify the zero-fill (<0>) and the blank-if-zero (<%>) formats within the same print field.

## PRINT USING

- An underscore ( `_` ) forces the next formatting character in the format string to be interpreted as a literal. It affects only the next character. If the next character is not a valid formatting character, the underscore has no effect and will itself be printed as a literal.
5. VAX BASIC interprets any other characters in a numeric format string as string literals.
  6. Depending on usage, the same format string characters can be combined to form one or more print fields within a format string. For example:

- When a dollar sign ( `$$` ) or asterisk-fill ( `**` ) format precedes a number sign ( `#` ), it modifies the number sign format. The dollar sign or asterisk-fill format reserves two places, and with the number signs forms one print field. For example:

`$$###`                Forms one field and reserves five spaces

`**###`                Forms one field and reserves four spaces

When these formats are not followed by a number sign or a blank-if-zero ( `<%>` ) format, they reserve two places and form a separate print field.

- When a zero-fill ( `<0>` ) or blank-if-zero format precedes a number sign, it modifies the number sign format. The `<0>` or `<%>` reserves one place, and with the number signs forms one print field. For example:

`<0>####`            Forms one field and reserves five spaces

`<%>###`             Forms one field and reserves four spaces

When these formats are not followed by a number sign, they reserve one space and form a separate print field.

- When a blank-if-zero ( `<%>` ) format follows a dollar sign or asterisk-fill format ( `**` ), it modifies the dollar sign ( `$$` ) or asterisk fill ( `**` ) format string. The blank-if-zero reserves one space, and with the dollar signs or asterisks forms one print field. For example:

`$$<%>###`           Forms one field and reserves six spaces

`**<%>###`           Forms one field and reserves five spaces

When the blank-if-zero precedes the dollar signs or asterisks, it reserves one space and forms a separate print field.

## PRINT USING

7. The comma (digit separator), dollar sign (currency symbol), and decimal point (radix point) are the defaults for U.S. currency. On VMS systems, you can change the digit separator, currency symbol and radix point by assigning the logical names `SYSDIGIT_SEP`, `SYS$CURRENCY` and `SYS$RADIX_POINT`. Once you make each assignment, the `PRINT USING` statement accesses these logical names for these symbols.
8. For E-format notation, `PRINT USING` left-justifies the number in the format field and adjusts the exponent to compensate, except when printing zero. When printing zero in E-format notation, VAX BASIC prints leading spaces, leading zeros, a decimal point, and zeros in the fractional portion if the `PRINT USING` string contains these formatting characters, and then the string "E+00".
9. Zero cannot be negative. If a small negative number rounds to zero, it is represented as a positive zero.
10. If there are reserved positions to the left of the decimal point, and the printed number is less than 1, VAX BASIC prints one zero to the left of the decimal point and pads with spaces to the left of the zero.
11. If there are more reserved positions to the right of the decimal point than fractional digits, VAX BASIC prints trailing zeros in those positions.
12. If there are fewer reserved positions to the right of the decimal point than fractional digits, VAX BASIC rounds the number to fit the reserved positions.
13. If a number does not fit in the specified format field, VAX BASIC prints a percent sign warning symbol (%), followed by the number in `PRINT` format.
14. **Formatting String Output**
  - Format string characters control string output and can be entered as either uppercase or lowercase characters. All format characters except the backslash and exclamation point must start with a single quotation mark ('). A single quote by itself reserves one character position. A single quote followed by any format characters marks the beginning of a character format field and reserves one character position.

## PRINT USING

- **L** reserves one character position. The number of **L**s plus the leading single quote determines the field's size. VAX BASIC left-justifies the print expression and pads with spaces if the print expression is less than or equal to the field's width. If the print expression is larger than the field, VAX BASIC left-justifies the expression and truncates its right side to fit the field.
  - **R** reserves one character position. The number of **R**s plus the leading single quote determines the field's size. VAX BASIC right-justifies the print expression and pads with spaces if the print expression is less than or equal to the field's width. If the print expression is larger than the field, VAX BASIC truncates the right side to fit the field.
  - **C** reserves one character position. The number of **C**s plus the leading single quote determines the field's size. If the string does not fit in the field, VAX BASIC truncates its right side. Otherwise, VAX BASIC centers the print expression in this field. If the string cannot be centered exactly, it is offset one character to the left.
  - **E** reserves one character position. The number of **E**s plus the leading single quote determines the field's size. VAX BASIC left-justifies the print expression if it is less than or equal to the field's width and pads with spaces. Otherwise, VAX BASIC expands the field to hold the entire print expression.
  - Two backslashes (\ \) when separated by  $n$  spaces reserve  $n+2$  character positions. PRINT USING left-justifies the string in this field. VAX BASIC does not allow a leading quotation mark with this format.
  - An exclamation point (!) creates a 1-character field. The exclamation point both starts and ends the field. VAX BASIC does not allow a leading quotation mark with this format.
15. VAX BASIC interprets any other characters in the format string as string literals and prints them exactly as they appear.
16. If a COMMA or semicolon follows the last item in *output-list*:
- When printing to a terminal, VAX BASIC does not generate a line terminator after printing the last item. The next item printed with a PRINT statement is printed at the position specified by the separator character following the last item in the first PRINT statement.

# PRINT USING

- When printing to a terminal-format file, VAX BASIC does not write out the record until a PRINT statement without trailing punctuation executes.
17. If no punctuation follows the last item in *output-list*:
- When printing to a terminal, VAX BASIC generates a line terminator after printing the last item.
  - When printing to a terminal-format file, VAX BASIC writes out the record after printing the last item.

---

## Examples

### Example 1

```
PRINT USING "###.###",-12.345
PRINT USING "##.###",12.345
```

### Output

```
-12.345
12.345
```

### Example 2

```
INPUT "Your Name";Winner$
    Jackpot = 10000.0
PRINT USING "CONGRATULATIONS, 'EEEEEEEEEE, YOU WON $$#####.##", Winner$, Jack
END
```

### Output

```
Your Name? Hortense Corabelle
CONGRATULATIONS, Hortense Corabelle, YOU WON $10000.00
```



---

## PROD\$

The PROD\$ function returns a numeric string that is the product of two numeric strings. The precision of the returned numeric string depends on the value of an integer argument.

---

### Format

*str-var* = PROD\$ (*str-exp1*, *str-exp2*, *int-exp*)

---

### Syntax Rules

1. *Str-exp1* and *str-exp2* specify the numeric strings you want to multiply. A numeric string can contain an optional minus sign (–), ASCII digits, and an optional decimal point (.)
2. If *str-exp* consists of more than 60 characters, VAX BASIC signals the error “Illegal number” (ERR=52).
3. *Int-exp* specifies the numeric precision of *str-exp*. Table 4–5 shows examples of rounding and truncation and the values of *int-exp* that produce them.

---

### Remarks

1. The PROD\$ function does not support E-format notation.
2. *Str-exp* is rounded or truncated, or both, according to the value of *int-exp*.
3. If *int-exp* is between –60 and 60, rounding and truncation occur as follows:
  - For positive integer expressions, rounding occurs to the right of the decimal place. For example, if *int-exp* is 1, rounding occurs one digit to the right of the decimal place (the number is rounded to

## PRODS

the nearest tenth). If *int-exp* is 2, rounding occurs two digits to the right of the decimal place (the number is rounded to the nearest hundredth), and so on.

- If *int-exp* is zero, VAX BASIC rounds to the nearest unit.
  - For negative integer expressions, rounding occurs to the left of the decimal point. If *int-exp* is  $-1$ , for example, VAX BASIC moves the decimal point one place to the left, then rounds to units. If *int-exp* is  $-2$ , rounding occurs two places to the left of the decimal point; VAX BASIC moves the decimal point two places to the left, then rounds to tens.
4. If *int-exp* is between 9940 and 10,060, truncation occurs as follows:
    - If *int-exp* is 10,000, VAX BASIC truncates the number at the decimal point.
    - If *int-exp* is greater than 10,000 (10,000 plus  $n$ ) VAX BASIC truncates the numeric string  $n$  places to the right of the decimal point. For example, if *int-exp* is 10,001 (10,000 plus 1), VAX BASIC truncates the number starting one place to the right of the decimal point. If *int-exp* is 10,002 (10,000 plus 2), VAX BASIC truncates the number starting two places to the right of the decimal point, and so on.
    - If *int-exp* is less than 10,000 (10,000 minus  $n$ ), VAX BASIC truncates the numeric string  $n$  places to the left of the decimal point. For example, if *int-exp* is 9999 (10,000 minus 1), VAX BASIC truncates the number starting one place to the left of the decimal point. If 9998 (10,000 minus 2), VAX BASIC truncates starting two places to the left of the decimal point, and so on.
  5. If *int-exp* is not between  $-60$  and  $60$  or 9940 and 10,060, VAX BASIC returns a value of zero.
  6. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to an integer of the default size.

---

**Example**

```
DECLARE STRING num_exp1, &  
             num_exp2, &  
             product  
num_exp1 = "34.555"  
num_exp2 = "297.676"  
product = PROD$(num_exp1, num_exp2, 1)  
PRINT product
```

**Output**

10286.2

# PROGRAM

---

## PROGRAM

The PROGRAM statement allows you to identify a main program with a name other than the file name.

---

### Format

**PROGRAM** *prog-name*

---

### Syntax Rules

1. *Prog-name* specifies the module name of the compiled source and cannot be the same as any SUB, FUNCTION or PICTURE name.
2. *Prog-name* also defines the global entry point name for the main program.
3. The first character of a *prog-name* must be an alphabetic character (A through Z). The remaining characters, if any, can be any combination of alphabetic characters, digits (0 through 9), dollar signs (\$), periods (.), and underscores (\_).
4. *Prog-name* cannot be a quoted name.

---

### Remarks

1. The PROGRAM statement must be the first statement in a main program and can be preceded only by comment fields and lexical directives.
2. If you insert the program into a text or object library or examine it using the VMS Debugger, the program name you specify will be the module name used.

# PROGRAM

3. The **PROGRAM** statement is optional; VAX BASIC allows you to specify an **END PROGRAM** statement and an **EXIT PROGRAM** statement without a matching **PROGRAM** statement.

---

## Example

```
PROGRAM first_test
.
.
.
END PROGRAM
```

# PUT

---

## PUT

The PUT statement transfers data from the record buffer to a file. PUT statements are valid on RMS sequential, relative, and indexed files. You cannot use PUT statements on terminal-format files or virtual array files.

---

### Format

**PUT** *#chnl-exp* [ , **RECORD** *num-exp* [ , **COUNT** *int-exp* ] ]

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. The **RECORD** clause allows you to randomly write records to a relative or sequential fixed file by specifying the record number. *Num-exp* must be between 1 and the maximum record number allowed for the file. VAX BASIC does not allow you to use the **RECORD** clause on sequential variable, sequential stream, or indexed files.
3. *Int-exp* in the **COUNT** clause specifies the record's size. If there is no **COUNT** clause, the record's size is that defined by the **MAP** or **RECORDSIZE** clause in the **OPEN** statement. The **RECORDSIZE** clause overrides the **MAP** clause.
  - If you write a record to a file with variable-length records, *int-exp* must be between zero and the maximum record size specified in the **OPEN** statement.
  - If you write a record to a file with fixed-length records, the **COUNT** clause serves no purpose. If used, *int-exp* must equal the record size specified in the **OPEN** statement.

---

## Remarks

1. For sequential access, the file associated with *chnl-exp* must be open with ACCESS WRITE, MODIFY, SCRATCH, or APPEND.
2. To add records to an existing sequential file, open it with ACCESS APPEND. If you are not at the end of the file when attempting a PUT to a sequential file, VAX BASIC signals “Not at end of file” (ERR=149).
3. After a PUT statement executes, there is no current record pointer. The next record pointer is set as follows:
  - For sequential files, variable and stream PUT operations set the next record pointer to the end of the file.
  - For relative files, a sequential PUT operation sets the next record pointer to the next record plus 1.
  - For relative and sequential fixed files, a random PUT operation leaves the next record pointer unchanged.
  - For indexed files, a PUT operation leaves the next record pointer unchanged.
4. When you specify a RECORD clause, VAX BASIC evaluates *num-exp* and uses this value as the relative record number of the target cell.
  - If the target cell is empty or occupied by a deleted record, VAX BASIC places the record in that cell.
  - If there is a record in the target cell and the file has not been opened as a VIRTUAL file, the PUT statement fails, and VAX BASIC signals the error “Record already exists” (ERR=153).
5. A PUT statement with no RECORD clause writes records to the file as follows:
  - For sequential variable and stream files, a PUT operation adds a record at the end of the file.
  - For relative and sequential fixed files, a PUT operation places the record in the empty cell pointed to by the next record pointer. If the file is empty, the first PUT operation places a record in cell number 1, the second in cell number 2, and so on.
  - For indexed files, RMS stores records in order of ascending primary key value and updates all indexes so that they point to the record.

# PUT

6. When you open a file as **ORGANIZATION VIRTUAL**, the file you open is a sequential fixed file with a record size that is a multiple of 512 bytes. You can then access the file with the **FIND**, **GET**, **PUT**, or **UPDATE** statements or through one or more virtual arrays. **VAX BASIC** allows you to overwrite existing records in a file not containing virtual arrays and opened as **ORGANIZATION VIRTUAL** by using the **PUT** statement with a **RECORD** clause. All other organizations require the **UPDATE** statement to change an existing record. **DIGITAL** recommends that you also use the **UPDATE** statement to change existing records in **VIRTUAL** files that do not contain virtual arrays.
7. If an existing record in an indexed file has a record with the same key value as the one you want to put in the file, **VAX BASIC** signals the error "Duplicate key detected" (**ERR=134**) if you did not specify **DUPLICATES** for the key in the **OPEN** statement. If you specified **DUPLICATES**, **RMS** stores the duplicate records in a first-in/first-out sequence.
8. The number specified in the **COUNT** clause determines how many bytes are transferred from the buffer to a file:
  - If you have not completely filled the record buffer before executing a **PUT** statement, **VAX BASIC** pads the record with nulls to equal the specified value.
  - If the specified **COUNT** value is less than the buffer size, the record is truncated to equal the specified value.
  - The number in the **COUNT** clause must not exceed the size specified in the **MAP** or **RECORDSIZE** clause in the **OPEN** statement or **VAX BASIC** signals "Size of record invalid" (**ERR=156**).
  - For files with fixed length records, the number in the **COUNT** clause must match the record size.



---

**Examples****Example 1**

```
!Sequential, Relative, Indexed, and Virtual Files  
PUT #3, COUNT 55%
```

**Example 2**

```
!Relative and Virtual Files Only  
PUT #5, RECORD 133, COUNT 16%
```

## QUO\$

---

## QUO\$

The QUO\$ function returns a numeric string that is the quotient of two numeric strings. The precision of the returned numeric string depends on the value of an integer argument.

---

### Format

*str-var* = QUO\$ (*str-exp1*, *str-exp2*, *int-exp*)

---

### Syntax Rules

1. *Str-exp1* and *str-exp2* specify the numeric strings you want to divide. A numeric string can contain an optional minus sign (-), ASCII digits, and an optional decimal point (.)
2. *Int-exp* specifies the numeric precision of *str-exp*. Table 4-5 shows examples of rounding and truncation and the values of *int-exp* that produce them.

---

### Remarks

1. The QUO\$ function does not support E-format notation.
2. If *str-exp* consists of more than 60 characters, VAX BASIC signals the error "Illegal number" (ERR=52).
3. *Str-exp* is rounded or truncated, or both, according to the value of *int-exp*.
4. If *int-exp* is between -60 and 60, rounding and truncation occur as follows:
  - For positive integer expressions, rounding occurs to the right of the decimal place. For example, if *int-exp* is 1, rounding occurs one digit to the right of the decimal place (the number is rounded to

the nearest tenth). If *int-exp* is 2, rounding occurs two digits to the right of the decimal place (the number is rounded to the nearest hundredth), and so on.

- If *int-exp* is zero, VAX BASIC rounds to the nearest unit.
  - For negative integer expressions, rounding occurs to the left of the decimal point. If *int-exp* is  $-1$ , for example, VAX BASIC moves the decimal point one place to the left, then rounds to units. If *int-exp* is  $-2$ , rounding occurs two places to the left of the decimal point; VAX BASIC moves the decimal point two places to the left, then rounds to tens.
5. If *int-exp* is between 9940 and 10,060, truncation occurs as follows:
    - If *int-exp* is 10,000, VAX BASIC truncates the number at the decimal point.
    - If *int-exp* is greater than 10,000 (10,000 plus  $n$ ) VAX BASIC truncates the numeric string  $n$  places to the right of the decimal point. For example, if *int-exp* is 10,001 (10,000 plus 1), VAX BASIC truncates the number starting one place to the right of the decimal point. If *int-exp* is 10,002 (10,000 plus 2), VAX BASIC truncates the number starting two places to the right of the decimal point, and so on.
    - If *int-exp* is less than 10,000 (10,000 minus  $n$ ), VAX BASIC truncates the numeric string  $n$  places to the left of the decimal point. For example, if *int-exp* is 9999 (10,000 minus 1), VAX BASIC truncates the number starting one place to the left of the decimal point. If 9998 (10,000 minus 2), VAX BASIC truncates starting two places to the left of the decimal point, and so on.
  6. If *int-exp* is not between  $-60$  and  $60$  or 9940 and 10,060, VAX BASIC returns a value of zero.
  7. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to an integer of the default size.

# QUO\$

---

## Example

```
DECLARE STRING num_str1, &  
             num_str2, &  
             quotient  
num_str1 = "458996.43"  
num_str2 = "123222.444"  
quotient = QUO$(num_str1, num_str2, 2)  
PRINT quotient
```

### Output

3.72

---

## RAD\$

The RAD\$ function converts a specified integer in Radix-50 format to a 3-character string.

### NOTE

The RAD\$ function is supported only for compatibility with BASIC-PLUS-2. DIGITAL recommends that you do not use the RAD\$ function for new program development.

---

### Format

*str-var* = RAD\$ (*int-var*)

---

### Syntax Rules

None.

---

### Remarks

1. The RAD\$ function does not support E-format notation.
2. The RAD\$ function converts *int-var* to a 3-character string in Radix-50 format and stores it in *str-var*. Radix-50 format allows you to store three characters of data as a 2-byte integer.
3. VAX BASIC supports the RAD\$ function, but not its complement, the FSS\$ function.
4. If you specify a floating-point variable for *int-var*, VAX BASIC truncates it to an integer of the default size.

# RAD\$

---

## Example

```
DECLARE STRING radix  
radix = RAD$(999)
```

---

## RANDOMIZE

The **RANDOMIZE** statement gives the random number function, **RND**, a new starting value.

---

### Format

```
{ RANDOMIZE }  
{ RANDOM   }
```

---

### Syntax Rules

None.

---

### Remarks

1. Without the **RANDOMIZE** statement, successive runs of the same program generate the same random number sequence.
2. If you use the **RANDOMIZE** statement before invoking the **RND** function, the starting point changes for each run. Therefore, a different random number sequence appears each time.

# RANDOMIZE

---

## Example

```
DECLARE REAL random_num
RANDOMIZE
  FOR I = 1 TO 2
    random_num = RND
    PRINT random_num
  NEXT I
```

### Output

```
.379784
.311572
```



---

# RCTRLC

The RCTRLC function disables CTRL/C trapping.

---

## Format

*int-var* = **RCTRLC**

---

## Syntax Rules

None.

---

## Remarks

1. After VAX BASIC executes the RCTRLC function, a CTRL/C typed at the terminal returns you to DCL command level or to the BASIC environment.
2. RCTRLC always returns a value of zero.

---

## Example

Y = RCTRLC

## RCTRLLO

---

## RCTRLLO

The RCTRLLO function cancels the effect of a CTRL/O typed on a specified channel.

---

### Format

*int-var* = RCTRLLO (*chnl-exp*)

---

### Syntax Rules

*Chnl-exp* must refer to a terminal.

---

### Remarks

1. If you type a CTRL/O to cancel terminal output, nothing is printed on the specified terminal until your program executes the RCTRLLO or until you type another CTRL/O, at which time normal terminal output resumes.
2. The RCTRLLO function always returns a value of zero.
3. RCTRLLO has no effect if the specified channel is open to a device that does not use the CTRL/O convention.

---

## Example

```
PRINT "A" FOR I% = 1% TO 10%  
Y% = RCTRL0(0%)  
PRINT "Normal output is resumed"
```

### Output

```
A  
A  
A  
A  
CTRL/O  
Output off  
  
Normal output is resumed
```

# READ

---

## READ

The READ statement assigns values from a DATA statement to variables.

---

### Format

**READ** *var*,...

---

### Syntax Rules

*Var* cannot be a DEF function name, unless the READ statement is inside the multi-line DEF body.

---

### Remarks

1. If your program has a READ statement without DATA statements, VAX BASIC signals a compile-time error.
2. When VAX BASIC initializes a program unit, it forms a data sequence of all values in all DATA statements. An internal pointer points to the first value in the sequence.
3. When VAX BASIC executes a READ statement, it sequentially assigns values from the data sequence to variables in the READ statement variable list. As VAX BASIC assigns each value, it advances the internal pointer to the next value.
4. VAX BASIC signals the error “Out of data” (ERR=57) if there are fewer data elements than READ statements. Extra data elements are ignored.
5. The data type of the value must agree with the data type of the variable to which it is assigned or VAX BASIC signals “Data format error” (ERR=50).

6. If you read a string variable, and the DATA element is an unquoted string, VAX BASIC ignores leading and trailing spaces. If the DATA element contains any commas, they must be inside quotation marks.
7. VAX BASIC evaluates subscript expressions in the variable list after it assigns a value to the preceding variable, and before it assigns a value to the subscripted variable. For instance, in the following example, VAX BASIC assigns the value of 10 to variable A, then assigns the string, LESTER, to array element A\$(A).

```
READ A, A$(10)
      .
      .
      .
DATA 10, LESTER
```

The string, LESTER, will be assigned to A\$(10).

---

## Example

```
DECLARE STRING A,B,C
READ A,B,C
DATA "X", "Y", "Z"
PRINT A + B + C
```

### Output

```
XYZ
```

# REAL

---

## REAL

The REAL function converts a numeric expression or numeric string to a specified or default floating-point data type.

---

### Format

$$real-var = \text{REAL} (exp \left[ \begin{array}{l} , \text{SINGLE} \\ , \text{DOUBLE} \\ , \text{GFLOAT} \\ , \text{HFLOAT} \end{array} \right] )$$

---

### Syntax Rules

*Exp* can be either numeric or string. If a string, it can contain the ASCII digits 0 through 9, uppercase E, a plus sign (+), a minus sign (-), and a period (.).

---

### Remarks

1. VAX BASIC evaluates *exp*, then converts it to the specified REAL size. If you do not specify a size, VAX BASIC uses the default REAL size.
2. VAX BASIC ignores leading and trailing spaces and tabs if *exp* is a string.
3. The REAL function returns a value of zero when a string argument contains only spaces and tabs, or when the argument is null.

---

**Example**

```
DECLARE STRING any_num
INPUT "Enter a number";any_num
PRINT REAL(any_num, DOUBLE)
```

**Output**

```
Enter a number? 123095959
.123096E+09
```

# RECORD

---

## RECORD

The RECORD statement lets you name and define data structures in a VAX BASIC program and provides the VAX BASIC interface to the VAX Common Data Dictionary/Plus (CDD/Plus). You can use the defined RECORD name anywhere a VAX BASIC data type keyword is valid if all data types are valid in that context.

---

### Format

**RECORD** *rec-name*  
*rec-component*

.  
.  
.

**END RECORD** [*rec-name*]

*rec-component*: { *data-type rec-item* [, ... ]  
*group-clause*  
*variant-clause* }

*rec-item*: { *unsubs-var* [= *int-const* ]  
*array* ([ *int-const1 TO* ] *int-const2* , ...) [= *int-const* ]  
**FILL** [ ( *int-const* ) ] [= *int-const* ] }

*group-clause*: **GROUP** *group-name* ([ *int-const1 TO* ] *int-const2*, ... ) ]  
*rec-component*

.  
.  
.

**END GROUP** [*group-name*]

*variant-clause*: **VARIANT**



*case-clause*

.

.

.

**END VARIANT**

*case-clause:*   **CASE**  
                  [ *rec-component* ]

.

.

.

## Syntax Rules

1. Each line of text in a RECORD, GROUP, or VARIANT block can have an optional line number.
2. *Data-type* can be a VAX BASIC data type keyword or a previously defined RECORD name. Table 1–2 lists and describes VAX BASIC data type keywords.
3. If the data type of a *rec-item* is STRING, the string is fixed-length. You can supply an optional string length with the = *int-const* clause. If you do not specify a string length, the default is 16.
4. When you create an array of components with GROUP or create an array as a *rec-item*, VAX BASIC allows you to specify both lower and upper bounds. The upper bound is required; the lower bound is optional.
  - *Int-const1* specifies the lower bounds of the array.
  - *Int-const2* specifies the upper bounds of the array and when accompanied by *int-const1*, must be preceded by the keyword TO.
  - *Int-const1* must be less than or equal to *int-const2*.
  - If you do not specify *int-const1*, VAX BASIC uses zero as the default lower bound.

# RECORD

---

## Remarks

1. The total size of a RECORD cannot exceed 65,535 bytes. Also, a RECORD that is used as an array component is limited to 32,767 bytes.
2. The declarations between the RECORD statement and the END RECORD statement are called a RECORD block.
3. Variables and arrays in a RECORD definition are also called RECORD components.
4. The RECORD statement names and defines a data structure called a RECORD template, but does not allocate any storage. When you use the RECORD template as a data type in a statement such as DECLARE, MAP, or COMMON, you declare a RECORD instance. This declaration of the RECORD instance allocates storage for the RECORD. For example:

```
DECLARE EMPLOYEE emp_rec
```

This statement declares a variable named *emp\_rec*, which is an instance of the user-defined data type EMPLOYEE.

5. **Rec-item**
  - The *rec-name* qualifies the *group-name* and the *group-name* qualifies the *rec-item*. You can access a particular *rec-item* within a record by specifying *rec-name::group-name::rec-item*. This specification is called a fully qualified reference. The full qualification of a *rec-item* is also called a component path name.
  - *Rec-item* must conform to the rules for naming VAX BASIC variables.
  - Whenever you access an elementary record component, that is, a variable named in a RECORD definition, you do it in the context of the record instance. Therefore, *rec-item* names need not be unique in your program. For example, you can have a variable called *first\_name* in any number of different RECORD definitions. However, you cannot use a VAX BASIC reserved keyword as a *rec-item* name and you cannot have two variables or arrays with the same name at the same level in the RECORD or GROUP definition.

- The *group-name* is optional in a *rec-item* specification unless there is more than one *rec-item* with the same name or the *group-name* has subscripts. For example:

```
DECLARE EMPLOYEE Emp_rec
    .
    .
RECORD Address
    STRING Street, City, State, Zip
END RECORD Address
RECORD Employee
    GROUP Emp_name
        STRING First = 15
        STRING Middle = 1
        STRING Last = 15
    END GROUP Emp_name
    ADDRESS Work
    ADDRESS Home
END RECORD Employee
```

You can access the *rec-item* “Last” by specifying only “Emp\_rec::Last” because only one *rec-item* is named “Last”. However, if you try to reference “Emp\_rec::City”, VAX BASIC signals an error because “City” is an ambiguous field, a component of both “Work” and “Home”. To access “City”, you must specify either “Emp\_rec::Work::City” or “Emp\_rec::Home::City”.

## 6. Group-clause

- The declarations between the GROUP keyword and the END GROUP keywords are called a GROUP block. The GROUP keyword is valid only within a RECORD block.
- A subscripted group is similar to an array within the record. The group can have both lower and upper bounds for one or more dimensions. Each group element consists of all the record items contained within the including other groups.

## 7. Variant-clause

- The declarations between the VARIANT keyword and the END VARIANT keywords are called a VARIANT block.
- The amount of space allocated for a VARIANT field in a RECORD is equal to the space needed for the variant field requiring the most storage.

# RECORD

- A variant defines the record items that overlay other items, allowing you to redefine the same storage one or more ways.

## 8. Case-clause

- Each case in a variant starts at the position in the record where the variant begins.
- The size of a variant is the size of the longest case in that variant.

---

## Example

```
1000  RECORD Employee
      GROUP Emp_name
          STRING Last = 15
          STRING First = 14
          STRING Middle = 1
      END GROUP Emp_name
      GROUP Emp_address
          STRING Street = 15
          STRING City = 20
          STRING State = 2
          DECIMAL(5,0) Zip
      END GROUP Emp_address
      STRING Wage_class = 2
      VARIANT
          CASE
              GROUP Hourly
                  DECIMAL(4,2) Hourly_wage
                  SINGLE Regular_pay_ytd
                  SINGLE Overtime_pay_ytd
              END GROUP Hourly
          CASE
              GROUP Salaried
                  DECIMAL(7,2) Yearly_salary
                  SINGLE Pay_ytd
              END GROUP Salaried
          CASE
              GROUP Executive
                  DECIMAL(8,2) Yearly_salary
                  SINGLE Pay_ytd
                  SINGLE Expenses_ytd
              END GROUP Executive
          END VARIANT
      END RECORD Employee
```

---

## RECOUNT

The RECOUNT function returns the number of characters transferred by the last input operation.

---

### Format

*int-var* = RECOUNT

---

### Syntax Rules

None.

---

### Remarks

1. The RECOUNT value is reset by every input operation on any channel, including channel #0.
  - After an input operation from your terminal, RECOUNT contains the number of characters (bytes), including line terminators, transferred.
  - After accessing a file record, RECOUNT contains the number of characters in the record.
2. Because RECOUNT is reset by every input operation on any channel, you should copy the RECOUNT value to a different storage location before executing another input operation.
3. If an error occurs during an input operation, the value of RECOUNT is undefined.
4. RECOUNT is unreliable after a CTRL/C interrupt because the CTRL/C trap may have occurred before VAX BASIC set the value for RECOUNT.
5. The RECOUNT function returns a LONG value.

# RECOUNT

---

## Example

```
DECLARE INTEGER character_count
INPUT "Enter a sequence of numeric characters";character_count
character_count = RECOUNT
PRINT character_count;"characters received (including CR and LF)"
```

### Output

```
Enter a sequence of numeric characters? 12345678
10 characters received (including CR and LF)
```

---

## **REM**

The REM statement allows you to document your program.

---

### **Format**

**REM** *[comment]*

---

### **Syntax Rules**

1. REM must be the only statement on the line or the last statement on a multi-statement line.
2. VAX BASIC interprets every character between the keyword REM and the next line number as part of the comment.
3. VAX BASIC does not allow you to specify the REM statement in programs that do not contain line numbers.

---

### **Remarks**

1. Because the REM statement is not executable, you can place it anywhere in a program, except where other statements, such as SUB and END SUB, must be the first or last statement in a program unit.
2. When the REM statement is the first statement on a line-numbered line, VAX BASIC treats any reference to that line number as a reference to the next higher-numbered executable statement.
3. The REM statement is similar to the comment field that begins with an exclamation point, with one exception: the REM statement must be the last statement on a BASIC line. The exclamation point comment field can be ended with another exclamation point or a line terminator and followed by a VAX BASIC statement. See Chapter 1 of this manual for more information on the comment field.

# REM

---

## Example

```
10 REM This is a multi-line comment
   All text up to BASIC line 20
   is part of this REM statement.
   Any BASIC statements on line 10
   are ignored. PRINT "This does not
   execute".
20 PRINT "This will execute"
```

### Output

This will execute



---

## REMAP

The REMAP statement defines or redefines the position in the storage area of variables named in the MAP DYNAMIC statement.

---

### Format

**REMAP** (*map-dyn-name*) *remap-item*,...

*map-dyn-name*: { *map-name*  
*static-str-var* }

*remap-item*: { *num-var*  
*num-array-name* ( [ *int-exp*,... ] )  
*str-var* [ = *int-exp* ]  
*str-array-name* ( [ *int-exp*,... ] ) [ = *int-exp* ]  
[ *data-type* ] **FILL** [ ( *int-exp* ) ] [ = *int-exp* ]  
**FILL%** [ ( *int-exp* ) ]  
**FILL\$** [ ( *int-exp* ) ] [ = *int-exp* ] }

---

### Syntax Rules

1. *Map-dyn-name* can be either a map name or a static string variable.
  - *Map-name* is the storage area named in a MAP statement.
  - If you specify a map name, then a MAP statement with the same name must precede both the MAP DYNAMIC statement and the REMAP statement.
  - When you specify a static string variable, the string must be declared before you can specify a MAP DYNAMIC statement or a REMAP statement.

# REMAP

- If you specify a *static-str-var*, the following restrictions apply:
  - *Static-str-var* cannot be a string constant.
  - *Static-str-var* cannot be the same as any previously declared *map-item* in a MAP DYNAMIC statement.
  - If *static-str-var* is a parameter to the subprogram containing the REMAP statement, *static-str-var* cannot be a RECORD component.
  - *Static-str-var* cannot be a subscripted variable.
  - *Static-str-var* cannot be a parameter declared in a DEF or DEF\* function.
- 2. *Remap-item* names a variable, array, or array element declared in a preceding MAP DYNAMIC statement:
  - *Num-var* specifies a numeric variable or array element. *Num-array-name* followed by a set of empty parentheses specifies an entire numeric array.
  - *Str-var* specifies a string variable or array element. *Str-array-name* followed by a set of empty parentheses, specifies an entire fixed-length string array. You can specify the number of bytes to be reserved for string variables and array elements with the *=int-exp* clause. The default string length is 16.
- 3. *Remap-item* can also be a FILL item. The FILL, FILL%, and FILL\$ keywords let you reserve parts of the record buffer. *Int-exp* specifies the number of FILL items to be reserved. The *=int-exp* clause allows you to specify the number of bytes to be reserved for string FILL items. Table 4–2 describes FILL item format and storage allocation.

## NOTE

In the FILL clause, (*int-exp*) represents a repeat count, not an array subscript. FILL (*n*), for example, represents *n* elements, not *n* + 1.

- 4. All *remap-items*, except FILL items, must have been named in a previous MAP DYNAMIC statement, or VAX BASIC signals an error.

5. *Data-type* can be any VAX BASIC data type keyword or a data type defined in a RECORD statement. Data type keywords and their size, range, and precision are listed in Table 1–2 in this manual. You can specify a data type only for FILL items.
  - When you specify a data type before a FILL keyword in a REMAP statement, the FILL item is of that data type. The specified data type applies only to that one FILL item.
  - If you do not specify any data type for a FILL item, the FILL item takes the current default data type and size.
6. *Remap-items* must be separated with commas.

---

## Remarks

1. The REMAP statement does not affect the amount of storage allocated to the map area.
2. Each time a REMAP statement executes, VAX BASIC sets record pointers to the named map area for the specified variables from left to right.
3. The REMAP statement must be preceded by a MAP DYNAMIC statement or VAX BASIC signals the error “No such MAP area <name>”. The MAP statement or static string variable creates a named area of static storage, the MAP DYNAMIC statement specifies the variables whose positions can change at runtime, and the REMAP statement specifies the new positions for the variables named in the MAP DYNAMIC statement.
4. Before you can specify a map name in a REMAP statement, there must be a MAP statement in the program unit with the same map name. Otherwise, VAX BASIC signals the error “<Name> is not a DYNAMIC MAP variable of MAP <name>”. Similarly, before you can specify a static string variable in a REMAP statement, the string variable must be declared. Otherwise, VAX BASIC signals the same error message.
5. If a static string variable is the same as a map name, VAX BASIC overrides the static string name and uses the map name.

## REMAP

6. Until the REMAP statement executes, all variables named in the MAP DYNAMIC statement point to the first byte of the MAP area and all string variables have a length of zero. When the REMAP statement executes, VAX BASIC sets the internal pointers as specified in the REMAP statement. For example:

```
100      MAP (DUMMY) STRING map_buffer = 50
        MAP DYNAMIC (DUMMY) LONG A, STRING B, SINGLE C(7)
        REMAP (DUMMY) B=14, A, C()
```

The REMAP statement sets a pointer to byte 1 of *DUMMY* for string variable *B*, a pointer to byte 15 for LONG variable *A*, and pointers to bytes 19, 23, 27, 31, 35, 39, 43, and 47 for the elements in SINGLE array *C*.

7. You can use the REMAP statement to redefine the pointer for an array element or variable more than once in a single REMAP statement. For example:

```
100      MAP (DUMMY) STRING FILL = 48
        MAP DYNAMIC (DUMMY) LONG A, B(10)
        REMAP (DUMMY) B(), B(0)
```

This REMAP statement sets a pointer to byte 1 in *DUMMY* for array *B*. Because array *B* uses a total of 44 bytes, the pointer for the first element of array *B*, *B*(0) points to byte 45. References to array element *B*(0) will be to bytes 45 through 48. Pointers for array elements 1 through 10 are set to bytes 5, 9, 13, 17 and so forth.

8. Because the REMAP statement is local to a program module, it affects pointers only in the program module in which it executes.

---

## Examples

**Example 1**

```

DECLARE LONG CONSTANT emp_fixed_info = 4 + 9 + 2
MAP (emp_buffer) LONG badge,           &
                STRING social_sec_num = 9,      &
                BYTE name_length,             &
                address_length,              &
                FILL (60)

MAP DYNAMIC (emp_buffer) STRING emp_name,      &
                emp_address

WHILE 1%
    GET #1
    REMAP (emp_buffer) STRING FILL = emp_fixed_info,      &
                emp_name = name_length,                  &
                emp_address = address_length

    PRINT emp_name
    PRINT emp_address
    PRINT

NEXT
END

```

**Example 2**

```

SUB deblock (STRING input_rec, STRING item())
MAP DYNAMIC (input_rec) STRING A(1 TO 3)
REMAP (input_rec) &
    A(1) = 5, &
    A(2) = 3, &
    A(3) = 4
FOR I = LBOUND(A) TO UBOUND(A)
    item(I) = A(I)
NEXT I
END SUB

```

## RESET

---

## RESET

The **RESET** statement is a synonym for the **RESTORE** statement. See the **RESTORE** statement for more information.

---

### Format

**RESET** [*#chnl-exp* [, **KEY** *#int-exp* ]]

---

## RESTORE

The RESTORE statement resets the DATA pointer to the beginning of the DATA sequence, or sets the record pointer to the first record in a file.

---

### Format

**RESTORE** [*#chnl-exp* [, **KEY** *#int-exp* ]]

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Int-exp* must be between zero and the number of keys in the file minus 1. It must be immediately preceded by a number sign (#).

---

### Remarks

1. If you do not specify a channel, RESTORE resets the DATA pointer to the beginning of the DATA sequence.
2. RESTORE affects only the current program unit. Thus, executing a RESTORE statement in a subprogram does not affect the DATA pointer in the main program.
3. If there is no channel specified, and the program has no DATA statements, RESTORE has no effect.
4. The file specified by *chnl-exp* must be open.
5. If *chnl-exp* specifies a magnetic tape file, VAX BASIC rewinds the tape to the first record in the file.

# RESTORE

6. The **KEY** clause applies to indexed files only. It sets a new key of reference equal to *int-exp* and sets the next record pointer to the first logical record in that key.
7. For indexed files, the **RESTORE** statement without a **KEY** clause sets the next record pointer to the first logical record specified by the current key of reference. If there is no current key of reference, the **RESTORE** statement sets the next record pointer to the first logical record of the primary key.
8. If you use the **RESTORE** statement on any file type other than indexed, **VAX BASIC** sets the next record pointer to the first record in the file.
9. The **RESTORE** statement is not allowed on virtual array files or on files opened on unit record devices.
10. For information on the **RESTORE GRAPHICS** statement, see *Programming with VAX BASIC Graphics*.

---

## Example

```
RESTORE #7%, KEY #4%
```



---

## RESUME

The **RESUME** statement marks an exit point from an **ON ERROR** error-handling routine. **VAX BASIC** clears the error condition and returns program control to a specified line number, label or to the program block in which the error occurred.

### NOTE

The **RESUME** statement is supported for compatibility with other **DIGITAL BASICs**. For new program development, **DIGITAL** recommends that you use **WHEN** blocks.

---

### Format

**RESUME** [ *target* ]

---

### Syntax Rules

*Target* must be a valid **VAX BASIC** line number or label and must exist in the same program unit.

---

### Remarks

1. The following restrictions apply:
  - The **RESUME** statement cannot appear within a protected region, or within an attached or detached handler.
  - The target of a **RESUME** statement cannot exist within a protected region or handler.
  - The **RESUME** statement cannot be used in a multi-line **DEF** unless the target is also in the **DEF** function definition.

# RESUME

- The execution of a **RESUME** with no target is illegal if there is no error active.
  - A **RESUME** statement cannot transfer control out of the current program unit. Therefore, a **RESUME** statement with no target cannot terminate an error handler if the error handler is handling an error that occurred in a subprogram or an external function, and the error was passed to the calling program's error handler by an **ON ERROR GO BACK** statement or by default.
2. When no target is specified in a **RESUME** statement, **VAX BASIC** transfers control based on where the error occurs. If the error occurs on a numbered line containing a single statement, **VAX BASIC** always transfers control to that statement. When the error occurs within a multi-statement line under the following conditions, **VAX BASIC** acts as follows:
    - Within a **FOR**, **WHILE**, or **UNTIL** loop, **VAX BASIC** transfers control to the first statement that follows the **FOR**, **WHILE**, or **UNTIL** statement.
    - Within a **SELECT** block, **VAX BASIC** transfers control to the start of the **CASE** block in which the error occurs.
    - After a loop or **SELECT** block, **VAX BASIC** transfers control to the statement that follows the **NEXT** or **END SELECT** statement.
    - If none of the above conditions occurs, **VAX BASIC** transfers control back to the statement that follows the most recent line number.
  3. A **RESUME** statement with a specified line number transfers control to the first statement of a multi-statement line, regardless of which statement caused the error.
  4. A **RESUME** statement with a specified label transfers control to the block of code indicated by that label.

---

## Example

```
Error_routine:
IF ERR = 11
  THEN
    CLOSE #1
    RESUME end_of_prog
  ELSE
    RESUME
  END IF
end_of_prog: END
```

# RETRY

---

## RETRY

The **RETRY** statement clears an error condition and reexecutes the statement that caused the error inside a protected region of a **WHEN** block.

---

### Format

**RETRY**

---

### Syntax Rules

The **RETRY** statement must appear lexically inside of a handler associated with a **WHEN** block.

---

### Remarks

The following rules apply to errors that occur during execution of loop control statements (not the statements inside the loop body):

- In **FOR...NEXT** loops, the **RETRY** statement reexecutes the **FOR** statement if the error occurs while **VAX BASIC** is evaluating the limit or increment values.
- In **FOR...NEXT** loops, if the error occurs while **VAX BASIC** is evaluating the index variable, the **RETRY** statement reexecutes the **NEXT** statement.
- In a **FOR...UNTIL** or **FOR...WHILE** loop, if an error occurs while **VAX BASIC** is evaluating the relational expression, the **RETRY** statement reexecutes the **NEXT** statement.

---

## Example

```
10 DECLARE LONG YOUR_AGE
   WHEN ERROR IN
       INPUT "Enter your age";your_age
   USE
       IF ERR = 50
           THEN RETRY
           ELSE EXIT HANDLER
       END IF
   END WHEN
```

# RETURN

---

## RETURN

The RETURN statement transfers control to the statement immediately following the most recently executed GOSUB or ON...GOSUB statement in the current program unit.

---

### Format

**RETURN**

---

### Syntax Rules

None.

---

### Remarks

1. Once the RETURN is executed in a subroutine, no other statements in the subroutine are executed, even if they appear after the RETURN statement.
2. Execution of a RETURN statement before the execution of a GOSUB or ON...GOSUB causes VAX BASIC to signal "RETURN without GOSUB" (ERR=72).

---

## Example

```
GOSUB subroutine_1
.
.
.
subroutine_1:
.
.
.
RETURN
```

# RIGHT\$

---

## RIGHT\$

The RIGHT\$ function extracts a substring from a string's right side, leaving the string unchanged.

---

### Format

*str-var* = RIGHT[\$] (*str-exp*, *int-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. The RIGHT\$ function extracts a substring from *str-exp* and stores the substring in *str-var*. The substring begins with the character in the position specified by *int-exp* and ends with the rightmost character in the string.
2. If *int-exp* is less than or equal to zero, RIGHT\$ returns the entire string.
3. If *int-exp* is greater than the length of *str-exp*, RIGHT\$ returns a null string.
4. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to a LONG integer.



---

**Example**

```
DECLARE STRING main_str,  &  
            end_result  
main_str = "1234567"  
end_result = RIGHT$(main_str, 3)  
PRINT end_result
```

**Output**

34567

# RMSSTATUS

---

## RMSSTATUS

The RMSSTATUS function returns the RMS status or the status value of the last I/O operation on a specified open I/O channel.

---

### Format

$$\text{long-var} = \text{RMSSTATUS} (\text{chnl-exp} \left[ \begin{array}{l} \text{, STATUS} \\ \text{, VALUE} \end{array} \right] )$$

---

### Syntax Rules

1. *Chnl-exp* must be the number of a channel opened from a VAX BASIC routine.
2. *Chnl-exp* cannot be zero.

---

### Remarks

1. If *chnl-exp* does not represent an open channel, VAX BASIC signals the error "I/O channel not open" (ERR=9).
2. If you do not specify either *STATUS* or *VALUE*, RMSSTATUS returns the *STATUS* value by default.
3. If you specify *STATUS*, RMSSTATUS returns the FAB\$L\_STS or the RAB\$L\_STS status value. However, if you specify *VALUE*, RMSSTATUS returns the FAB\$L\_STV or the RAB\$L\_STV status value.
4. Use the RMSSTATUS function to return the status of the following operations:
  - RESTORE
  - GET

- PUT
- UPDATE
- UNLOCK
- PRINT and PRINT USING
- INPUT, INPUT LINE, and LINPUT
- SCRATCH
- FREE
- Virtual array references

To determine the reason for the failure of an OPEN, CLOSE, KILL, or NAME...AS statement, use the RMSSTATUS function within an error handler.

---

## Examples

### Example 1

```
%TITLE "RMSSTATUS Example"
%SBTTL "Reference Manual Examples"
%IDENT "V1.0"

PROGRAM Demo_RMSSTATUS_function
  OPTION CONSTANT TYPE = INTEGER

  OPEN "DOES_NOT_EXIST.LIS" FOR OUTPUT AS 1, &
    SEQUENTIAL VARIABLE, &
    RECORDSIZE 80

  WHEN ERROR IN
    GET #1
  USE
    PRINT "GET Operation failed"
    PRINT "RMS Status ="; RMSSTATUS(1,STATUS)
    PRINT "RMS Status Value ="; RMSSTATUS(1,VALUE)
  END WHEN
END PROGRAM
```

# RMSSTATUS

## Example 2

```
OPTION TYPE=EXPLICIT
EXTERNAL LONG CONSTANT RMS$_OK_DUP

MAP (ORDER) LONG ORD_ENTRY, STRING ORD_CUST_NO = 6%,    &
  STRING ORD_REMARK = 50%

OPEN "ORD_DB" FOR INPUT AS FILE 1%,                      &
  ORGANIZATION INDEXED FIXED,                            &
  MAP ORDER,                                              &
  PRIMARY ORD_ENTRY NODUPLICATES,                        &
  ALTERNATE ORD_CUST_NO DUPLICATES

INPUT "Enter order number";ORD_ENTRY
INPUT "Enter customer number";ORD_CUST_NO
INPUT "Remark";ORD_REMARK

!
! Enter the order in the order database
! Check if the customer has other orders
!
PUT #1%
IF RMSSTATUS( 1%, STATUS ) = RMS$_OK_DUP
THEN
  !
  ! The customer has other orders; compute the customer's
  ! discount for other orders
  !
END IF

CLOSE 1%
END
```

---

## **RND**

The RND function returns a random number greater than or equal to zero and less than 1.

---

### **Format**

*real-var* = RND

---

### **Syntax Rules**

None.

---

### **Remarks**

1. If the RND function is preceded by a RANDOMIZE statement, VAX BASIC generates a different random number or series of numbers each time a program executes.
2. The RND function returns a pseudorandom number if not preceded by a RANDOMIZE statement; that is, each time a program runs, VAX BASIC generates the same random number or series of random numbers.
3. The RND function returns a floating-point SINGLE value.
4. The RND function returns values over a uniform distribution between 0 and 1. For example, a value between 0 and .1 is as likely as a value between .5 and .6. Note the difference between this and a bell-curve distribution where the probability of values in the range .3 to .7 is higher than the outer ranges.

# RND

---

## Example

```
DECLARE REAL random_num
RANDOMIZE
FOR I = 1 TO 3   !FOR loop causes VAX BASIC to print three random numbers
    random_num = RND
    PRINT random_num
NEXT I
```

### Output

```
.865243
.477417
.734673
```

---

## **RSET**

The RSET statement assigns right-justified data to a string variable. RSET does not change a string variable's length.

---

### **Format**

**RSET** *str-var,... = str-exp*

---

### **Syntax Rules**

*Str-var* cannot be a DEF function name unless the RSET statement is inside the DEF function definition.

---

### **Remarks**

1. The RSET statement treats strings as fixed-length. It does not change the length of *str-var*, nor does it create new storage locations.
2. If *str-var* is longer than *str-exp*, RSET right-justifies the data and pads it with spaces on the left.
3. If *str-var* is shorter than *str-exp*, RSET truncates *str-exp* on the left.

# RSET

---

## Example

```
DECLARE STRING test
test = "ABCDE"
RSET test = "123"
PRINT "X" + test
```

### Output

```
X 123
```



---

## SCRATCH

The SCRATCH statement deletes the current record and all following records in a sequential file.

---

### Format

**SCRATCH** *#chnl-exp*

---

### Syntax Rules

*Chnl-exp* is a numeric expression that specifies a channel associated with a file. It must be immediately preceded by a number sign (#).

---

### Remarks

1. Before you execute the SCRATCH statement, the file must be opened with ACCESS SCRATCH.
2. The SCRATCH statement applies to ORGANIZATION SEQUENTIAL files only.
3. The SCRATCH statement has no effect on terminals or unit record devices.
4. For disk files, the SCRATCH statement discards the current record and all that follows it in the file. The physical length of the file does not change.
5. For magnetic tape files, the SCRATCH statement overwrites the current record with two end-of-file marks.
6. Use of the SCRATCH statement on shared sequential files is not recommended.

# SCRATCH

---

## Example

SCRATCH #4%

---

## SEG\$

The SEG\$ function extracts a substring from a main string, leaving the original string unchanged.

---

### Format

*str-var* = SEG\$ (*str-exp*, *int-exp1*, *int-exp2*)

---

### Syntax Rules

None.

---

### Remarks

1. VAX BASIC extracts the substring from *str-exp*, the main string, and stores the substring in *str-var*. The substring begins with the character in the position specified by *int-exp1* and ends with the character in the position specified by *int-exp2*.
2. If *int-exp1* is less than 1, VAX BASIC assumes a value of 1.
3. If *int-exp1* is greater than *int-exp2* or the length of *str-exp*, the SEG\$ function returns a null string.
4. If *int-exp1* equals *int-exp2*, the SEG\$ function returns the character at the position specified by *int-exp1*.
5. Unless *int-exp2* is greater than the length of *str-exp*, the length of the returned substring equals *int-exp2* minus *int-exp1* plus 1. If *int-exp2* is greater than the length of *str-exp*, the SEG\$ function returns all characters from the position specified by *int-exp1* to the end of *str-exp*.
6. If you specify a floating-point expression for *int-exp1* or *int-exp2*, VAX BASIC truncates it to LONG integer.

# SEG\$

---

## Example

```
DECLARE STRING alpha, center  
alpha = "ABCDEFGHIJK"  
center = SEG$(alpha, 4, 8)  
PRINT center
```

### Output

```
DEFGH
```

---

## SELECT

The **SELECT** statement lets you specify an expression, a number of possible values the expression may have, and a number of alternative statement blocks to be executed for each possible case.

---

### Format

```
SELECT  exp1
         case-clause
         .
         .
         .
         [ else-clause ]
```

### END SELECT

*case-clause*: **CASE** *case-item*,...  
              [ *statement* ]...

*case-item*: { [ *rel-op* ] *exp2*  
              [ *exp3 TO exp4* [ , *exp5 TO exp6* ], ... } }

*else-clause*: **CASE ELSE**  
              [ *statement* ]...

---

## Syntax Rules

1. *Exp1* is the expression to be tested against the *case-clauses* and the *else-clause*. It can be numeric or string.
2. *Case-clause* consists of the **CASE** keyword followed by a *case-item* and statements to be executed when the *case-item* is true.

# SELECT

3. *Else-clause* consists of the CASE ELSE keywords followed by statements to be executed when no previous *case-item* has been selected as true.
4. *Case-item* is either an expression to be compared with *exp1* or a range of values separated with the keyword TO.
  - *Rel-op* is a relational operator specifying how *exp1* is to be compared to *exp2*. If you do not include a *rel-op*, VAX BASIC assumes the equals (=) operator. VAX BASIC executes the statements in the CASE block when the specified relational expression is true.
  - *Exp3* and *exp4* specify a range of numeric or string values separated by the keyword TO. Multiple ranges must be separated with commas. VAX BASIC executes the statements in the CASE block when *exp1* falls within any of the specified ranges.

---

## Remarks

1. A SELECT statement can have only one *else-clause*. The *else-clause* is optional and, when present, must be the last CASE block in the SELECT block.
2. Each statement in a SELECT block can have its own line number.
3. The SELECT statement begins the SELECT BLOCK and the END SELECT keywords terminate it. VAX BASIC signals an error if you do not include the END SELECT keywords.
4. Each CASE keyword establishes a CASE block. The next CASE or END SELECT keyword ends the CASE block.
5. You can nest SELECT blocks within a CASE or CASE ELSE block.
6. VAX BASIC evaluates *exp1* when the SELECT statement is first encountered; VAX BASIC then compares *exp1* with each *case-clause* in order of occurrence until a match is found or until a CASE ELSE block or END SELECT is encountered.
7. The following conditions constitute a match:
  - *Exp1* satisfies the relationship to *exp2* specified by *rel-op*.
  - *Exp1* is greater than or equal to *exp3*, but less than or equal to *exp4*, greater than or equal to *exp5* but less than or equal to *exp6*, and so on.

## SELECT

8. When a match is found between *exp1* and a *case-item*, VAX BASIC executes the statements in the CASE block where the match occurred. If ranges overlap, the first match causes VAX BASIC to execute the statements in the CASE block. After executing CASE block statements, control passes to the statement immediately following the END SELECT keywords.
9. If no CASE match occurs, VAX BASIC executes the statements in the *else-clause*, if present, and then passes control to the statement immediately following the END SELECT keywords.
10. If no CASE match occurs and you do not supply a *case-else* clause, control passes to the statement following the END SELECT keywords.

---

### Example

```
100  SELECT A% + B% + C%
      CASE = 100
          PRINT 'THE VALUE IS EXACTLY 100'
      CASE 1 TO 99
          PRINT 'THE VALUE IS BETWEEN 1 AND 99'
      CASE > 100
          PRINT 'THE VALUE IS GREATER THAN 100'
      CASE ELSE
          PRINT 'THE VALUE IS LESS THAN 1'
      END SELECT
```

## SET PROMPT

---

## SET PROMPT

The **SET PROMPT** statement enables a question mark prompt to appear after **VAX BASIC** executes either an **INPUT**, **LINPUT**, **INPUT LINE**, **MAT INPUT**, or **MAT LINPUT** statement on channel #0. The **SET NO PROMPT** statement disables the question mark prompt.

---

### Format

**SET [NO] PROMPT**

---

### Syntax Rules

None.

---

### Remarks

1. If you do not specify a **SET PROMPT** statement, the default is **SET PROMPT**.
2. **SET NO PROMPT** disables **VAX BASIC** from issuing a question mark prompt for the **INPUT**, **LINPUT**, **INPUT LINE**, **MAT INPUT**, and **MAT LINPUT** statements on channel #0.
3. Prompting is reenabled when either a **SET PROMPT** statement or a **CHAIN** statement is executed, or when a **NEW**, **OLD**, **RUN** or **SCRATCH** command is executed in the **BASIC** environment.
4. The **SET NO PROMPT** statement does not affect the string constant you specify as the input prompt with the **INPUT** statement.



---

## Example

```
DECLARE STRING your_name, your_age, your_grade
INPUT "Enter your name";your_name
SET NO PROMPT
INPUT "Enter your age"; your_age
SET PROMPT
INPUT "Enter the last school grade you completed";your_grade
```

### Output

```
Enter your name? Katherine Kelly
Enter your age 15
Enter the last school grade you completed? 9
```

# SGN

---

## SGN

The SGN function determines whether a numeric expression is positive, negative, or zero. It returns a 1 if the expression is positive, a -1 if the expression is negative, and zero if the expression is zero.

---

### Format

*int-var* = **SGN** (*real-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. If *real-exp* does not equal zero, SGN returns  $\text{MAG}(\text{real-exp})/\text{real-exp}$ .
2. If *real-exp* equals zero, SGN returns a value of zero.
3. SGN returns a LONG integer.

---

### Example

```
DECLARE INTEGER sign
sign = SGN(46/23)
PRINT sign
```

#### Output

1

---

**SIN**

The SIN function returns the sine of an angle in radians or degrees.

---

**Format**

*real-var* = **SIN** (*real-exp*)

---

**Syntax Rules**

*Real-exp* is an angle specified in radians or degrees depending upon which angle clause you choose with the OPTION statement.

---

**Remarks**

1. The returned value is between -1 and 1.
2. VAX BASIC expects the argument of the SIN function to be a real expression. When the argument is a real expression, VAX BASIC returns a value of the same floating-point size. When the argument is not a real expression, VAX BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

# SIN

---

## Example

```
OPTION ANGLE = RADIANS
DECLARE REAL s1_angle
s1_angle = SIN(PI/2)
PRINT s1_angle
```

### Output

1

---

## SLEEP

The SLEEP statement suspends program execution for a specified number of seconds or until a carriage return is entered from the controlling terminal.

---

### Format

**SLEEP** *int-exp*

---

### Syntax Rules

1. *Int-exp* is the number of seconds VAX BASIC waits before resuming program execution.
2. *Int-exp* must be between 0 and 2,147,483,647; if it is greater than 2,147,483,647, VAX BASIC signals the error "Integer error or overflow" (ERR=51).

---

### Remarks

1. Pressing the RETURN key on the controlling terminal cancels the effect of the SLEEP statement.
2. All characters typed while SLEEP is in effect, including a RETURN character entered to terminate the SLEEP statement, will remain in the typeahead buffer. Therefore, if you type RETURN without preceding data, an INPUT statement that follows the SLEEP will complete without data.

# SLEEP

---

## Example

SLEEP 120%

---

## SPACE\$

The SPACE\$ function creates a string containing a specified number of spaces.

---

### Format

*str-var* = SPACE\$ (*int-exp*)

---

### Syntax Rules

*Int-exp* specifies the number of spaces in the returned string.

---

### Remarks

1. VAX BASIC treats an *int-exp* less than 0 as zero.
2. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to a LONG integer.

---

### Example

```
DECLARE STRING A, B
A = "1234"
B = "5678"
PRINT A + SPACE$(5%) + B
```

#### Output

```
1234      5678
```

## SQR

---

## SQR

The SQR function returns the square root of a positive number.

---

### Format

$$real-var = \left\{ \begin{array}{l} \text{SQRT} \\ \text{SQR} \end{array} \right\} (real-exp)$$

---

### Syntax Rules

None.

---

### Remarks

1. VAX BASIC signals the error “Imaginary square roots” (ERR=54) when *real-exp* is negative.
2. VAX BASIC assumes that the argument of the SQR function is a real expression. When the argument is a real expression, VAX BASIC returns a value of the same floating-point size. When the argument is not a real expression, VAX BASIC returns a value of the default floating-point size.



---

**Example**

```
DECLARE REAL root
root = SQR(20*5)
PRINT root
```

**Output**

10

# STATUS

---

## STATUS

The **STATUS** function returns an integer value containing information about the last opened channel. Your program can test each bit to determine the status of the channel.

### NOTE

The **STATUS** function is supported only for compatibility with other **DIGITAL BASICs**. **DIGITAL** recommends that you use the **RMSSTATUS** function for new program development.

---

### Format

*int-var* = **STATUS**

---

### Syntax Rules

None.

---

### Remarks

1. The **STATUS** function returns a **LONG** integer.
2. The value returned by the **STATUS** function is undefined until **VAX BASIC** executes an **OPEN** statement.
3. The **STATUS** value is reset by every input operation on any channel. Therefore, you should copy the **STATUS** value to a different storage location before your program executes another input operation.

4. If an error occurs during an input operation, the value of STATUS is undefined. When no error occurs, the six low-order bits of the returned value contain information about the type of device accessed by the last input operation. Table 4–6 lists STATUS bits set by VAX BASIC.

**Table 4–6: VAX BASIC STATUS Bits**

<b>Bit Set</b>	<b>Device Type</b>
0	Record-oriented device
1	Carriage-control device
2	Terminal
3	Directory device
4	Single directory device
5	Sequential block-oriented device (magnetic tape)

---

## Example

```
150      Y% = STATUS
```

# STOP

---

## STOP

The **STOP** statement halts program execution allowing you to optionally continue execution.

---

### Format

**STOP**

---

### Syntax Rules

None.

---

### Remarks

1. The **STOP** statement cannot appear before a **PROGRAM**, **SUB** or **FUNCTION** statement.
2. The **STOP** statement does not close files.
3. When a **STOP** statement executes in a program executed with the **RUN** command in the **BASIC** environment, **VAX BASIC** prints the line number and module name associated with the **STOP** statement, then displays the Ready prompt. In response to the prompt, you can type immediate mode statements, **CONTINUE** to resume program execution, or any valid compiler command. See the *VAX BASIC User Manual* for more information on immediate mode.
4. When a **STOP** statement is in an executable image, the line number, module name, and a number sign (#) prompt are printed. In response to the prompt, you can type **CONTINUE** to continue program execution or **EXIT** to end the program. If the program module was compiled with the **/NOLINE** qualifier, no line number is displayed.

---

## Example

```
PROGRAM Stopper
  PRINT "Type CONTINUE when the program stops"
  INPUT "Do you want to stop now"; Quit$

  IF Quit$ = "Y"
  THEN
    STOP
  ELSE
    PRINT "So what are you waiting for?"
    STOP
  END IF

  PRINT "You told me to continue...thank you"
END PROGRAM
```

### Output

```
Do you want to stop now? CONTINUE
So what are you waiting for?
```

## STR\$

---

## STR\$

The STR\$ function changes a numeric expression to a numeric character string without leading and trailing spaces.

---

### Format

*str-var* = STR\$ (*num-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. If *num-exp* is negative, the first character in the returned string is a minus sign (-).
2. The STR\$ function does not return leading or trailing spaces.
3. When you print a floating-point number that has six decimal digits or more but the integer portion has six digits or less (for example, 1234.567), VAX BASIC rounds the number to six digits (1234.57). If a floating-point number's integer part is seven decimal digits or more, VAX BASIC rounds the number to six digits and prints it in E format.
4. When you print a floating-point number with magnitude between 0.1 and 1, VAX BASIC rounds it to six digits. When you print a number with magnitude smaller than 0.1, VAX BASIC rounds it to six digits and prints it in E format.
5. The STR\$ function returns up to 10 digits for LONG integers and up to 31 digits for DECIMAL numbers.

---

## Example

```
DECLARE STRING new_num  
new_num = STR$(1543.659)  
PRINT new_num
```

### Output

```
1543.66
```

# STRING\$

---

## STRING\$

The **STRING\$** function creates a string containing a specified number of identical characters.

---

### Format

*str-var* = **STRING\$** (*int-exp1*, *int-exp2*)

---

### Syntax Rules

1. *Int-exp1* specifies the character string's length.
2. *Int-exp2* is the decimal ASCII value of the character that makes up the string. This value is treated modulo 256.

---

### Remarks

1. VAX BASIC signals the error "String too long" (ERR=227) if *int-exp1* is greater than 65535.
2. If *int-exp1* is less than or equal to zero, VAX BASIC treats it as zero.
3. VAX BASIC treats *int-exp2* as an unsigned 8-bit integer. For example, -1 is treated as 255.
4. If either *int-exp1* or *int-exp2* is a floating-point expression, VAX BASIC truncates it to a LONG integer.



---

## Example

```
DECLARE STRING output_str
output_str = STRING$(10%, 50%) !50 is the ASCII value of the
PRINT output_str                !character "2"
```

### Output

```
2222222222
```

# SUB

---

## SUB

The SUB statement marks the beginning of a VAX BASIC subprogram and specifies the number and data type of its parameters.

---

### Format

**SUB** *sub-name* [ *pass-mech* ] [ ( [ *formal-param* ],... ) ]  
[ *statement* ]...

{ **END SUB**  
**SUBEND** }

*pass-mech*: { **BY REF**  
**BY DESC** }

*formal-param*: [ *data-type* ] { *unsubs-var*  
*array-name* ( [ *int-const* ] ,... ) }  
[ = *int-const* ] [ *pass-mech* ]

---

### Syntax Rules

1. *Sub-name* is the name of the separately compiled subprogram.
2. *Formal-param* specifies the number and type of parameters for the arguments the SUB subprogram expects to receive when invoked.
  - Empty parentheses indicate that the SUB subprogram has no parameters.

- *Data-type* specifies the data type of a parameter. If you do not specify a data type, parameters are of the default data type and size. When you do specify a data type, all following parameters are of that data type until you specify a new data type. Data type keywords and their size, range, and precision are listed in Table 1–2 in this manual.
3. *Sub-name* can have from 1 to 31 characters and must conform to the following rules:
    - The first character of an unquoted name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs (\$), periods (.), or underscores (\_).
    - A quoted name can consist of any combination of printable ASCII characters.
  4. *Data-type* can be any VAX BASIC data type keyword or a data type defined by a RECORD statement.
  5. *Pass-mech* specifies the parameter passing mechanism by which the subprogram receives arguments.
  6. A *pass-mech* clause outside the parentheses applies by default to all SUB parameters. A *pass-mech* clause in the *formal-param* list overrides the specified default and applies only to the immediately preceding parameter.

---

## Remarks

1. The SUB statement must be the first statement in the SUB subprogram.
2. Compiler directives and comment fields created with an exclamation point (!), can precede the SUB statement because they are not VAX BASIC statements. Note that REM is a VAX BASIC statement; therefore, it cannot precede the SUB statement.
3. Every SUB statement must have a corresponding END SUB statement or SUBEND statement.
4. If you do not specify a passing mechanism, the SUB program receives arguments by the default passing mechanisms, as shown in Table 4–1.

## SUB

5. Parameters defined in *formal-param* must agree in number, type, ordinality, and passing mechanism with the arguments specified in the CALL statement of the calling program.
6. You can specify up to 255 parameters.
7. Any VAX BASIC statement except those that refer to other program unit types (FUNCTION, PICTURE or PROGRAM) can appear in a SUB subprogram.
8. All variables, except those named in MAP and COMMON statements are local to that subprogram.
9. VAX BASIC initializes local variables to zero or the null string.
10. SUB subprograms receive parameters by reference or by descriptor. A SUB subprogram cannot receive any parameter BY VALUE. Table 4–1 lists and describes VAX BASIC parameter-passing mechanisms.
  - BY REF specifies that the subprogram receives the argument's address.
  - BY DESC specifies that the subprogram receives the address of a VAX BASIC descriptor. For information about the format of a VAX BASIC descriptor for strings and arrays, see the *VAX BASIC User Manual*. For information on other types of descriptors, see the *VAX Architecture Handbook*.
11. By default, VAX BASIC subprograms receive numeric unsubscripted variables by reference, and all other parameters by descriptor. You can override these defaults for strings and arrays with a BY clause:
  - If you specify a string length with the *=int-const* clause, you must also specify BY REF. If you specify BY REF and do not specify a string length, VAX BASIC uses the default string length of 16.
  - If you specify array bounds, you must also specify BY REF.
12. Subprograms can be called recursively.

---

**Example**

```
SUB SUB3 BY REF (DOUBLE A, B,      &
  STRING Emp_nam BY DESC,        &
  wage(20))
.
.
.
END SUB
```

## **SUBEND**

---

## **SUBEND**

The **SUBEND** statement is a synonym for the **END SUB** statement. See the **END** statement for more information.

---

### **Format**

**SUBEND**

---

**SUBEXIT**

The **SUBEXIT** statement is a synonym for the **EXIT SUB** statement. See the **EXIT** statement for more information.

---

**Format**

**SUBEXIT**

# SUM\$

---

## SUM\$

The SUM\$ function returns a string whose value is the sum of two numeric strings.

---

### Format

*str-var* = **SUM\$** (*str-exp1*, *str-exp2*)

---

### Syntax Rules

None.

---

### Remarks

1. The SUM\$ function does not support E-format notation.
2. Each string expression can contain up to 54 ASCII digits and an optional decimal point and sign.
3. VAX BASIC adds *str-exp2* to *str-exp1* and stores the result in *str-var*.
4. If *str-exp1* and *str-exp2* are integers, *str-var* takes the precision of the larger string unless trailing zeros generate that precision.
5. If *str-exp1* and *str-exp2* are decimal fractions, *str-var* takes the precision of the more precise fraction, unless trailing zeros generate that precision.
6. SUM\$ omits trailing zeros to the right of the decimal point.
7. The sum of two fractions takes precision as follows:
  - The sum of the integer parts takes the precision of the larger part.
  - The sum of the decimal fraction part takes the precision of the more precise part.



8. **SUM\$** truncates leading and trailing zeros.
- 

## **Example**

```
DECLARE STRING A, B, total
A = "46"
B = "87"
total = SUM$(A,B)
PRINT total
```

### **Output**

133

## SWAP%

---

## SWAP%

The SWAP% function transposes a WORD integer's bytes.

### NOTE

The SWAP% function is supported only for compatibility with BASIC-PLUS-2. DIGITAL recommends that you do not use the SWAP% function for new program development.

---

### Format

*int-var* = SWAP% (*int-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. SWAP% is a WORD function. VAX BASIC evaluates *int-exp* and converts it to the WORD data type, if necessary.
2. VAX BASIC transposes the bytes of *int-exp* and returns a WORD integer.

---

**Example**

```
DECLARE INTEGER word_int
word_int = SWAP%(23)
PRINT word_int
```

**Output**

5888

# TAB

---

## TAB

When used with the PRINT statement, the TAB function moves the cursor or print mechanism to a specified column.

When used outside the PRINT statement, the TAB function creates a string containing the specified number of spaces.

---

### Format

*str-var* =**TAB**(*int-exp*)

---

### Syntax Rules

1. When used with the PRINT statement, *int-exp* specifies the column number of the cursor or print mechanism.
2. When used outside the PRINT statement, *int-exp* specifies the number of spaces in the returned string.

---

### Remarks

1. You cannot tab beyond the current MARGIN restriction.
2. The leftmost column position is zero.
3. If *int-exp* is less than the current cursor position, the TAB function has no effect.
4. The TAB function can move the cursor or print mechanism only from the left to the right.
5. You can use more than one TAB function in the same PRINT statement.
6. Use semicolons to separate multiple TAB functions in a single statement. If you use commas, VAX BASIC moves to the next print zone before executing the TAB function.

7. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to LONG integer.

---

**Example**

```
PRINT "Number 1"; TAB(15); "Number 2"; TAB(30); "Number 3"
```

**Output**

```
Number 1      Number 2      Number 3
```

# TAN

---

## TAN

The TAN function returns the tangent of an angle in radians or degrees.

---

### Format

*real-var* =TAN(*real-exp*)

---

### Syntax Rules

*Real-exp* is an angle specified in radians or degrees, depending on which angle clause you choose with the OPTION statement.

---

### Remarks

VAX BASIC expects the argument of the TAN function to be a real expression. When the argument is a real expression, VAX BASIC returns a value of the same floating-point size. When the argument is not a real expression, VAX BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

---

### Example

```
OPTION ANGLE = DEGREES
DECLARE REAL tangent
tangent = TAN(45.0)
PRINT tangent
```

#### Output

1

---

## TIME

The **TIME** function returns the time of day (in seconds) as a floating-point number. The **TIME** function can also return process CPU time and connect time.

---

### Format

*real-var* =**TIME**(*int-exp*)

---

### Syntax Rules

None.

---

### Remarks

1. The value returned by the **TIME** function depends on the value of *int-exp*.
2. If *int-exp* equals zero, **TIME** returns the number of seconds since midnight.
3. **VAX BASIC** also accepts values 1 and 2 and returns values as shown in Table 4-7. All other arguments to the **TIME** function are undefined and cause **VAX BASIC** to signal "Not implemented" (ERR=250).
4. The **TIME** function returns a **SINGLE** floating-point value.
5. If you specify a floating-point expression for *int-exp*, **VAX BASIC** truncates it to a **LONG** integer.

# TIME

**Table 4-7: TIME Function Values**

---

<b>Argument Value:</b>	<b>VAX BASIC Returns:</b>
0	The amount of time elapsed since midnight in seconds
1	The CPU time of the current process in tenths of a second
2	The connect time of the current process in minutes

---

---

## Example

```
PRINT TIME(0)
```

### **Output**

```
49671
```



---

## TIME\$

The TIME\$ function returns a string displaying the time of day in the form *hh:mm* AM or *hh:mm* PM.

---

### Format

*str-var* =TIME\$(*int-exp*)

---

### Syntax Rules

*Int-exp* specifies the number of minutes before midnight.

---

### Remarks

1. If *int-exp* equals zero, TIME\$ returns the current time of day.
2. The value of *int-exp* must be between 0 and 1440 or VAX BASIC signals an error.
3. The TIME\$ function uses a 12-hour, AM/PM clock. Before 12:00 noon, TIME\$ returns *hh:mm* AM, and after 12:00 noon, *hh:mm* PM.
4. If you specify a floating-point expression for *int-exp*, VAX BASIC truncates it to a LONG integer.

# TIME\$

---

## Example

```
DECLARE STRING current_time  
current_time = TIME$(0)  
PRINT current_time
```

### Output

```
01:51 PM
```

---

## TRM\$

The TRM\$ function removes all trailing blanks and tabs from a specified string.

---

### Format

*str-var* =TRM\$(*str-exp*)

---

### Syntax Rules

None.

---

### Remarks

The returned *str-var* is identical to *str-exp*, except that it has all the trailing blanks and tabs removed.

---

### Example

```
DECLARE STRING old_string, new_string
old_string = "ABCDEFGG      "
new_string = TRM$(old_string)
PRINT old_string;"XYZ"
PRINT new_string;"XYZ"
```

#### Output

```
ABCDEFGG      XYZ
ABCDEFGXYZ
```

# UBOUND

---

## UBOUND

The **UBOUND** function returns the upper bounds of a compile-time or run-time dimensioned array.

---

### Format

*num-var* = **UBOUND** (*array-name* [ , *num-exp* ])

---

### Syntax Rules

1. *Array-name* must specify an array that has been previously explicitly or implicitly declared.
2. *Num-exp* specifies the number of the dimension for which you have requested the upper bound.

---

### Remarks

1. If you do not specify a numeric expression, VAX BASIC automatically returns the upper bound of the first dimension.
2. If you specify a numeric expression that is less than or equal to zero, VAX BASIC signals an error message.
3. If you specify a numeric expression that exceeds the number of dimensions, VAX BASIC signals an error message.

---

## Example

```
DECLARE INTEGER CONSTANT B = 5
DIM A(B)
account_num = 1
FOR dim_num = 0 TO UBOUND(A)
    A(dim_num) = account_num
    account_num = account_num + 1
    PRINT A(dim_num)
NEXT dim_num
```

### Output

```
1
2
3
4
5
6
```

# UNLESS

---

## UNLESS

The **UNLESS** qualifier modifies a statement. VAX BASIC executes the modified statement only if a conditional expression is false.

---

### Format

*statement* **UNLESS** *cond-exp*

---

### Syntax Rules

None.

---

### Remarks

1. The **UNLESS** statement cannot be used on nonexecutable statements or on statements such as **SELECT**, **IF**, and **DEF** that establish a statement block.
2. VAX BASIC executes the statement only if *cond-exp* is false (value zero).

---

### Example

```
PRINT "A DOES NOT EQUAL 3" UNLESS A% = 3%
```

---

## UNLOCK

The UNLOCK statement unlocks the current record or bucket locked by the last FIND or GET statement.

---

### Format

**UNLOCK** *#chnl-exp*

---

### Syntax Rules

*Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).

---

### Remarks

1. A file must be opened on the specified channel before UNLOCK can execute.
2. The UNLOCK statement only applies to files on disk.
3. If the current record is not locked by a previous GET or FIND statement, the UNLOCK statement has no effect and VAX BASIC does not signal an error.
4. The UNLOCK statement does not affect record buffers.
5. After VAX BASIC executes the UNLOCK statement, you cannot update or delete the current record.
6. Once the UNLOCK statement executes, the position of the current record pointer is undefined.

# UNLOCK

---

## Example

UNLOCK #10%



---

## UNTIL

The UNTIL statement marks the beginning of an UNTIL loop or modifies the execution of another statement.

---

### Format

#### Conditional

```
UNTIL cond-exp  
    [statement]...
```

```
    .  
    .  
    .
```

#### NEXT

#### Statement Modifier

```
statement UNTIL cond-exp
```

---

## Syntax Rules

None.

# UNTIL

---

## Remarks

### 1. Conditional

- A NEXT statement must end the UNTIL loop.
- VAX BASIC evaluates *cond-exp* before each loop iteration. If the expression is false (value zero), VAX BASIC executes the loop. If the expression is true (value nonzero), control passes to the first executable statement after the NEXT statement.

### 2. Statement Modifier

VAX BASIC executes the statement repeatedly until *cond-exp* is true.

---

## Examples

### Example 1

```
!Conditional
UNTIL A >= 5
    A = A + .01
    TOTAL = TOTAL + 1
NEXT
```

### Example 2

```
!Statement Modifier
A = A + 1 UNTIL A >= 200
```

---

## UPDATE

The UPDATE statement replaces a record in a file with a record in the record buffer. The UPDATE statement is valid on sequential, relative, and indexed files.

---

### Format

**UPDATE** *#chnl-exp* [ , **COUNT** *int-exp* ]

---

### Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Int-exp* specifies the size of the new record.

---

### Remarks

1. The file associated with *chnl-exp* must be a disk file opened with ACCESS MODIFY.
2. Each UPDATE statement must be preceded by a successful GET or FIND operation or VAX BASIC signals "No current record" (ERR=131). FIND locates but does not retrieve records. Therefore, you must specify a COUNT clause when retrieving variable-length records when the preceding operation was a FIND. *Int-exp* must exactly match the size of the old record.
3. If you are updating a variable-length record, and the record that you want to write out is not the same size as the record you retrieved, you must use a COUNT clause.

# UPDATE

4. After an UPDATE statement executes, there is no current record pointer. The next record pointer is unchanged.
5. The length of the new record must be the same as that of the existing record for all files with fixed-length records and for all sequential files. If you specify a COUNT clause, the *int-exp* must match the size of the existing record.
6. For relative files with variable-length records, the new record can be larger or smaller than the record it replaces.
  - The new record must be smaller than or equal to the maximum record size set with the MAP or RECORDSIZE clause when the file was opened.
  - You must use the COUNT clause to specify the size of the new record if it is different from that of the record last accessed by a GET operation on that channel.
7. For indexed files with variable-length records, the new record can be larger or smaller than the record it replaces. When the program does not permit duplicate primary keys, the new record can be no longer than the size specified by the MAP or RECORDSIZE clause when the file was opened. The record must include at least the primary key field.
8. An indexed file alternate key for the new record can differ from that of the existing record only if the OPEN statement for that file specified CHANGES for the alternate key.

---

## Example

```
UPDATE #4%, COUNT 32
```

---

## VAL

The VAL function converts a numeric string to a floating-point value.

### NOTE

DIGITAL recommends that you use the DECIMAL, REAL, and INTEGER functions to convert numeric strings to numeric data types.

---

### Format

*real-var* = VAL (*str-exp*)

---

### Syntax Rules

*Str-exp* can contain the ASCII digits 0 through 9, uppercase E, a plus sign (+), a minus sign (–), and a period (.).

---

### Remarks

1. The VAL function ignores spaces and tabs.
2. If *str-exp* is null, or contains only spaces and tabs, VAL returns a value of zero.
3. The value returned by the VAL function is of the default floating-point size.

# VAL

---

## Example

```
DECLARE REAL real_num  
real_num = VAL("990.32")  
PRINT real_num
```

### Output

990.32

---

## VAL%

The VAL% function converts a numeric string to an integer.

### NOTE

DIGITAL recommends that you use the DECIMAL, REAL, and INTEGER functions to convert numeric strings to numeric data types.

---

### Format

*int-var* = VAL% (*str-exp*)

---

### Syntax Rules

*Str-exp* can contain the ASCII digits 0 through 9, a plus sign (+), or a minus sign (-).

---

### Remarks

1. The VAL% function ignores spaces and tabs.
2. If *str-exp* is null or contains only spaces and tabs, VAL% returns a value of zero.
3. The value returned by the VAL% function is an integer of the default size.

## VAL%

---

### Example

```
DECLARE INTEGER ret_int
ret_int = VAL%( "789" )
PRINT ret_int
```

#### Output

789



---

## VMSSTATUS

VMSSTATUS returns the underlying VMS condition code when control is transferred to a VAX BASIC error handler.

---

### Format

*int-var* = VMSSTATUS

---

### Syntax Rules

None.

---

### Remarks

1. If ERR contains the value 194, you can specify VMSSTATUS to examine the actual error that was signaled to VAX BASIC.
2. If an error is raised by an underlying system component such as the Run-Time Library, you can specify VMSSTATUS to determine the underlying error.
3. If you are writing a utility routine that may be called from languages other than VAX BASIC, you can specify VMSSTATUS in a call to LIB\$SIGNAL to signal the underlying error to the caller of the utility routine.
4. When there is no error pending, VMSSTATUS remains undefined.

# VMSSTATUS

---

## Example

```
PROGRAM
WHEN ERROR USE global_handler
.
.
.
END WHEN
.
.
.
HANDLER global_handler
final_status% = VMSSTATUS
END HANDLER
END PROGRAM final_status%
```

---

**WAIT**

The WAIT statement specifies the number of seconds the program waits for terminal input before signaling an error.

---

**Format**

**WAIT** *int-exp*

---

**Syntax Rules**

*Int-exp* must be between 0 and 255; if it is greater than 255, VAX BASIC assumes a value of 255.

---

**Remarks**

1. The WAIT statement must precede a GET operation to a terminal or an INPUT, INPUT LINE, LINPUT, MAT INPUT, or MAT LINPUT statement. Otherwise, it has no effect.
2. *Int-exp* is the number of seconds VAX BASIC waits for input before signaling the error, "Keyboard wait exhausted" (ERR=15).
3. After VAX BASIC executes a WAIT statement, all input statements wait the specified amount of time before VAX BASIC signals an error.
4. WAIT 0 disables the WAIT statement.

# WAIT

---

## Example

```
10 DECLARE STRING your_name
   WAIT 60
   INPUT "You have sixty seconds to type your name";your_name
   WAIT 0
```

### Output

```
You have sixty seconds to type your name?
%BAS-F-KEYWAIEXH, Keyboard wait exhausted
-BAS-I-ON_CHAFIL, on channel 0 for file SYS$INPUT.;; at user PC 00000644
-RMS-W-TMO, timeout period expired
-BAS-I-FROLINMOD, from line 10 in module WAIT
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name      routine name      line      rel PC      abs PC
                                00007334 00007334
----- above condition handler called with exception 001A807C:
%BAS-F-KEYWAIEXH, keyboard wait exhausted
-BAS-I-ON_CHAFIL, on channel 0 for file SYS$INPUT.;; at user PC 00000644
-RMS-W-TMO, timeout period expired
----- end of exception message
                                00011618 00011618
                                0000F02F 0000F02F
                                0000E3F6 0000E3F6
                                0001387A 0001387A
WAIT$MAIN        WAIT$MAIN        3         00000044 00000644
```

---

## WHEN ERROR

The **WHEN ERROR** statement marks the beginning of a **WHEN ERROR** construct. The **WHEN ERROR** construct contains a protected region and can include an attached handler or identify a detached handler.

---

### Format

#### With an Attached Handler

```
WHEN ERROR IN
    protected-statement
    [ protected-statement,... ]
    .
    .
    .
USE
    handler-statement
    [ handler-statement,... ]
    .
    .
    .
END WHEN
```

#### With a Detached Handler

```
WHEN ERROR USE handler-name

    protected-statement
    [ protected-statement,... ]
    .
    .
    .
END WHEN
```

# WHEN ERROR

**HANDLER** *handler-name*

*handler-statement...*

·  
·  
·

**END HANDLER**

---

## Syntax Rules

1. *Protected-statement* specifies a statement that appears within a protected region. A protected region is a special block of code that is monitored by VAX BASIC for the occurrence of a run-time error.
2. *Handler-statement* specifies the statement that appears inside an error handler.
3. **With an Attached Handler**
  - The keyword **USE** marks the start of handler statements.
  - An attached handler must be delimited by a **USE** and **END WHEN** statement.
4. **With a Detached Handler**
  - The keyword **USE** names the associated handler for the protected region.
  - *Handler-name* must be a valid VAX BASIC identifier and cannot be the same as any label, **DEF**, **DEF\***, **SUB**, **FUNCTION** or **PICTURE** name within the same program unit.
  - A detached handler must be delimited by a **HANDLER** and **END HANDLER** statement.
  - You can specify the same detached handler with more than one **WHEN ERROR USE** statement.

---

### Remarks

1. The **WHEN ERROR** statement designates the start of a block of protected statements.
2. If an error occurs inside a protected region, **VAX BASIC** transfers control to the error handler associated with the **WHEN ERROR** statement.
3. **VAX BASIC** does not allow you branch into a **WHEN** block.
4. When **VAX BASIC** encounters an **END WHEN** statement for an attached handler or an **END HANDLER** statement for a detached handler, **VAX BASIC** clears the exception and transfers control to the following statement.
5. **VAX BASIC** allows you to nest **WHEN** blocks. If an exception occurs within a nested protected region, **VAX BASIC** transfers control to the handler associated with the innermost protected region in which the error occurred.
6. **WHEN** blocks cannot exist inside a handler.
7. **WHEN** blocks cannot cross other block structures.
8. You cannot specify a **RESUME** statement within a **WHEN ERROR** construct.
9. You cannot specify an **ON ERROR** statement within a protected region.
10. An attached handler must immediately follow the protected region of a **WHEN ERROR IN** block.
11. Exit from a handler must occur through a **RETRY**, **CONTINUE**, or **EXIT HANDLER** statement, or by reaching the end of the handler delimited by **END WHEN** or **END HANDLER**.
12. For more information about detached handlers, see the **HANDLER** statement.

# WHEN ERROR

---

## Examples

### Example 1

```
!With an attached handler
PROGRAM salary
DECLARE REAL hourly_rate, no_of_hours, weekly_pay
WHEN ERROR IN
    INPUT "Enter your hourly rate";hourly_rate
    INPUT "Enter the number of hours you worked this week";no_of_hours
    weekly_pay = no_of_hours * hourly_rate
    PRINT "Your pay for this week is";weekly_pay

USE
    SELECT ERR
        CASE = 50
            PRINT "Invalid data"
            RETRY
        CASE ELSE
            EXIT HANDLER
    END SELECT
END WHEN
END PROGRAM
```

### Output

```
Enter your hourly rate? 35.00
Enter the number of hours you worked this week? 45
Your pay for this week is 1575
```



# WHEN ERROR

## Example 2

```
!With a detached handler
PROGRAM salary
DECLARE REAL hourly_rate, no_of_hours, weekly_pay
WHEN ERROR USE patch_work
    INPUT "Enter your hourly rate";hourly_rate
    INPUT "Enter the number of hours you worked this week";no_of_hours
    weekly_pay = no_of_hours * hourly_rate
    PRINT "Your pay for this week is";weekly_pay
END WHEN

HANDLER patch_work
    SELECT ERR
        CASE = 50
            PRINT "Invalid data"
            RETRY
        CASE ELSE
            EXIT HANDLER
    END SELECT
END HANDLER
END PROGRAM
```

## Output

```
Enter your hourly rate? Nineteen dollars and fifty cents
Invalid data
Enter your hourly rate? 19.50
Enter the number of hours you worked this week? 40
Your pay for this week is 780
```

# WHILE

---

## WHILE

The WHILE statement marks the beginning of a WHILE loop or modifies the execution of another statement.

---

### Format

#### Conditional

```
WHILE cond-exp  
    [statement]...
```

```
    .  
    .  
    .
```

**NEXT**

#### Statement Modifier

```
statement WHILE cond-exp
```

---

## Syntax Rules

A NEXT statement must end the WHILE loop.

---

## Remarks

### 1. Conditional

VAX BASIC evaluates *cond-exp* before each loop iteration. If the expression is true (value nonzero), VAX BASIC executes the loop. If the expression is false (value zero), control passes to the first executable statement after the NEXT statement.

## 2. Statement Modifier

VAX BASIC executes the statement repeatedly as long as *cond-exp* is true.

---

## Examples

### Example 1

```
!Conditional
WHILE X < 100
    X = X + SQR(X)
NEXT
```

### Example 2

```
!Statement Modifier
X% = X% + 1% WHILE X% < 100%
```

# XLATE\$

---

## XLATE\$

The XLATE\$ function translates one string to another by referencing a table string you supply.

---

### Format

*str-var* = XLATE[\$] (*str-exp1*, *str-exp2*)

---

### Syntax Rules

1. *Str-exp1* is the input string.
2. *Str-exp2* is the table string.

---

### Remarks

1. *Str-exp2* can contain up to 256 ASCII characters, numbered from 0 to 255; the position of each character in the string corresponds to an ASCII value. Because 0 is a valid ASCII value (null), the first position in the table string is position zero.
2. XLATE\$ scans *str-exp1* character by character, from left to right. It finds the ASCII value *n* of the first character in *str-exp1* and extracts the character it finds at position *n* in *str-exp2*. XLATE\$ then appends the character from *str-exp2* to *str-var*. XLATE\$ continues this process, character by character, until the end of *str-exp1* is reached.

3. The output string may be smaller than the input string for the following reasons:
  - XLATE\$ does not translate nulls. If the character at position *n* in *str-exp2* is a null, XLATE\$ does not append that character to *str-var*.
  - If the ASCII value of the input character is outside the range of positions in *str-exp2*, XLATE\$ does not append any character to *str-var*.

---

## Example

```
DECLARE STRING A, table, source
A = "abcdefghijklmnopqrstuvwxyz"
table = STRING$(65, 0) + A
INPUT " Type a string of uppercase letters"; source
PRINT XLATE$(source, table)
```

### Output

```
Type a string of uppercase letters? ABCDEFG
abcdefg
```



# Transporting Programs Between VAX BASIC and BASIC-PLUS-2

---

This appendix summarizes transportability issues between BASIC-PLUS-2 and VAX BASIC.

---

## A.1 Overview

This appendix is for users who want to write BASIC programs that can be used in both VAX BASIC and PDP-11 BASIC-PLUS-2. It describes functionality that is particular to one language, as well as statements and functions that appear the same in both languages, but produce different results.

Note that this appendix does not describe all the differences between VAX BASIC and BASIC-PLUS-2. To assist you in writing transportable programs, use the DCL command `BASIC/FLAG=BP2COMPATIBILITY` when you invoke VAX BASIC. This command causes VAX BASIC to signal an informational message whenever you use functionality that is not compatible with BASIC-PLUS-2.

This appendix contains the following sections:

- Language-specific functionality
- I/O differences
- Procedure calling
- Generated errors
- Miscellaneous differences

---

## A.2 Language-Specific Functionality

The following statements, functions, and clauses are available only in BASIC-PLUS-2:

- Specifying a line number with the CHAIN statement †
- The FSS\$ function
- The ONECHR function (see the VAX BASIC INKEY\$ function)
- The PEEK function †
- The SPEC% function †
- The CLUSTERSIZE and MODE clauses on the OPEN statement †

VAX BASIC supports the following functionality which is not available in BASIC-PLUS-2:

- Graphics
- Support for the VAX Language-Sensitive Editor (LSE)
- Support for the Program Design Facility (PDF)
- Support for the VAX Source Code Analyzer (SCA)
- Non-zero lower bounds for arrays
- WHEN blocks
- Return values on the END, EXIT DEF, END DEF, EXIT FUNCTION, and END FUNCTION statements
- Dollar sign (\$) and percent sign (%) suffixes in explicitly declared variables
- User-defined data types with the RECORD statement
- DECIMAL, GFLOAT, and HFLOAT data types
- Hexadecimal, binary, and octal literal notation

The following statements, functions, and commands are available only in VAX BASIC:

- The %DECLARED directive
- The %INCLUDE %FROM %CDD directive
- The %INCLUDE %FROM %LIBRARY directive
- The %PRINT directive
- The %REPORT %DEPENDENCY directive

---

† Specific to BASIC-PLUS-2 on RSTS/E systems.



- All graphics statements
- The CAUSE ERROR statement
- The FREE statement
- The FIND statement with the ALLOW, REGARDLESS, and WAIT clauses
- The GET statement with the ALLOW, REGARDLESS, and WAIT clauses
- The MID\$ assignment statement
- The OPTION CONSTANT TYPE, OPTION HANDLE, and OPTION ANGLE statements
- The PROGRAM, END PROGRAM and EXIT PROGRAM statements
- The RECORD statement
- All WHEN block statements and clauses (WHEN...END WHEN, HANDLER...END HANDLER, CONTINUE, RETRY, and EXIT HANDLER)
- The DECIMAL function
- The INKEY\$ function
- The LBOUND and UBOUND functions
- The LOC function
- The MARGIN and NOMARGIN functions
- The MAX, MIN, and MOD functions
- The VMSSTATUS and RMSSTATUS functions
- The ANY and OPTIONAL keywords in EXTERNAL routine declarations

---

## A.3 I/O Differences

This section discusses some I/O differences between VAX BASIC and BASIC-PLUS-2.

---

### A.3.1 The MAGTAPE Function

VAX BASIC does not support the MAGTAPE function except for the rewind tape function (code 3). Table A-1 describes the VAX BASIC actions you can perform to obtain other MAGTAPE functionality.

**Table A-1: MAGTAPE Functionality in VAX BASIC**

<b>Code</b>	<b>Function</b>	<b>VAX BASIC Action</b>
2	Write EOF	Close channel with the CLOSE statement
3	Rewind tape	Use the RESTORE # statement, the REWIND clause on an OPEN statement, or the MAGTAPE function
4	Skip records	Perform GET operations, ignore data until reaching desired record
5	Backspace	Rewind tape, perform GET operations, ignore data until reaching desired record
6	Set density or set parity	Use the DCL commands MOUNT/DENSITY and MOUNT/FOREIGN or the \$MOUNT system service
7	Get status	Use the RMSSTATUS function

### **A.3.2 The OPEN Statement**

The following differences exist in the OPEN statement when used in VAX BASIC and BASIC-PLUS-2:

- In VAX BASIC, a map named in an OPEN statement is never initialized; in BASIC-PLUS-2, variables in the map are initialized to zero or to the null string.
- In VAX BASIC, an OPEN error causes the STATUS variable to be set to zero. In BASIC-PLUS-2, an OPEN error causes the STATUS variable to be set to the RMS STS field value. Use the VAX BASIC RMSSTATUS function to return the RMS STS field value.
- Both VAX BASIC and BASIC-PLUS-2 allow you to omit key clauses when opening an existing indexed file. However, VAX BASIC requires that you explicitly specify FOR INPUT; BASIC-PLUS-2 does not.

The following OPEN statement clauses produce different results when used in VAX BASIC and BASIC-PLUS-2:

- **The ALLOW Clause**

VAX BASIC requires that you have write access to a file in order to specify ALLOW NONE in an OPEN statement; BASIC-PLUS-2 does not.

- **The CLUSTERSIZE Clause**

The CLUSTERSIZE clause can be used with an OPEN statement on RSTS/E systems only. In VAX BASIC and BASIC-PLUS-2 on RSX systems, you can obtain functionality similar to the CLUSTERSIZE clause by using the EXTENDSIZE and WINDOWSIZE clauses. The EXTENDSIZE clause specifies the amount of space a file is extended after the existing space is full. The WINDOWSIZE clause specifies the number of block retrieval pointers that are kept in memory for the file.

- **The CONTIGUOUS Clause**

In VAX BASIC, using the CONTIGUOUS clause with an OPEN statement does not necessarily mean the file will occupy contiguous disk space. If there is not enough contiguous space available, RMS allocates the largest possible contiguous space and does not signal an error. In BASIC-PLUS-2, if there is not enough contiguous space available, RMS signals an error.

- **The MODE Clause**

The MODE clause cannot be used with an OPEN statement in VAX BASIC. In BASIC-PLUS-2 on RSX systems, the MODE clause is ignored except when used for device-specific I/O. Table A-2 lists transportable BASIC statements that correspond to RSTS/E disk MODE values. You can use these statements in VAX BASIC as well as in BASIC-PLUS-2 on RSX systems.

**Table A-2: RSTS/E Disk MODE Values and Corresponding BASIC Statements**

<b>MODE Value</b>	<b>Function</b>	<b>Transportable Statement</b>
0%	Normal read/write	Use OPEN with ACCESS MODIFY and ALLOW MODIFY
1%	Update file	Use OPEN with ACCESS MODIFY
2%	Append to file	Use OPEN with ACCESS APPEND
5%	Update file	Use OPEN with ACCESS MODIFY and ALLOW MODIFY
16%	Create contiguous	Use OPEN with CONTIGUOUS
32%	Create tentative file	Use OPEN with TEMPORARY

(continued on next page)

**Table A-2 (Cont.): RSTS/E Disk MODE Values and Corresponding BASIC Statements**

---

<b>MODE Value</b>	<b>Function</b>	<b>Transportable Statement</b>
64%	Create contiguous file conditionally	Use OPEN with CONTIGUOUS
4096%	Read only regardless	Use GET or FIND with REGARDLESS <sup>1</sup>
8192%	Read Only	Use OPEN with ACCESS READ

---

<sup>1</sup>The REGARDLESS clause is not available in BASIC-PLUS-2.

---

- **The RECORDSIZE Clause**  
In all VAX BASIC files and BASIC-PLUS-2 RMS files, the RECORDSIZE clause specifies the RMS logical record size. In BASIC-PLUS-2 virtual files, the RECORDSIZE clause specifies the size of the I/O buffer for the channel.

---

### **A.3.3 The PUT Statement**

In VAX BASIC, a PUT statement with a count of zero to a variable length file creates a record with a length of zero; in BASIC-PLUS-2, the size of the record created is the size specified in the RECORDSIZE clause or, if no RECORDSIZE clause is specified, the length of the longest map.

---

## **A.4 Procedure Calling**

This section discusses some differences in procedure calling between VAX BASIC and BASIC-PLUS-2.

---

### **A.4.1 The CALL Statement**

The following difference occur in the CALL statement when used in VAX BASIC and BASIC-PLUS-2:

- In VAX BASIC, you can use the CALL statement to call a procedure that is written in any language which supports the VAX Procedure Calling Standard. In BASIC-PLUS-2, you can use the CALL statement to call only BASIC-PLUS-2 and MACRO subprograms.

- In VAX BASIC, individual array element parameters (except virtual array elements) are passed by reference and are modifiable. In BASIC-PLUS-2, individual array element parameters are passed by local copy and are not modifiable.
- VAX BASIC does not allow passing of entire virtual arrays; BASIC-PLUS-2 allows passing of entire virtual arrays.
- In VAX BASIC, you can pass up to 255 parameters in each subprogram. In BASIC-PLUS-2, you can pass up to 8 parameters in each subprogram.

---

## A.4.2 The CHAIN Statement

In VAX BASIC and in BASIC-PLUS-2 on RSX systems, you cannot specify a line number in a CHAIN statement. In BASIC-PLUS-2 on RSTS/E systems, line numbers are allowed.

---

## A.4.3 SYS and FIP SYS Calls

VAX BASIC supports a subset of RSTS/E SYS and FIP calls. These are summarized in Tables A-3 and A-4.

**Table A-3: VAX BASIC Subset of RSTS/E SYS Calls**

<b>Function Code</b>	<b>Function</b>
0	Cancel CTRL/O effect on terminal
2	Enable echoing on terminal
3	Disable echoing on terminal
5	Exit with no prompt message
6	SYS call to the file processor
7	Get core common string
8	Put core common string
9	Exit and clear program
11	Cancel all type ahead

**Table A-4: VAX BASIC Subset of RSTS/E FIP SYS Calls**

---

<b>Function Code</b>	<b>Function</b>
-23	Terminate file name string scan
-13	Change priority/run burst/job size
-10	Begin file name string scan
-7	Enable CTRL/C trap
9	Return error messages
10	Assign user logical
11	Deallocate a device or deassign user logical
12	Deallocate all devices
18	Obsolete (use function code 22)
22	Message send/receive

---

Note that the FIP call for sending or receiving messages (code 22) produces different results when used in VAX BASIC and BASIC-PLUS-2 on RSTS/E systems. In VAX BASIC, a receiver identification is removed from the receiver table when the image that declared it exits; in BASIC-PLUS-2 on RSTS/E systems, a receiver identification stays in the receiver table until it is explicitly removed or the job terminates. This difference causes incompatible behavior with chained programs, because in VAX BASIC the receiver identification is valid only for the program that declares it; whereas in BASIC-PLUS-2 on RSTS/E systems, the chained programs can share a receiver identification. DIGITAL recommends the use of mailboxes or DECnet task-to-task communication as a replacement for these FIP calls in VAX BASIC.

---

## A.5 Generated Errors

In VAX BASIC and BASIC-PLUS-2, the same errors are handled differently or signal different error messages and numbers:

- If you press CTRL/Z after responding to an INPUT statement, VAX BASIC makes the assignment and signals the error “End of file on device” (ERR=11) when the next INPUT statement executes; BASIC-PLUS-2 signals the same error, does not make the assignment, and terminates the current input line.

- When nonnumeric or floating-point data is input to an INPUT or READ statement expecting numeric or integer data, VAX BASIC signals the error “Data format error” (ERR=50); BASIC-PLUS-2 signals the error “Illegal number” (ERR=52).
- When the integer index of a FOR loop exceeds the default integer size, VAX BASIC signals the error “Integer error or overflow” (ERR=51); BASIC-PLUS-2 signals the error “Integer overflow, FOR loop” (ERR=60).
- When a program opens channel 0, VAX BASIC signals the error “Illegal I/O channel” (ERR=46); BASIC-PLUS-2 signals the error “I/O channel already open at line <line number>” (ERR=7).
- If no error handler is active, the errors in Table A-5 are fatal errors in VAX BASIC, but warnings in BASIC-PLUS-2.

**Table A-5: Fatal Errors in VAX BASIC that are Warnings in BASIC-PLUS-2**

<b>Error Number</b>	<b>Message</b>
48	Floating point error
51	Integer error
52	Illegal number
53	Illegal argument in LOG
54	Imaginary square roots
61	Division by zero

- When a user has WRITE access to a block in a virtual file and a second user attempts to access that block, VAX BASIC and BASIC-PLUS-2 on RSX signal the error “Record/bucket locked” (ERR=154); BASIC-PLUS-2 on RSTS/E signals the error “Disk block is interlocked” (ERR=19).

---

## A.6 Miscellaneous Differences

This section discusses some miscellaneous differences between VAX BASIC and BASIC-PLUS-2.

---

## A.6.1 Data Types

VAX BASIC supports DECIMAL, HFLOAT, GFLOAT and user-defined data types; BASIC-PLUS-2 does not. In BASIC-PLUS-2, as an alternative to the DECIMAL data type, use the DOUBLE data type with the OPTION SCALE statement or the SCALE command to minimize floating-point inaccuracy.

---

## A.6.2 The DEF and DEF\* Statements

In VAX BASIC, you can specify a maximum of 255 parameters in a DEF statement and a maximum of 8 parameters in a DEF\* statement; in BASIC-PLUS-2, you can specify a maximum of 8 parameters in both the DEF and DEF\* statements.

---

## A.6.3 Default Integer Size

If you do not specify a default integer size when invoking VAX BASIC, the default integer size is LONGWORD; in BASIC-PLUS-2, the default integer size is WORD. In VAX BASIC, use the OPTION SIZE=INTEGER WORD statement or the COMPILE/WORD command to make BASIC-PLUS-2 programs transportable to VAX BASIC.

---

## A.6.4 Integer Overflow

When performing integer arithmetic, VAX BASIC signals an error message if the size of the value returned exceeds the default integer size. BASIC-PLUS-2 does not signal an error message, truncates the result, and returns a signed integer quantity.

To disable error checking for integer overflow in VAX BASIC, use the OPTION INACTIVE=INTEGER OVERFLOW statement or the DCL command BASIC/CHECK=NOOVERFLOW.

---

## A.6.5 Line Numbers and Labels

In VAX BASIC, line numbers are optional. In BASIC-PLUS-2, you need at least one line number in each program.

In a VAX BASIC program, labels, compiler directives, and comments can begin in column zero. In BASIC-PLUS-2, they cannot.



---

## A.6.6 The MAP and COMMON Statements

The MAP and COMMON statements have different functionality when used in VAX BASIC and BASIC-PLUS-2. The differences are as follows:

- In VAX BASIC, both COMMON areas and maps can have names containing a maximum of 31 characters and include underscore (\_) characters. In BASIC-PLUS-2, COMMON areas and maps can have names containing a maximum of six characters; underscore characters are not allowed.
- If two strings overlap in a map, VAX BASIC performs string assignments as if no overlap exists; BASIC-PLUS-2 performs string assignments one character at a time from left to right. (Note that RSET assignment is performed from right to left.) For example:

```
10      MAP (FOO) A$ = 5
        MAP (FOO) FILL$ = 2, B$ = 5
        A$ = 'ABCDE'
        PRINT A$
        B$ = A$
        PRINT B$, A$
```

When you run this program in VAX BASIC, the following output is displayed:

```
ABCDE
ABCDE          ABABC
```

When you run this program in BASIC-PLUS-2, the following output is displayed:

```
ABCDE
ABABA          ABABA
```

---

## A.6.7 The MAP DYNAMIC Statement

The following differences occur in the MAP DYNAMIC statement when used in VAX BASIC and BASIC-PLUS-2:

- If the MAP DYNAMIC statement is in an external subprogram or function, BASIC-PLUS-2 resets all pointers to the first byte each time the external module is called. VAX BASIC initializes the map area once when the external module is first called, and does not reset the variables pointers on subsequent calls.

- VAX BASIC allows you to specify a PSECT name or a static string variable in a MAP DYNAMIC statement; BASIC-PLUS-2 allows you to specify only a PSECT name.

---

## A.6.8 The PRINT Statement

In VAX BASIC and RSX BASIC-PLUS-2, when you print to a terminal-format file and the line is to exceed 72 characters, you must either specify a record size, or a map, or use the VAX BASIC MARGIN function; this is not the case in BASIC-PLUS-2 on RSTS/E.

---

## A.6.9 The PRINT USING Statement

In VAX BASIC, the PRINT USING string formatting characters (L,R,C, and E) can be either uppercase or lowercase. In BASIC-PLUS-2, string formatting characters must be uppercase; otherwise, they are treated as string literals.

---

## A.6.10 The REPLACE Command

In VAX BASIC, the REPLACE command writes the source program to the device and directory you specify in the OLD command. In BASIC-PLUS-2, the REPLACE command writes the source program to the current directory.

---

## A.6.11 The SPEC% and PEEK Functions

You can use the SPEC% and PEEK functions only on RSTS/E systems. These functions are not transportable.

---

## A.6.12 String Comparisons

When making string comparisons in all relational operations, VAX BASIC pads the shorter string with a blank space (ASCII value 32). BASIC-PLUS-2 pads the shorter string with a blank space only on equals (=) and not equals (<>) relational operations.

---

### A.6.13 Assigning Symbols

In VAX BASIC, you use the logical name SYS\$CURRENCY to change the currency symbol, SYS\$RADIX\_POINT to change the radix point symbol, and SYS\$DIGIT\_SEP to change the separator symbol. In BASIC-PLUS-2, you select the symbols for currency, radix point, and separator when installing the compiler.

---

### A.6.14 The TIME Function

The numeric arguments to the TIME function have different results when used in VAX BASIC and BASIC-PLUS-2. These differences are summarized in Table A-6.

**Table A-6: VAX BASIC and BASIC-PLUS-2 TIME Function Differences**

<b>TIME Argument</b>	<b>Result</b>
0 <sup>1</sup>	In both VAX BASIC and BASIC-PLUS-2, TIME returns the number of seconds that elapsed since midnight.
1	In VAX BASIC and BASIC-PLUS-2 on RSTS/E systems, TIME returns the current job's CPU time in tenths of a second.
2	In VAX BASIC and BASIC-PLUS-2 on RSTS/E systems, TIME returns the current job's connect time in minutes.
3	In VAX BASIC, TIME returns zero. In BASIC-PLUS-2 on RSTS/E systems, TIME returns kilo-core ticks.
4	In VAX BASIC, TIME returns zero. In BASIC-PLUS-2 on RSTS/E systems, TIME returns device time in minutes.

---

<sup>1</sup>Zero is the only valid argument for TIME in BASIC-PLUS-2 on RSX systems.

---

---

### A.6.15 The TIME\$ Function

In VAX BASIC, the value returned by the TIME\$ function is always displayed in AM/PM format. In BASIC-PLUS-2, the time can be expressed in either AM/PM format or 24-hour format, depending on the option you select when installing BASIC-PLUS-2.



# ANSI Minimal BASIC

---

This appendix explains the operation of the VAX BASIC compiler when used with the /ANSI\_STANDARD qualifier.

---

## B.1 Introduction

The American National Standard for Minimal BASIC (ANSI X3.60-1978) describes a nucleus of the BASIC programming language. This nucleus will be a part of any BASIC implementation that conforms to this standard. Thus, writing programs that conform to the ANSI Minimal BASIC standard helps assure that they will run under any standard implementation of BASIC.

The ANSI Minimal BASIC Standard allows both extensions to the current standard and features whose behavior is defined by each implementation. This chapter describes these extensions and implementation-defined features. Many features of VAX BASIC are allowed as extensions to ANSI Minimal BASIC. For example, programs with 31-character variable names will compile correctly; however, VAX BASIC reports an informational message for each instance of a long variable name. This tells you that your program does not strictly conform to ANSI Minimal BASIC.

Certain features of VAX BASIC are invalid in ANSI Minimal BASIC programs. For example, variables ending in a percent sign are invalid because ANSI Minimal BASIC does not allow integer variables. If you try to use this VAX feature in ANSI Minimal BASIC, VAX BASIC signals the error "Integer data type not supported in ANSI."

For a thorough understanding of ANSI Minimal BASIC, read ANSI X3.60-1978.

Note that the descriptions and explanations in this chapter apply only to programs compiled or run with the `/ANSI_STANDARD` qualifier in effect.

---

## B.2 The `/ANSI_STANDARD` Qualifier

`/ANSI_STANDARD` is a qualifier to both the DCL command `BASIC` and to the `SET` command in the BASIC environment. When you specify this qualifier, the following qualifiers are not allowed:

- `/ANALYSIS_DATA`
- `/DESIGN`
- `/OPTIMIZE` (`/NOSETUP` in the BASIC environment)
- `/SCALE`
- `/SYNTAX_CHECK`
- `/TYPE_DEFAULT`

VAX BASIC signals an error if you use any of these qualifiers in addition to the `/ANSI_STANDARD` qualifier.

Note that you cannot use compiler directives in programs compiled with the `/ANSI_STANDARD` qualifier. In addition, you cannot use immediate mode statements while the `/ANSI_STANDARD` qualifier is in effect.

---

## B.3 Extensions To ANSI Minimal BASIC Standard X3.60-1978

The following items are extensions to the ANSI Minimal BASIC Standard. In order to write completely transportable programs, you should avoid these extensions and use only the capabilities allowed by the standard. In most cases VAX BASIC reports an informational error if you use any extensions.

---

### B.3.1 Program Format

The ANSI Minimal BASIC Standard permits comments only with the `REM` statement. With the `/ANSI_STANDARD` qualifier in effect, VAX BASIC allows comment fields beginning with an exclamation point and ending with an exclamation point or a carriage return.

VAX BASIC also allows explicit line continuation with ampersands. However, implicit line continuation is invalid.

---

### B.3.2 Statements

The ANSI Minimal BASIC Standard requires that each program have an END statement. In VAX BASIC the END statement is optional. If your program does not have an END statement, VAX BASIC reports the informational message, "ENDSTAREQ, END statement required in ANSI."

Also, the LET keyword is required in ANSI Minimal BASIC. VAX BASIC signals "LETKEYREQ, LET keyword required in ANSI" when it encounters an assignment statement without the LET keyword.

---

### B.3.3 Delimiters

The ANSI Minimal BASIC Standard specifies that all keywords must be preceded by one space and followed by at least one space, if the keyword is not at the end of a line. With the /ANSI\_STANDARD qualifier in effect, you can delimit keywords with either spaces or tabs.

---

### B.3.4 Variables

The ANSI Minimal BASIC Standard limits variable names in the following ways:

- String variables and string arrays can be named with only one alphabetic character, followed by a dollar sign. For example, valid identifiers for string variables are K\$, T\$, and Q\$(17).
- Numeric arrays can be named with only one alphabetic character, followed by the subscript reference. For example, valid identifiers for numeric arrays are M(25), K(n), and A(n/2).
- Simple numeric variables can be named with a maximum of two characters: one alphabetic character, followed by an optional digit. For example, valid identifiers for simple numeric variables are R5, K1, and T2.

With the /ANSI\_STANDARD qualifier in effect, VAX BASIC allows up to 31-character variable and array names. Names ending in a percent sign (%) are invalid, as are any explicitly declared variables.

Note that VAX BASIC initializes all numeric variables to zero and all string variables to the null string. To conform to the minimal standard, you should explicitly initialize all variables in your program.

All VAX BASIC keywords remain reserved words when the /ANSI\_STANDARD qualifier is in effect; you cannot use these reserved words as variable names. See Appendix D in this manual for a list of reserved keywords.

---

### B.3.5 Numeric Constants

The ANSI Minimal BASIC Standard allows numeric constants of the following form:

sd...d	Implicit point representation
sd...drd...d	Explicit point unscaled representation
sd...drd...dEsd...d	Explicit point scaled representation
sd...dEsd...d	implicit point scaled representation

**d**

Is a decimal digit.

**r**

Is a period.

**s**

Is an optional sign.

**E**

Is the explicit character E.

In addition to constants of this form, VAX BASIC with the /ANSI\_STANDARD qualifier in effect allows integer constants that end in a percent sign (%) and explicitly typed numeric constants.

---

### B.3.6 Data Input

VAX BASIC provides several extensions to the ANSI Minimal BASIC Standard for data input.



---

### B.3.6.1 Unquoted String Data

The ANSI Minimal BASIC Standard limits unquoted strings occurring in a DATA statement, or in response to the input prompt, to the following subset of ASCII characters:

- Uppercase letters (A–Z)
- Digits (0–9)
- The period (.)
- The plus sign (+)
- The minus sign (–)
- The space character

In VAX BASIC, an unquoted string in a DATA statement can contain any ASCII character, with the following exceptions:

- The comma (,)
- The null character
- The form feed character
- An ampersand (&), if it is the last character on the line

As input to the INPUT statement, VAX BASIC allows unquoted strings to contain any printable ASCII character (that is, all characters with an ASCII code greater than 31, except character code 127, the delete character). VAX BASIC also allows the following nonprinting characters in the INPUT statement:

- The back space character
- The horizontal tab
- The vertical tab
- The form feed character

---

### B.3.6.2 Null Input

The ANSI Minimal BASIC Standard requires that the DATA statement consist of numeric constants, string constants, or unquoted strings. VAX BASIC allows these items, and also allows null items (that is, two successive commas not within a quoted string) as input in a DATA statement. A null item in a DATA statement results in the assignment of either a null string or a numeric value of zero to the corresponding variable in the READ statement.

---

### **B.3.7 User-Defined Functions (the DEF Statement)**

The ANSI Minimal BASIC Standard requires that user-defined functions accept a single parameter. The formal parameter in the DEF function must be an unsubscripted numeric variable. With the `/ANSI_STANDARD` qualifier in effect, VAX BASIC reports a syntax error for DEF functions that specify more than one parameter.

The ANSI Minimal BASIC Standard makes no mention of DEFs with string parameters. VAX BASIC in ANSI Standard mode allows string DEFs but signals the informational error "String DEF not ANSI."

DEF functions can be recursive. However, VAX BASIC does not detect infinitely recursive DEF functions. If your program invokes an infinitely recursive DEF function, your program will eventually terminate with a fatal error (typically, an access violation).

---

### **B.3.8 Built-In Functions**

The ANSI Minimal BASIC Standard allows only the following implementation-supplied (built-in) functions:

- ABS
- ATN
- COS
- EXP
- INT
- LOG
- RND
- SGN
- SIN
- SQR
- TAN

With the `/ANSI_STANDARD` qualifier in effect, if you use any built-in functions other than these, VAX BASIC reports an informational error "Language feature not ANSI."

Further, VAX BASIC reports an error if you use any of the following functions in a program compiled with the /ANSI\_STANDARD qualifier:

- DECIMAL
- INTEGER
- REAL
- GETRFA

---

### B.3.9 Arrays

The ANSI Minimal BASIC Standard requires that array declarations are valid only for numerics, and allows only one or two dimensions. With the /ANSI\_STANDARD qualifier in effect, VAX BASIC allows multi-dimensional arrays of all data types, including strings.

All arrays have a lower bound of zero unless an OPTION BASE statement specifies a lower bound of 1. The format of OPTION BASE is as follows:

OPTION BASE *n*

***n***

Is either zero or 1. A value of zero specifies that the lower bound of arrays is either (0) or (0,0). A value of 1 specifies that the lower bound is either (1) or (1,1).

Although you can have arrays of any floating-point data type, you control this feature with qualifiers to the DCL command BASIC or the SET command. This means that all floating-point arrays in a single program are of the same data type.

The ANSI Minimal BASIC Standard requires that all variables and arrays have unique names. However, VAX BASIC allows a variable and an array to have the same name.

---

## B.4 Implementation-Defined Features

The ANSI Minimal BASIC Standard leaves the following features to be defined by the implementation. The behavior of these implementation-defined features in VAX BASIC is as described in the following sections.

---

## B.4.1 Initial Values for Variables

The ANSI Minimal BASIC Standard recommends that all variables are “detectably undefined in the sense that an exception will result from any attempt to access the value of a variable before that variable is explicitly assigned a value.” Therefore you should explicitly initialize all variables. VAX BASIC initializes all numeric variables to zero and all dynamic string variables to the null string.

---

## B.4.2 Retention of Long Strings

The ANSI Minimal BASIC Standard states that string variables must be able to contain strings of at least 18 characters. VAX BASIC lets you use strings of up to 65535 characters.

---

## B.4.3 Accuracy of Evaluation of Numeric Expressions

The ANSI Minimal BASIC Standard does not specify a minimum accuracy, but recommends at least six significant decimal digits of precision. In VAX BASIC, the accuracy of numeric expressions is always the same as that of the operands. This is specified with the `/REAL_SIZE` qualifier.

---

## B.4.4 Machine Infinitesimal

The ANSI Minimal BASIC Standard recommends that machine infinitesimal be at most  $1E-38$ . For programs compiled with `/REAL_SIZE` of `SINGLE` or `DOUBLE`, machine infinitesimal is approximately  $2.9E-39$ ; with `/REAL_SIZE=GFLOAT`, machine infinitesimal is approximately  $5.6E-308$ ; and with `/REAL_SIZE=HFLOAT`, machine infinitesimal is approximately  $8.4E-4933$ .

---

## B.4.5 Machine Infinity

The ANSI Minimal BASIC Standard recommends that machine infinity be at least  $1E38$ . For programs compiled with `/REAL_SIZE` of `SINGLE` or `DOUBLE`, machine infinity is approximately  $1.7E38$ ; with `/REAL_SIZE=GFLOAT`, machine infinity is approximately  $9.0E309$ ; and with `/REAL_SIZE=HFLOAT`, machine infinity is approximately  $8.4E4933$ .

---

## B.4.6 Precision For Numeric Values

The ANSI Minimal BASIC Standard recommends at least six significant decimal digits of precision. This corresponds to the `SINGLE` argument of the `VAX BASIC REAL_SIZE` qualifier. You can also specify `DOUBLE` or `GFLOAT` (up to 15 significant decimal digits of accuracy), or `HFLOAT` (up to 33 significant decimal digits of accuracy). See the *VAX BASIC User Manual* for more information.

Note that the accuracy of numeric expressions is always the same as the precision specified with an argument to the `REAL_SIZE` qualifier.

---

## B.4.7 Exrad-Width For Printing Numeric Representations

The ANSI Minimal BASIC Standard requires at least two positions for the representation of the exrad component of a numeric representation. In `VAX BASIC` for programs compiled with `/REAL_SIZE` of `SINGLE` or `DOUBLE`, exrad-width is two. For programs compiled with `/REAL_SIZE=GFLOAT`, exrad-width is three. For programs compiled with `/REAL_SIZE=HFLOAT`, exrad-width is four.

---

## B.4.8 Significance-Width For Printing Numeric Representations

The ANSI Minimal BASIC Standard specifies at least six positions for controlling the number of significant decimal digits printed in numeric representations. In `VAX BASIC` the `PRINT` statement provides up to six significant positions for numeric values, regardless of the floating-point data type in effect.

---

## B.4.9 Print Zone Length

`VAX BASIC` always has five 14-position print zones per print line.

---

## B.4.10 Margin for Output Line

The ANSI Minimal BASIC Standard makes no recommendation for the width of the output line. With the `/ANSI_STANDARD` qualifier in effect, the margin width for the controlling terminal is 80 characters. Note that the margin width for the controlling terminal is infinite for programs compiled with the `/NOANSI_STANDARD` qualifier.

---

## **B.4.11 Pseudorandom Number Sequence**

In VAX BASIC the RND function produces a pseudorandom sequence of numbers until the RANDOMIZE statement is executed. After RANDOMIZE executes, the RND function produces a random sequence of numbers.

---

## **B.4.12 Unique Line Numbers**

VAX BASIC follows the ANSI Minimal BASIC Standard's recommendations for local editing of statement lines. Statement lines can be entered in any order; VAX BASIC sorts the program into the proper order. If you enter two lines with the same line number, VAX BASIC keeps the second line and deletes the first. VAX BASIC deletes any line containing only a line number; it also deletes any line containing only a line number and formatting characters (such as spaces or form feeds).

---

## **B.4.13 Input Prompt**

The ANSI Minimal BASIC Standard recommends that the input prompt be a question mark followed by a single space. VAX BASIC conforms to this recommendation. Note that you cannot supply a string constant to be displayed as an input prompt. If you attempt to supply a string prompt, VAX BASIC signals "Language feature not ANSI."

---

## **B.4.14 End of Input Reply**

In VAX BASIC, the end of input reply is a carriage return.

---

## **B.4.15 End of Print Line**

In VAX BASIC, the end of print line is a carriage return/line feed combination (ASCII code 13 and 10).

---

## **B.4.16 Exponentiation Operator**

VAX BASIC accepts two asterisk characters (\*\*) as the exponentiation operator.

## Appendix C

# ASCII Character Codes

---

ASCII is a 7-bit character code with an optional parity bit (8) added for many devices. Programs normally use seven bits internally with the eighth bit being zero; the extra bit is either stripped (on input) or added by a device driver (on output) so the program will operate with either parity- or nonparity-generating devices. The eighth bit is reserved for future standardization.

The International Reference Version (IRV) of ISO Standard 646 is identical to the IRV in CCITT Recommendation V.3 (International alphabet No. 5). The character sets are the same as ASCII except that the ASCII dollar sign (hexadecimal 24) is the international currency sign, which looks like ###.

ISO Standard 646 and CCITT V.3 also specify the structure for national character sets, of which ASCII is the U.S. national set. Certain specific characters are reserved for national use. Table C-1 contains the values and symbols:

**Table C-1: ASCII Characters Reserved for National Use**

Hexadecimal Value	IRV	ASCII
23	#	#
24	###	\$ (General currency symbol vs. dollar sign)
40	@	@
5B	[	[
5C	\	\

(continued on next page)

**Table C-1 (Cont.): ASCII Characters Reserved for National Use**

Hexadecimal Value	IRV	ASCII
5D	]	]
5E	^	^
60	`	`
7B	{	{
7C		
7D	}	}
7E	(tbs)	~ (Overline vs. tilde)

ISO Standard 646 and CCITT Recommendation V.3 (International Alphabet No. 5) are identical to ASCII except that the number sign (23) is represented as ## instead of #, and certain characters are reserved for national use.

**Table C-2: ASCII Codes**

Decimal Code	8-Bit Hexadecimal Code	Character	Remarks
0	00	NUL	Null (tape feed)
1	01	SOH	Start of heading (^A)
2	02	STX	Start of text (end of address, ^B)
3	03	ETX	End of text (^C)
4	04	EOT	End of transmission (shuts off the TWX machine ^D)
5	05	ENQ	Enquiry (WRU, ^E)
6	06	ACK	Acknowledge (RU, ^F)
7	07	BEL	Bell (^G)
8	08	BS	Backspace (^H)
9	09	HT	Horizontal tabulation (^I)
10	0A	LF	Line feed (^J)
11	0B	VT	Vertical tabulation (^K)

(continued on next page)



**Table C-2 (Cont.): ASCII Codes**

<b>Decimal Code</b>	<b>8-Bit Hexadecimal Code</b>	<b>Character</b>	<b>Remarks</b>
12	0C	FF	Form feed (page, ^L)
13	0D	CR	Carriage return ( ^M)
14	0E	SO	Shift out ( ^N)
15	0F	SI	Shift in ( ^O)
16	10	DLE	Data link escape ( ^P)
17	11	DC1	Device control 1 ( ^Q)
18	12	DC2	Device control 2 ( ^R)
19	13	DC3	Device control 3 ( ^S)
20	14	DC4	Device control 4 ( ^T)
21	15	NAK	Negative acknowledge (ERR, ^U)
22	16	SYN	Synchronous idle ( ^V)
23	17	ETB	End-of-transmission block ( ^W)
24	18	CAN	Cancel ( ^X)
25	19	EM	End of medium ( ^Y)
26	1A	SUB	Substitute ( ^Z)
27	1B	ESC	Escape (prefix of escape sequence)
28	1C	FS	File separator
29	1D	GS	Group separator
30	1E	RS	Record separator
31	1F	US	Unit separator
32	20	SP	Space
33	21	!	Exclamation point
34	22	"	Double quotation mark
35	23	#	Number sign
36	24	\$	Dollar sign
37	25	%	Percent sign
38	26	&	Ampersand

(continued on next page)

**Table C-2 (Cont.): ASCII Codes**

<b>Decimal Code</b>	<b>8-Bit Hexadecimal Code</b>	<b>Character</b>	<b>Remarks</b>
39	27	'	Apostrophe
40	28	(	Left (open) parenthesis
41	29	)	Right (close) parenthesis
42	2A	*	Asterisk
43	2B	+	Plus sign
44	2C	,	Comma
45	2D	-	Minus sign, hyphen
46	2E	.	Period (decimal point)
47	2F	/	Slash (slant)
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less than (left angle bracket)
61	3D	=	Equal sign
62	3E	>	Greater than (right angle bracket)
63	3F	?	Question mark
64	40	@	Commercial at
65	41	A	Uppercase A

(continued on next page)

**Table C-2 (Cont.): ASCII Codes**

<b>Decimal Code</b>	<b>8-Bit Hexadecimal Code</b>	<b>Character</b>	<b>Remarks</b>
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K
76	4C	L	Uppercase L
77	4D	M	Uppercase M
78	4E	N	Uppercase N
79	4F	O	Uppercase O
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Left square bracket
92	5C	\	Backslash (reverse slant)

(continued on next page)

**Table C-2 (Cont.): ASCII Codes**

<b>Decimal Code</b>	<b>8-Bit Hexadecimal Code</b>	<b>Character</b>	<b>Remarks</b>
93	5D	]	Right square bracket
94	5E	^	Circumflex (caret)
95	5F	_	Underscore (underline)
96	60	`	Grave accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g
104	68	h	Lowercase h
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u

(continued on next page)

**Table C-2 (Cont.): ASCII Codes**

<b>Decimal Code</b>	<b>8-Bit Hexadecimal Code</b>	<b>Character</b>	<b>Remarks</b>
118	76	v	Lowercase v
119	77	w	Lowercase w
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Left brace
124	7C		Vertical line
125	7D	}	Right brace
126	7E	~	Tilde
127	7F	DEL	Delete (rubout)



# VAX BASIC Keywords

---

The following is a list of the VAX BASIC keywords. Most of the keywords are reserved. The unreserved keywords are marked with a dagger.

**%ABORT**  
**%CDD †**  
**%CROSS**  
**%ELSE**  
**%END**  
**%FROM**  
**%IDENT**  
**%IF**  
**%INCLUDE**  
**%LET**  
**%LIBRARY**  
**%LIST**  
**%NOCROSS**  
**%NOLIST**  
**%PAGE**  
**%PRINT**  
**%SBTTL**  
**%THEN**  
**%TITLE**  
**%VARIANT**  
**ABORT**  
**ABS**  
**ABS%**  
**ACCESS**  
**ACCESS%**  
**ACTIVATE**

---

† Unreserved keyword.

ACTIVE  
ALIGNED  
ALLOW  
ALTERNATE  
AND  
ANGLE †  
ANY  
APPEND  
AREA †  
AS  
ASC  
ASCENDING  
ASCII  
ASK  
AT †  
ATN  
ATN2  
BACK  
BASE  
BASIC  
BEL  
BINARY  
BIT  
BLOCK  
BLOCKSIZE  
BS  
BUCKETSIZE  
BUFFER  
BUFSIZ  
BY  
BYTE  
CALL  
CASE  
CAUSE  
CCPOS  
CHAIN  
CHANGE  
CHANGES  
CHECKING  
CHOICE †  
CHR\$  
CLEAR

---

† Unreserved keyword.



CLIP †  
CLK\$  
CLOSE  
CLUSTERSIZE  
COLOR †  
COM  
COMMON  
COMP%  
CON  
CONNECT  
CONSTANT  
CONTIGUOUS  
CONTINUE  
COS  
COT  
COUNT  
CR  
CTRLC  
CVTF\$  
CVT\$F  
CVT\$\$  
CVT\$%  
CVT%\$  
DAT  
DAT\$  
DATA  
DATE\$  
DEACTIVATE  
DECIMAL  
DECLARE  
DEF  
DEFAULTNAME  
DEL  
DELETE  
DESC  
DESCENDING  
DET  
DEVICE  
DIF\$  
DIM  
DIMENSION  
DOUBLE

---

† Unreserved keyword.

DOUBLEBUF  
DRAW  
DUPLICATES  
DYNAMIC  
ECHO  
EDIT\$  
ELSE  
END  
EQ  
EQV  
ERL  
ERN\$  
ERR  
ERROR  
ERT\$  
ESC  
EXIT  
EXP  
EXPAND †  
EXPLICIT  
EXTEND  
EXTENDSIZE  
EXTERNAL  
FF  
FIELD  
FILE  
FILESIZE  
FILL  
FILL\$  
FILL%  
FIND  
FIX  
FIXED  
FLUSH  
FNAME\$  
FNEND  
FNEXIT  
FONT †  
FOR  
FORMAT\$  
FORTRAN  
FREE

---

† Unreserved keyword.

FROM  
FSP\$  
FSS\$  
FUNCTION  
FUNCTIONEND  
FUNCTIONEXIT  
GE  
GET  
GETRFA  
GFLOAT  
GO  
GOBACK  
GOSUB  
GOTO  
GRAPH  
GRAPHICS †  
GROUP  
GT  
HANDLE  
HANDLER  
HEIGHT †  
HFLOAT  
HT  
IDN  
IF  
IFEND  
IFMORE  
IMAGE  
IMP  
IN †  
INACTIVE  
INDEX †  
INDEXED  
INFORMATIONAL  
INITIAL  
INKEY\$  
INPUT  
INSTR  
INT  
INTEGER  
INV  
INVALID

---

† Unreserved keyword.

ITERATE  
JSB  
JUSTIFY †  
KEY  
KILL  
LBOUND  
LEFT  
LEFT\$  
LEN  
LET  
LF  
LINE  
LINES †  
LINO  
LINPUT  
LIST  
LOC  
LOCKED  
LOG  
LOG10  
LONG  
LSET  
MAG  
MAGTAPE  
MAP  
MAR  
MAR%  
MARGIN  
MAT  
MAX  
METAFILE †  
MID  
MID\$  
MIN  
MIX †  
MOD  
MOD%  
MODE  
MODIFY  
MOVE  
MULTIPOINT †  
NAME

---

† Unreserved keyword.

NEXT  
NO †  
NOCHANGES  
NODATA  
NODUPLICATES  
NOECHO  
NOEXTEND  
NOMARGIN  
NONE  
NOPAGE  
NOREWIND  
NOSPAN  
NOT  
NUL\$  
NUM  
NUM\$  
NUM1\$  
NUM2  
NX  
NXEQ  
OF  
ON  
ONECHR  
ONERROR  
OPEN  
OPTION  
OPTIONAL  
OR  
ORGANIZATION  
OTHERWISE  
OUTPUT  
OVERFLOW  
PAGE  
PATH †  
PEEK  
PI  
PICTURE  
PLACE\$  
PLOT  
POINT †  
POINTS †  
POS

---

† Unreserved keyword.

POS%  
PPS%  
PRIMARY  
PRINT  
PRIORITY †  
PROD\$  
PROGRAM  
PROMPT †  
PUT  
QUO\$  
RAD\$  
RANDOM  
RANDOMIZE  
RANGE †  
RCTRLC  
RCTRLO  
READ  
REAL  
RECORD  
RECORDSIZE  
RECORDTYPE  
RECOUNT  
REF  
REGARDLESS  
RELATIVE  
REM  
REMAP  
RESET  
RESTORE  
RESUME  
RETRY  
RETURN  
RFA  
RIGHT  
RIGHT\$  
RMSSTATUS  
RND  
ROTATE  
ROUNDING  
RSET  
SCALE  
SCRATCH

---

† Unreserved keyword.

SEG\$  
SELECT  
SEQUENTIAL  
SET  
SETUP  
SEVERE  
SGN  
SHEAR  
SHIFT  
SI  
SIN  
SINGLE  
SIZE  
SLEEP  
SO  
SP  
SPACE †  
SPACE\$  
SPAN  
SPEC%  
SQR  
SQRT  
STATUS  
STEP  
STOP  
STR\$  
STREAM  
STRING  
STRING\$  
STYLE †  
SUB  
SUBEND  
SUBEXIT  
SUBSCRIPT  
SUM\$  
SWAP%  
SYS  
TAB  
TAN  
TEMPORARY  
TERMINAL  
TEXT †

---

† Unreserved keyword.

THEN  
TIM  
TIME  
TIME\$  
TO  
TRAN †  
TRANSFORM  
TRANSFORMATION †  
TRM\$  
TRN  
TYP  
TYPE  
TYPE\$  
UBOUND  
UNALIGNED  
UNDEFINED  
UNIT †  
UNLESS  
UNLOCK  
UNTIL  
UPDATE  
USAGE\$  
USEROPEN  
USING  
USR\$  
VAL  
VAL%  
VALUE  
VARIABLE  
VARIANT  
VFC  
VIEWPORT †  
VIRTUAL  
VPS%  
VT  
WAIT  
WARNING  
WHEN  
WHILE  
WINDOW †  
WINDOWSIZE  
WITH †

---

† Unreserved keyword.



**WORD  
WRITE  
XLATE  
XLATE\$  
XOR  
ZER**



## A

- `%ABORT` directive, 3–2
- `ABS%` function, 4–4
- `ABS` function, 4–2
- Absolute value
  - `ABS%` function, 4–4
  - `ABS` function, 4–2
  - `MAG` function, 4–197
- `ACCESS` clause, 4–272, 4–311
- `ACTIVE` clause, 4–287
- `ALLOW` clause, 4–273
- Alphanumeric label, 1–3
  - See also Labels
- `ALTERNATE KEY` clause, 4–278
- Ampersand (&)
  - as a continuation character, 1–7, 1–8, 1–10
  - in `DATA` statement, 4–48
- `/ANALYSIS_DATA` qualifier, 2–11
- Angle types
  - with `OPTION` statement, 4–285
- ANSI standards, B–1 to B–10
- `/ANSI_STANDARD` qualifier, 2–12
- `APPEND` command, 2–5 to 2–7
- Arc tangent, 4–7
- Arithmetic operators, 1–36
- Array
  - assigning values to, 4–214, 4–217, 4–226, 4–325
  - bounds, 4–77, 4–213, 4–217, 4–220, 4–224, 4–226
  - converting with `CHANGE` statement, 4–22
  - creating with `COMMON` statement, 4–30
  - creating with `DECLARE` statement, 4–54
  - creating with `DIM` statement, 4–76
  - creating with `MAP` statement, 4–202

## Array (Cont.)

- creating with `MAT` statement, 4–212, 4–217, 4–220, 4–223, 4–226
- data type of, 4–76
- dimensions of, 4–76
- dynamic, 4–76, 4–78, 4–79
- elements, 4–77
- element zero, 4–78, 4–215, 4–218, 4–221, 4–224, 4–227, 4–240
- initialization of, 4–214
- initializing, 4–80
- inversion of, 4–215
- redimensioning with `MAT` statement, 4–213, 4–215, 4–218, 4–221, 4–227
- static, 4–76, 4–77
- transposing, 4–215
- virtual, 4–56, 4–76, 4–79, 4–109

## Arrays, 1–19

- array elements, 1–19
- definition of, 1–20
- dimensions of, 1–21
- element zero, 1–20
- naming, 1–19, 1–22
- size limits, 1–21
- virtual, 1–23

## ASCENDING keys, 4–276, 4–278

## ASCII

- character codes, C–1
- characters, 1–34, 1–43, 4–25, C–1
- character set, 1–10
- conversion, 4–22, 4–25
- converting to, 4–6
- radix, 1–31

## ASCII function, 4–6

## ASSIGN command, 2–7 to 2–9

## Asterisk (\*)

## Asterisk (\*) (Cont.)

- in PRINT USING statement, 4-300
- with HELP command, 2-29

ATN function, 4-7

/AUDIT qualifier, 2-13

## B

---

### Backslash (\)

- in continued lines, 1-8
- in multi-statement lines, 1-8
- in PRINT USING statement, 4-303
- statement separator, 1-8

### BASIC-PLUS-2

- compatibility, A-1 to A-13
- transporting programs, A-1 to A-13

Binary radix, 1-31

### Blank-if-zero field

- in PRINT USING statement, 4-300

### Block I/O file

- finding records in, 4-117
- opening, 4-276
- retrieving records sequentially in, 4-145
- writing records to, 4-311

BLOCKSIZE clause, 4-273

### Block statement

- ending, 4-85
- exiting, 4-98

### Bounds, 1-19

- default for implicit arrays, 4-78, 4-213, 4-217, 4-220, 4-224, 4-226
- lower bounds with COMMON statement, 4-30
- lower bounds with DECLARE statement, 4-55
- lower bounds with DIM statement, 4-77
- lower bounds with MAP DYNAMIC statement, 4-206
- lower bounds with records, 4-329
- maximum, 1-20
- upper bounds with COMMON statement, 4-30
- upper bounds with DECLARE statement, 4-55
- upper bounds with DIM statement, 4-77
- upper bounds with MAP DYNAMIC statement, 4-206

### Bucket

- creating with BUCKETSIZE clause, 4-273
- locking, 4-114, 4-145
- unlocking, 4-114, 4-129, 4-145

BUCKETSIZE clause, 4-273

BUFFER clause, 4-273

BUFSIZ function, 4-9

BYTE data type, 1-11

/BYTE qualifier, 2-13

## C

---

%CROSS directive, 3-3

CALL statement, 4-10 to 4-15

- with SUB subprograms, 4-387

Caret (^) in PRINT USING statement, 4-300

CASE clause, 4-365

CASE ELSE clause, 4-365

CAUSE ERROR statement, 4-16 to 4-17

CCPOS function, 4-18

### CDD

- including definitions from, 3-10, 3-23

CDD (Common Data Dictionary)

- including definitions from, 1-9

CDD/Plus (Common Data Dictionary/Plus)

- and RECORD statement, 4-328

CD formatting character

- in PRINT USING statement, 4-300

Centered field

- in PRINT USING statement, 4-303

C formatting character

- in PRINT USING statement, 4-303

CHAIN statement, 4-20 to 4-21

CHANGES clause, 4-278

CHANGE statement, 4-22 to 4-24

### Character

- ASCII, 4-6, 4-25

- formatting with PRINT USING statement, 4-299 to 4-304

- lowercase, 4-302

- uppercase, 4-302

CHARACTER data type, 1-33

Character position

- CCPOS function, 4-18

- of substring, 4-169, 4-292

Characters

- ASCII, 1-34, 1-43

- data type suffix, 1-14

- lowercase, 2-25

- nonprinting, 1-34

- processing of, 1-11

- uppercase, 2-25

- wildcard, 2-29

Character set

- translating with XLATE\$ function, 4-428

Character sets

- ASCII, 1-10

- VAX BASIC, 1-10

/CHECK qualifier, 2-11

CHR\$ function, 4-25

Clauses

ACCESS, 4-116, 4-144, 4-272, 4-311  
ACTIVE, 4-287  
ALLOW, 4-114, 4-141, 4-273  
ALTERNATE KEY, 4-278  
BLOCKSIZE, 4-273  
BUCKETSIZE, 4-273  
BUFFER, 4-273  
BY, 4-11, 4-106, 4-135, 4-388  
CASE, 4-365  
CASE ELSE, 4-365  
CHANGES, 4-278  
CONNECT, 4-274  
CONTIGUOUS, 4-273  
COUNT, 4-310, 4-412  
DEFAULTNAME, 4-271, 4-274  
DUPLICATES, 4-278, 4-312  
ELSE, 4-155  
END IF, 4-155  
EXTENDSIZE, 4-274  
FILESIZE, 4-274  
FOR, 4-271  
GROUP, 4-329  
KEY, 4-113, 4-141, 4-143, 4-343  
MAP, 4-204, 4-274  
NOREWIND, 4-275  
NOSPAN, 4-275  
ORGANIZATION, 4-276  
OTHERWISE, 4-265, 4-267  
PRIMARY KEY, 4-277  
RECORD, 4-112, 4-140, 4-310, 4-311  
RECORDSIZE, 4-204, 4-279, 4-310  
RECORDTYPE, 4-278  
REGARDLESS, 4-114, 4-141  
RFA, 4-112, 4-140  
STEP, 4-124  
TEMPORARY, 4-280  
UNLOCK EXPLICIT, 4-113, 4-114, 4-141, 4-280  
UNTIL, 4-125  
USEROPEN, 4-280  
VARIANT, 4-329  
WAIT, 4-142  
WHILE, 4-125  
WINDOWSIZE, 4-281

CLOSE statement, 4-27

Colon (:)

in labels, 1-3

Comma (,)

in DATA statement, 4-49

in DELETE command, 2-22

Comma (,) (Cont.)

in INPUT LINE statement, 4-166

in INPUT statement, 4-162

in LINPUT statement, 4-186

in LIST command, 2-33

in MAT PRINT statement, 4-223

in PRINT statement, 4-294

in PRINT USING statement, 4-299

Command files for environment, 2-2

Command qualifiers, 2-10 to 2-40

VAX BASIC, 2-10 to 2-40

Comment

field, 1-52

in DATA statement, 1-53, 4-48

in REM statement, 4-335

processing of, 1-11

REM statement, 1-54

transferring control to, 1-53

Comment fields

terminating, 1-53

Comments

in environment command files, 2-2

processing of, 1-11

COMMON area

size of, 4-32

COMMON statement, 4-29 to 4-34

with FIELD statement, 4-109

COMP% function, 4-35

Compilation

conditional, 3-8, 3-29

controlling with OPTION statement, 4-285

control of, 1-9, 2-58

control of listing, 3-3, 3-17, 3-18, 3-20, 3-21,  
3-22, 3-25, 3-27

creating relationships, 3-23

including from CDD, 1-9, 3-10

including from text library, 3-10

including source code, 1-9, 3-10

listing, 2-10

terminating with %ABORT directive, 3-2

Compilation qualifiers, 2-10 to 2-40

VAX BASIC, 2-10 to 2-40

COMPILE command, 2-10 to 2-20

VAX BASIC qualifiers, 2-10 to 2-40

Compiler directives, 1-9

Concatenation

of COMMON areas, 4-32

string, 1-7, 1-36, 1-41

Conditional branching

IF statement, 4-155

ON...GOSUB statement, 4-265

## Conditional branching (Cont.)

- ON...GOTO statement, 4-267
  - SELECT statement, 4-365
- ## Conditional compilation, 1-9
- %VARIANT directive, 3-29
  - with %IF directive, 3-8
- ## Conditional expression
- FOR statement, 4-125
  - IF statement, 4-155
  - UNLESS statement, 4-406
  - UNTIL statement, 4-410
  - WHILE statement, 4-426
- ## Conditional expressions, 1-41
- definition of, 1-42
  - in %LET directive, 3-15
- ## Conditional loops, 4-124, 4-410, 4-426
- ## CON function, 4-214
- ## CONNECT clause, 4-274
- ## Constant
- declaring, 4-56
  - external, 4-103
  - with OPTION CONSTANT TYPE, 4-286
- ## Constants, 1-23
- declaring, 1-30
  - default data type, 1-23
  - definition of, 1-23
  - floating-point, 1-24
  - integer, 1-26
  - lexical, 3-8
  - named, 1-28
  - naming, 1-24, 1-29
  - numeric, 1-24
  - packed decimal, 1-27
  - string, 1-27
  - types of, 1-23
- ## CONTIGUOUS clause, 4-273
- ## Continuation characters
- ampersand, 1-8
  - backslash, 1-8
- ## CONTINUE command, 2-21
- with RUN command, 2-49
- ## CONTINUE statement, 4-37
- ## Control
- transferring into DEF functions, 4-265, 4-267
  - transferring into FOR...NEXT loops, 4-149, 4-151, 4-265, 4-267
  - transferring into SELECT blocks, 4-265, 4-267
  - transferring into UNTIL loops, 4-149, 4-151, 4-265, 4-267
  - transferring into WHILE loops, 4-149, 4-151, 4-265, 4-267

## Control (Cont.)

- transferring to a label, 4-149, 4-151
  - transferring with CALL statement, 4-10
  - transferring with CHAIN statement, 4-20
  - transferring with GOSUB statement, 4-149
  - transferring with GOTO statement, 4-151
  - transferring with IF statement, 4-155
  - transferring with ON...GOSUB statement, 4-265
  - transferring with ON...GOTO statement, 4-267
  - transferring with RESUME statement, 4-168, 4-187, 4-346
  - transferring with RETURN statement, 4-350
- ## Conversion
- array to string variable, 4-23
  - string variable to array, 4-22
- ## Conversion functions
- CVT\$% function, 4-44
  - CVT\$F function, 4-44
  - CVT%\$ function, 4-44
  - CVTF\$ function, 4-44
  - DECIMAL function, 4-52
  - INTEGER function, 4-173
  - NUM\$ function, 4-254
  - NUM1\$ function, 4-256
  - RAD\$ function, 4-317
  - REAL function, 4-326
  - STR\$ function, 4-382
  - VAL% function, 4-415
  - VAL function, 4-413
  - XLATE\$ function, 4-428
- ## Copying VAX BASIC source text, 3-10
- ## COS function, 4-39
- ## Cosine, 4-39
- ## COUNT clause, 4-310
- with fixed-length records, 4-412
  - with variable-length records, 4-412
- ## CPU time, 4-399
- ## Credit-debit field
- in PRINT USING statement, 4-300
- ## Cross-reference table
- %CROSS directive, 3-3
  - %NOCROSS directive, 3-18
- ## /CROSS\_REFERENCE qualifier, 2-13
- ## CTRL/C function
- trapping, 4-321
  - with RECOUNT function, 4-333
- ## CTRL/C key, 4-41
- ## CTRL/Z function, 2-27
- with INPUT LINE statement, 4-168
  - with INPUT statement, 4-164
  - with LINPUT statement, 4-188

CTRLC function, 4-41  
 See also RCTRLC function  
 Cursor position  
 CCPOS function, 4-18  
 TAB function, 4-396  
 CVT\$\$ function, 4-43  
 See also EDIT\$ function  
 CVTxx function, 4-44  
 with FIELD statement, 4-108

## D

%DECLARED directive, 3-3 to 3-5, A-2  
 Data  
 transferring with MOVE statement, 4-238  
 DATA statement, 4-48 to 4-49  
 See also READ statement  
 comment fields in, 1-53  
 in DEF\* function, 4-68  
 in DEF function, 4-62  
 in multi-statement lines, 1-9  
 terminating, 4-48  
 with MAT READ statement, 4-226  
 with READ statement, 4-324  
 with RESTORE statement, 4-343  
 Data structures  
 defining, 4-328  
 Data type defaults, 1-14, 1-15  
 constants, 1-23  
 Data type functions  
 DECIMAL function, 4-52  
 INTEGER function, 4-173  
 REAL function, 4-326  
 Data type keywords, 1-11  
 Data types, 1-11, 1-13t  
 BYTE, 1-11  
 CHARACTER, 1-33  
 DECIMAL, 1-12  
 decimal overflow checking, 4-286  
 defining with RECORD statement, 4-328  
 DOUBLE, 1-12  
 GFLOAT, 1-12  
 HFLOAT, 1-12  
 in LET statement, 4-184  
 in logical expressions, 1-46  
 in numeric expressions, 1-38  
 INTEGER, 1-11  
 integer overflow checking, 4-286  
 keywords, 1-11, 1-13  
 LONG, 1-11  
 numeric literal notation, 1-32

Data types (Cont.)  
 precision, 1-13  
 precision in PRINT statement, 4-296  
 precision in PRINT USING statement, 4-299  
 promotion rules, 1-38  
 range, 1-13  
 REAL, 1-12  
 results for DECIMAL data, 1-40  
 results for GFLOAT and HFLOAT, 1-39  
 results in expressions, 1-38  
 RFA, 1-12  
 setting defaults with OPTION statement, 4-286  
 SINGLE, 1-12  
 size, 1-13  
 storage of, 1-11, 1-13  
 STRING, 1-12  
 suffix characters, 1-14  
 WORD, 1-11  
 Data typing  
 explicit, 1-15  
 implicit, 1-14  
 with declarative statements, 1-15  
 with suffix characters, 1-14  
 DATE\$ function, 4-50  
 Date and time functions  
 TIME\$ function, 4-401  
 TIME function, 4-399  
 DCL commands  
 in environment, 2-4  
 Debit-credit field  
 in PRINT USING statement, 4-300  
 /DEBUG qualifier, 2-14  
 /DEBUG qualifier  
 with RUN command, 2-49  
 DECIMAL data type, 1-12  
 constants, 1-27  
 format of, 1-12  
 overflow checking, 4-286  
 promotion rules, 1-39  
 rounding, 4-286  
 storage of, 1-12  
 DECIMAL function, 4-52  
 Decimal radix, 1-31  
 /DECIMAL\_SIZE qualifier, 2-14  
 Declarative statements  
 COMMON statement, 4-33  
 DECLARE statement, 4-54  
 EXTERNAL statement, 4-102  
 MAP statement, 4-202  
 DECLARE statement, 4-54 to 4-58  
 CONSTANT, 1-30, 1-35

- DEF\* function
  - error handling in, 4-68, 4-86
  - multi-line, 4-67
  - parameters, 4-66, 4-67
  - recursion in, 4-68
  - single-line, 4-67
- DEF\* statement, 4-65 to 4-69
- Default
  - BUCKETSIZE clause, 4-273
  - COMMON name, 4-29
  - data type, 4-286
  - DEFAULTNAME clause, 4-274
  - error handling, 4-259
  - file name, 4-20, 4-271
  - overriding with DECLARE statement, 4-55, 4-57
  - overriding with EXTERNAL statement, 4-102
  - parameter-passing mechanisms, 4-11 to 4-13, 4-105, 4-135, 4-388
  - RECORDSIZE clause, 4-280
  - scale factor, 4-286
  - setting with OPTION statement, 4-283
  - WINDOWSIZE clause, 4-281
- DEFAULTNAME clause, 4-271, 4-274
- Defaults
  - COMPILE command, 2-10
  - constants, 1-23
  - data type, 1-14, 1-15
  - displaying, 2-60
  - EDIT command, 2-24
  - file name, 2-38, 2-40, 2-41, 2-43, 2-48, 2-51, 2-62
  - floating-point constants, 1-24
  - implicitly declared variables, 1-17
  - integer constants, 1-26
  - listing file, 2-10
  - LOAD command, 2-35
  - numeric constants, 1-24
  - object module name, 2-10
  - overriding with COMPILE command, 2-10
  - overriding with RUN command, 2-48
  - radix, 1-31
  - RESEQUENCE command, 2-45
  - SCALE command, 2-53
  - SEQUENCE command, 2-56
  - SET command, 2-58
  - SHOW command, 2-60
- DEF function
  - ending, 4-85
  - error handling in, 4-62, 4-258, 4-260, 4-345
  - exiting, 4-97
  - recursion in, 4-62
- DEF function (Cont.)
  - transferring control into, 4-62, 4-265, 4-267
  - with INPUT LINE statement, 4-166
  - with INPUT statement, 4-162
  - with LINPUT statement, 4-186
  - with READ statement, 4-324
- DEF statement, 4-59 to 4-64
  - multi-line, 4-61
  - parameters, 4-60, 4-61
  - single-line, 4-61
- DELETE command, 2-22 to 2-23
- DELETE statement, 4-70
  - with UNLOCK statement, 4-407
- Delimiter
  - EDIT command, 2-24
  - in DATA statement, 4-49
  - string literal, 1-27
- /DEPENDENCY\_DATA qualifier, 2-11
- DESCENDING keys, 4-276, 4-278
- Descriptor, 4-15, 4-106, 4-135, 4-388
- /DESIGN qualifier, 2-11
- Detached handler, 4-422
- Determinant, 4-72
- DET function, 4-72
- /DIAGNOSTICS qualifier, 2-11
- DIF\$ function, 4-74
- Dimension
  - of arrays, 4-76
- Dimensions
  - of arrays, 1-21
- DIMENSION statement, 4-76 to 4-80
  - See also DIM statement
- DIM statement, 4-76 to 4-80
  - executable, 4-78, 4-79
  - nonvirtual, nonexecutable, 4-77
  - virtual, 4-77
  - with MAT statement, 4-213, 4-215, 4-218, 4-221, 4-224, 4-226
- Directives
  - %DECLARED, A-2
  - %INCLUDE %FROM %CDD, A-2
  - %INCLUDE %FROM %LIBRARY, A-2
  - %PRINT, A-2
  - %REPORT %DEPENDENCY, A-2
- Documentation
  - program, 1-52
- Dollar sign (\$)
  - in DECLARE statement, 4-55
  - in DEF\* statement names, 4-66
  - in DEF names, 4-59
  - in PRINT USING statement, 4-300



Dollar sign (\$) (Cont.)

- in variable names, 1-17
- suffix character, 1-14

DOUBLE data type, 1-12

/DOUBLE qualifier, 1-16, 2-14

DUPLICATES clause, 4-278, 4-312

Dynamic array, 4-76, 4-78, 4-79

Dynamic mapping, 4-108, 4-205, 4-337

Dynamic storage, 4-205, 4-337, 4-339

## E

---

ECHO function, 4-81

- See also NOECHO function

EDIT\$ function, 4-83

- values, 4-83 to 4-84

EDIT command, 2-24 to 2-26

E-format notation

- field in PRINT USING statement, 4-300
- with PRINT statement, 4-295
- with STR\$ function, 4-382

E formatting character

- in PRINT USING statement, 4-303

Elementary record components, 4-330

ELSE clause, 4-155

E mathematical constant, 4-100

END statement, 4-85 to 4-88

- SUB subprograms, 4-387

E notation, 1-25

- numbers in, 1-25t

Environment

- comments in immediate mode, 2-2

Equivalence name, 2-7

ERL function, 4-89

- with labels, 1-4
- with RESEQUENCE command, 2-46

ERN\$ function, 4-91

ERR function, 4-93

Error

- severity level, 4-285

Error condition

- clearing with CONTINUE statement, 4-37

Error handling

- disabling, 4-263
- ERL function, 4-89
- ERN\$ function, 4-91
- ERR function, 4-93
- ERT\$ function, 4-95
- in DEF\* functions, 4-68
- in DEF functions, 4-62, 4-86, 4-258, 4-260
- in FOR...NEXT loops, 4-346

Error handling (Cont.)

- in subprograms, 4-87, 4-98, 4-136, 4-259
- in UNTIL loops, 4-346
- in WHILE loops, 4-346
- ON ERROR GO BACK statement, 4-258
- ON ERROR GOTO 0 statement, 4-263
- ON ERROR GOTO statement, 4-260
- OPTION HANDLE, 4-285
- recursion in, 4-261
- RESUME statement, 4-345

Error handling functions

- CTRLC function, 4-41
- ERL function, 4-89
- ERN\$ function, 4-91
- ERR function, 4-93
- ERT\$ function, 4-95
- RCTRLC function, 4-321

Error number, 4-93

Error text, 4-95

ERT\$ function, 4-95

Evaluation

- of expressions, 1-50
- of logical expressions, 1-48
- of numeric relational expressions, 1-42
- of operators, 1-50
- of SELECT statement, 4-366
- of string relational expressions, 1-43

Exclamation point (!)

- in comment fields, 1-52
- in PRINT USING statement, 4-303

Executable

- statements, 1-4

Execution

- continuing, 2-21, 2-49
- of multi-statement lines, 1-8
- of statements, 1-8
- of system commands, 2-4
- program, 2-48
- stopping, 2-21, 2-49, 4-380
- suspending, 4-373, 4-419

EXIT command, 2-27 to 2-28

EXIT statement, 4-97 to 4-99

EXP function, 4-100

Explicit

- creation of arrays, 4-76
- data typing, 1-15, 4-283
- declaration of variables, 1-19
- literal notation, 1-31
- loop iteration, 4-175
- record locking, 4-70, 4-113, 4-114, 4-141, 4-145, 4-280

Exponential notation, 1–25, 4–295  
in PRINT USING statement, 4–300  
numbers in, 1–25t  
with PRINT statement, 4–295

Exponentiation, 4–100

Exponentiation operator, B–10

Expressions, 1–36

- conditional, 1–41
- conditional in %LET directive, 3–15
- definition of, 1–36
- evaluation of, 1–50
- lexical, 3–8, 3–15, 3–29
- logical, 1–45
- mixed-mode, 1–38
- numeric, 1–36
- numeric relational, 1–42
- operator precedence in, 1–50
- parentheses in, 1–50
- relational, 1–42
- string, 1–41
- string relational, 1–43
- types of, 1–36

Extended field

- in PRINT USING statement, 4–303

EXTENDSIZE clause, 4–274

External

- constant, 4–103
- function, 4–103
- picture, 4–103
- subprogram, 4–135
- subroutine, 4–103
- variable, 4–103

External constants, 1–31

- naming, 1–31

EXTERNAL statement, 4–102 to 4–107

- CONSTANT, 1–31
- parameters, 4–103

External variables

- naming, 1–17

## F

---

FAB status, 4–354

Field

- asterisk-filled, 4–300
- blank-if-zero, 4–300
- centered, 4–303
- credit or debit, 4–300
- exponential, 4–300
- extended, 4–303
- floating dollar sign, 4–300

Field (Cont.)

- GROUP, 4–331
- left-justified, 4–302
- multiple fields within a format string, 4–301
- one-character, 4–303
- right-justified, 4–303
- trailing minus sign, 4–300
- VARIANT, 4–331
- zero-filled, 4–300

Fields

- comment, 1–52

FIELD statement, 4–108 to 4–110

File attributes

- BLOCKSIZE clause, 4–273
- CONTIGUOUS clause, 4–273
- EXTENDSIZE clause, 4–274
- FILESIZE clause, 4–274
- magnetic tape, 4–273

File names

- CHAIN statement default, 4–20
- COMPILE command default, 2–10
- LOAD command default, 2–35
- NEW command default, 2–38
- OLD command default, 2–40
- OPEN default, 4–271
- RENAME command default, 2–41
- REPLACE command default, 2–43
- RUN command default, 2–48
- SAVE command default, 2–51
- UNSAVE command default, 2–62

File organization

- indexed, 4–276
- relative, 4–277
- sequential, 4–277
- undefined, 4–276
- virtual, 4–276

File-related functions

- BUFSIZ function, 4–9
- CCPOS function, 4–18
- FSP\$ function, 4–131
- GETRFA function, 4–147
- MAR function, 4–209
- RECOUNT function, 4–333
- STATUS function, 4–378

Files

- accessing, 4–139
- block I/O, 4–117, 4–145, 4–276, 4–311
- closing, 4–27
- deleting, 2–62, 4–177, 4–280
- deleting records in, 4–70, 4–361
- finding buffer size, 4–9

## Files (Cont.)

- %INCLUDE, 3-10, 3-11
- %INCLUDE directive, 2-45
- indexed, 4-70, 4-117, 4-145, 4-273, 4-275, 4-276, 4-311, 4-344, 4-412
- locating, 4-110
- magnetic tape, 4-199, 4-273, 4-343
- opening, 4-269
- relative, 4-70, 4-117, 4-145, 4-273, 4-276, 4-280, 4-311, 4-412
- renaming, 4-242
- restoring data, 4-343
- sequential, 4-117, 4-145, 4-275, 4-276, 4-280, 4-294, 4-311, 4-361, 4-412
- terminal-format, 4-162, 4-166, 4-186, 4-210, 4-217, 4-220, 4-223, 4-248, 4-279, 4-280, 4-294
- virtual, 4-280, 4-344

FILESIZE clause, 4-274

FILL, 4-202, 4-239, 4-338

FILL\$, 4-202, 4-239, 4-338

FILL\$ keyword, 4-30

FILL%, 4-202, 4-239, 4-338

FILL% keyword, 4-30

FILL items

- formats and storage, 4-31 to 4-32
- in COMMON statement, 4-30
- in MAP statement, 4-202
- in MOVE statement, 4-239
- in REMAP statement, 4-338

FILL keyword, 4-30

FIND statement, 4-111 to 4-118

- with UNLOCK statement, 4-407
- with UPDATE statement, 4-411

FIX function, 4-119

- compared with INT function, 4-172

/FLAG qualifier, 2-14

Floating dollar sign field

- in PRINT USING statement, 4-300

Floating-point

- constants, 1-24
- data types, 1-12
- promotion rules, 1-38
- variables, 1-18

FNEND statement, 4-121

- See also END statement

FNEXIT statement, 4-122

- See also EXIT statement

FOR...NEXT loops, 4-123 to 4-126, 4-244

- conditional, 4-124
- error handling in, 4-346

## FOR...NEXT loops (Cont.)

- explicit iteration of, 4-175
- nested, 4-124
- transferring control into, 4-124, 4-149, 4-151, 4-265, 4-267
- unconditional, 4-124

FOR clause, 4-271

Format

- characters in PRINT USING statement, 4-299
- combination of characters in PRINT USING statement, 4-301
- defaults for U.S. currency, 4-302
- E, 4-295
- exponential, 4-295
- multiple print fields with PRINT USING statement, 4-301
- of data in DATA statement, 4-49
- of FILL items, 4-31 to 4-32
- of keywords, 1-5
- of labels, 1-3
- of line numbers, 1-2
- of multi-line REM statement, 4-335
- of multi-statement lines, 1-8, 1-9
- of program lines, 1-1
- of statements, 1-4
- Radix-50, 4-317

FORMAT\$ function, 4-127

FOR statement, 4-123 to 4-126

FOR\_NEXT loops

- exiting, 4-98

FREE statement, 4-129

FSP\$ function, 4-131

Function

- declaring, 4-55, 4-59, 4-65
- external, 4-103
- initialization of, 4-62, 4-68
- invocation of, 4-61, 4-67
- naming, 4-59, 4-65
- parameters, 4-60, 4-66
- user-defined, 4-59, 4-65

FUNCTIONEND statement, 4-137

- See also END statement

FUNCTIONEXIT statement, 4-138

- See also EXIT statement

Functions

- lexical, 3-8, 3-15, 3-29

FUNCTION statement, 4-133 to 4-136

FUNCTION subprograms

- naming, 4-133
- parameters, 4-134

## G

---

GETRFA function, 4-147  
GET statement, 4-139 to 4-146  
    with UNLOCK statement, 4-407  
    with UPDATE statement, 4-411  
GFLOAT data type, 1-12  
/GFLOAT qualifier, 2-15  
GOSUB statement, 4-149  
    inside WHEN blocks, 4-149  
    with RETURN statement, 4-350  
GOTO statement, 4-151  
    inside WHEN blocks, 4-151  
GROUP clause, 4-329

## H

---

Handler  
    attached, 4-422  
    enter, 4-153  
    exit, 4-153  
HANDLER statement, 4-153 to 4-154  
HELP command, 2-29 to 2-30  
Hexadecimal radix, 1-31  
HFLOAT data type, 1-12  
/HFLOAT qualifier, 2-15  
Hyphen (-)  
    in DELETE command, 2-22  
    in LIST command, 2-33

## I

---

%IDENT directive, 3-6 to 3-7  
%IF-%THEN-%ELSE-%END %IF directive, 3-8 to 3-9  
%IF...%THEN...%ELSE...%END %IF directive  
    with RESEQUENCE command, 2-45  
%INCLUDE %FROM %CDD directive, A-2  
%INCLUDE %FROM %LIBRARY directive, A-2  
%INCLUDE directive, 3-10 to 3-14  
    with RESEQUENCE command, 2-45  
I/O  
    characters transferred, 4-333  
    closing files, 4-27, 4-87  
    deleting records, 4-70  
    dynamic mapping, 4-337  
    finding records, 4-112  
    locking records, 4-113, 4-114, 4-141, 4-142, 4-280  
    matrix, 4-250, 4-252  
    moving data, 4-238

## I/O (Cont.)

    opening files, 4-269  
    retrieving records, 4-144  
    unlocking records, 4-129, 4-280, 4-407  
    updating records, 4-411  
    with CHAIN statement, 4-20  
    writing records, 4-311  
Identifiers  
    PROGRAM statement, 1-6  
IDENTIFY command, 2-31  
Identity matrix, 4-214  
IDN function, 4-214  
IF...THEN...ELSE statement, 4-155 to 4-157  
    labels in, 1-4  
    multi-line format, 1-8  
Immediate mode, 2-21, 2-49  
Implicit  
    continuation of lines, 1-8  
    creation of arrays, 4-78, 4-213, 4-217, 4-220, 4-224, 4-226  
    data typing, 1-14  
    declaration of variables, 1-17  
Indexed files, 4-276  
    ALTERNATE KEY clause, 4-278  
    BUCKETSIZE clause, 4-273  
    CHANGES clause, 4-278  
    deleting records in, 4-70  
    DUPLICATES clause, 4-278  
    finding records in, 4-117  
    MAP clause, 4-275  
    opening, 4-276  
    PRIMARY KEY clause, 4-277  
    restoring data in, 4-344  
    retrieving records sequentially in, 4-145  
    segmented keys in, 4-278  
    updating, 4-412  
    writing records to, 4-311  
Initialization  
    in subprograms, 4-136, 4-388  
    of arrays, 4-214  
    of DEF\* functions, 4-68  
    of DEF functions, 4-62  
    of dynamic arrays, 4-80  
    of variables, 1-22, 4-56  
    of variables in COMMON statement, 4-33  
    of virtual arrays, 4-79  
INKEY\$ function, 4-158 to 4-161  
    with WAIT clause, 4-158  
INPUT LINE statement, 4-166 to 4-168  
INPUT statement, 4-162 to 4-165

INQUIRE command 2-32  
  See also HELP command  
Instance, 4-331  
  RECORD, 4-331  
INSTR function, 4-169 to 4-170  
  See also POS function  
Integer  
  constants, 1-26  
  data types, 1-11  
  overflow checking, 4-286  
  promotion rules, 1-38  
  suffix character, 1-14  
  variables, 1-18  
INTEGER data type, 1-11  
INTEGER function, 4-173  
/INTEGER\_SIZE qualifier, 2-11  
Internal constants  
  naming, 1-29  
Internal variables  
  naming, 1-17  
INT function, 4-171  
INV function, 4-215  
ITERATE statement, 4-175  
Iteration  
  of FOR loops, 4-125  
  of loops, 4-175  
  of UNTIL loops, 4-410  
  of WHILE loops, 4-426

## K

---

KEY clause, 4-113, 4-143  
  FIND statement, 4-115  
  GET statement, 4-141  
  RESTORE statement, 4-343  
  segmented keys, 4-277  
Keys  
  ascending and descending, 4-276, 4-278  
Keywords  
  data type, 1-11  
  definition of, 1-5  
  function of, 1-5  
  in RECORD, 4-329  
  list of, D-1  
  reserved and unreserved, D-1  
  restrictions, 1-5  
  spacing requirements, 1-5  
KILL statement, 4-177

## L

---

%LET directive, 3-15 to 3-16  
%LIST directive, 3-17  
Labels  
  defining, 1-3  
  format of, 1-3  
  function of, 1-3, 1-4  
  referencing, 1-3  
  transferring control to, 4-149, 4-151  
  with ITERATE statement, 4-175  
LBOUND function, 4-178  
LEFT\$ function, 4-180  
  See also SEG\$ function  
Left-justification  
  PRINT USING statement, 4-302  
  with LSET statement, 4-195  
LEN function, 4-182  
Length  
  of labels, 1-3  
  of STRING data, 1-13  
  variable names, 1-17  
LET statement, 4-184  
Letters  
  lowercase, 1-11, 2-25, 4-302  
  uppercase, 1-11, 2-25, 4-302  
Lexical  
  constants, 3-8  
  expressions, 3-8, 3-15, 3-29  
  functions, 3-8, 3-15, 3-29  
  operators, 3-4, 3-8, 3-15  
  order, 1-10  
  variables, 3-15  
Lexical variables  
  assigning values to, 3-15  
  naming, 3-15  
L formatting character  
  in PRINT USING statement, 4-302  
Libraries  
  text, 3-13  
Line numbers  
  automatic sequencing, 2-56  
  in %INCLUDE file, 2-45, 3-10  
  in RESEQUENCE command, 2-45  
  range of, 1-2  
Lines  
  continued, 1-7  
  displaying, 2-33  
  editing, 2-24  
  elements of, 1-1  
  format of, 1-1

## Lines (Cont.)

- length of, 1-2
- multi-statement, 1-8
- order of, 1-10, 2-45
- single-statement, 1-7
- terminating, 1-2, 1-10, 1-11
- /LINES qualifier, 1-2, 2-15, 4-89, 4-380
- Line terminator, 1-2, 1-10, 1-11
  - with DATA statement, 4-48
  - with INPUT LINE statement, 4-167
  - with INPUT statement, 4-163
  - with LINPUT statement, 4-187
- LINPUT statement, 4-186 to 4-188
- LIST command, 2-33 to 2-34
- Listing file
  - control of, 1-9, 3-3, 3-17, 3-18, 3-20, 3-21
  - %CROSS directive, 3-3
  - defaults, 2-10
  - included code, 3-10
  - %LIST directive, 3-17
  - %NOCROSS directive, 3-18
  - %NOLIST directive, 3-20
  - %PAGE directive, 3-21
  - %PRINT directive, 3-22
  - %SBTTL directive, 3-25
  - subtitle, 3-25
  - title, 3-27
  - %TITLE directive, 3-27
  - version identification, 3-6
- Listing line numbers, 1-8
- LISTNH command, 2-33
  - See also LIST command
- Literal
  - explicit notation, 1-31
  - numeric, 1-24
  - string, 1-7, 1-11, 1-27, 1-45, 4-301, 4-303
- LOAD command, 2-35 to 2-36
  - with RUN command, 2-49
  - with SCRATCH command, 2-55
- Local copy, 4-15
- LOC function, 4-189
- Lock checking
  - REGARDLESS clause, 4-114, 4-141
  - WAIT clause, 4-142
- LOCK command, 2-37
  - See also SET command
- LOG10 function, 4-193
- Logarithms
  - common, 4-193
  - natural, 4-191
- LOG function, 4-191

- Logical expressions, 1-45
  - compared with relational, 1-46, 1-48
  - data types in, 1-46
  - definition of, 1-42
  - evaluation of, 1-48
  - format of, 1-45
  - logical operators, 1-46
  - truth tables, 1-47
  - truth tests, 1-47
- Logical name, 2-7
- Logical operators, 1-46
- LONG data type, 1-11
- /LONG qualifier, 2-16
- Loops

- conditional, 4-124
- exiting, 4-97
- FOR...NEXT, 4-123
- iteration of, 4-125, 4-175, 4-410, 4-426
- nested FOR...NEXT, 4-124
- unconditional, 4-124
- UNTIL statement, 4-409
- WHILE statement, 4-426
- Lowercase letters
  - in EDIT command, 2-25
  - in PRINT USING statement, 4-302
  - processing of, 1-11
- LSET statement, 4-195

## M

---

- /MACHINE\_CODE qualifier, 2-16
- MAG function, 4-197
- Magnetic tape files
  - BLOCKSIZE clause, 4-273
  - MAGTAPE function, 4-199
  - NOREWIND clause, 4-275
  - RESTORE statement, 4-343
- MAGTAPE function, 4-199 to 4-200
  - performing functions in VAX BASIC, 4-199 to 4-200
- MAP
  - FILL item formats and storage, 4-31 to 4-32
- MAP area
  - naming, 4-201
- MAP clause, 4-204, 4-274
- MAP DYNAMIC statement, 4-205 to 4-208
  - with REMAP statement, 4-337, 4-339
- MAP statement, 4-201 to 4-204
  - with FIELD statement, 4-109
  - with MAP DYNAMIC statement, 4-207
  - with REMAP statement, 4-337

MAR function, 4–209

Margin  
width, 4–209, 4–210, 4–248, 4–295

MARGIN statement, 4–210  
See also NOMARGIN statement  
with PRINT statement, 4–295

MAT  
with DET function, 4–72

MAT INPUT statement, 4–217 to 4–219

MAT LINPUT statement, 4–220 to 4–222

MAT PRINT statement, 4–223 to 4–225

MAT READ statement, 4–226 to 4–228

Matrix, 1–20  
identity, 4–214

Matrix arithmetic, 4–214

Matrix functions  
DET function, 4–72  
NUM2 function, 4–252  
NUM function, 4–250

Matrix operations  
arithmetic, 4–214  
assigning values, 4–217, 4–220, 4–226  
I/O, 4–250, 4–252  
inversion, 4–72, 4–215  
printing, 4–223  
redimensioning, 4–218, 4–220, 4–223, 4–226  
scalar multiplication, 4–215  
transposition, 4–215

MAT statement, 4–212 to 4–216  
with FIELD statement, 4–109

MAX function, 4–229 to 4–230

Memory  
clearing with SCRATCH command, 2–55

MID\$ function, 4–231  
See also SEG\$ function

MIN function, 4–234 to 4–235

Minus sign (–)  
in PRINT USING statement, 4–300

Mixed-mode expressions, 1–38

MOD function, 4–236 to 4–237

Modifiable parameters, 4–14

Modifiers  
FOR statement, 4–123  
IF statement, 4–155, 4–156  
UNLESS statement, 4–406  
UNTIL statement, 4–410  
WHILE statement, 4–427

Module names, 1–6

MOVE  
FILL item formats and storage, 4–31 to 4–32  
MOVE statement, 4–238 to 4–241

MOVE statement (Cont.)  
with FIELD statement, 4–109

Multi-line  
DEF\* functions, 4–67  
DEF statement, 4–61

Multi-statement lines, 1–8  
backslash in, 1–8  
branching to, 1–9  
execution of, 1–8  
format of, 1–8, 1–9  
implicit continuation, 1–8  
transferring control to, 1–8

## N

---

%NOCROSS directive, 3–18 to 3–19

%NOLIST directive, 3–20

NAME...AS statement, 4–242

Named constants, 1–28  
changing, 1–29  
external, 1–31, 4–103  
internal, 1–29, 4–56

NEW command, 2–38 to 2–39

NEXT statement, 4–244  
with FOR statement, 4–125  
with WHILE statement, 4–426

NOECHO function, 4–246  
See also ECHO function

NOMARGIN statement, 4–248  
See also MARGIN statement

Nonexecutable DIM statement, 4–77

Nonexecutable statements, 1–4, 1–10  
COMMON statement, 4–32  
DATA statement, 4–49  
DECLARE statement, 4–56  
DIM statement, 4–77  
EXTERNAL statement, 4–105  
MAP DYNAMIC statement, 4–207  
MAP statement, 4–203  
REM statement, 4–335  
UNLESS statement, 4–406

Nonmodifiable parameters, 4–14

Nonprinting characters  
processing of, 1–11  
using, 1–11

Nonvirtual DIM statement, 4–77

NOREWIND clause, 4–275

NOSPAN clause, 4–275

Notation  
E, 1–25, 1–25t, 4–295, 4–300, 4–302  
explicit literal, 1–31

## Notation (Cont.)

- exponential, 1–25, 4–295
- NOT operator
  - evaluation of, 1–51
- NUL\$, 4–214
- NUM\$ function, 4–254
- NUM1\$ function, 4–256
- NUM2 function, 4–252
  - after MAT INPUT statement, 4–218
  - after MAT LINPUT statement, 4–221
  - after MAT READ statement, 4–227
- Numbers
  - random, 4–319, 4–357
  - sign of, 4–370
- Number sign (#)
  - in PRINT USING statement, 4–299
- Numbers in E notation, 1–25t
- Numeric constants, 1–24
- Numeric conversion, 4–23
- Numeric expressions, 1–36
  - format of, 1–37
  - promotion rules, 1–38
  - result data types, 1–38
  - results for DECIMAL data, 1–40
  - results for GFLOAT and HFLOAT, 1–39
- Numeric functions, 4–44
  - ABS% function, 4–4
  - ABS function, 4–2
  - DECIMAL function, 4–52
  - FIX function, 4–119
  - INT function, 4–171
  - LOG10 function, 4–193
  - LOG function, 4–191
  - MAG function, 4–197
  - RND function, 4–357
  - SGN function, 4–370
  - SQR function, 4–376
  - SWAP% function, 4–394
- Numeric literal notation, 1–31
- Numeric operator precedence, 1–50
- Numeric precision
  - with PRINT statement, 4–296
  - with PRINT USING statement, 4–299
- Numeric relational expressions
  - evaluation of, 1–42
  - operators, 1–43
- Numeric string functions
  - CHR\$ function, 4–25
  - COMP% function, 4–35
  - DECIMAL function, 4–52
  - DIF\$ function, 4–74

## Numeric string functions (Cont.)

- FORMAT\$ function, 4–127
  - INTEGER function, 4–173
  - NUM\$ function, 4–254
  - NUM1\$ function, 4–256
  - PLACE\$ function, 4–288
  - PROD\$ function, 4–305
  - QUO\$ function, 4–314
  - REAL function, 4–326
  - STR\$ function, 4–382
  - SUM\$ function, 4–392
  - VAL% function, 4–415
  - VAL function, 4–413
- ## Numeric strings
- comparing, 4–35
  - precision, 4–74, 4–288, 4–305, 4–314, 4–392
  - rounding, 4–288, 4–305, 4–314
  - rounding and truncation values, 4–289 to 4–290
  - truncating, 4–288, 4–305, 4–314
- ## NUM function, 4–250
- after MAT INPUT statement, 4–218
  - after MAT LINPUT statement, 4–221
  - after MAT READ statement, 4–227

## O

---

- ## Object libraries
- module names in, 1–6
- ## Object module
- creating, 2–10
  - default name, 2–10
  - loading, 2–35
  - version identification, 3–6
- ## /OBJECT qualifier, 2–16
- ## Octal radix, 1–31
- ## OLD command, 2–40
- with RUN command, 2–49
- ## /OLD\_VERSION=CDD\_ARRAY qualifier, 2–11
- ## ON...GOSUB...OTHERWISE statement, 4–265
- with RETURN statement, 4–350
- ## ON...GOSUB statement, 4–265 to 4–266
- ## ON...GOTO...OTHERWISE statement, 4–267
- ## ON...GOTO statement, 4–267 to 4–268
- ## ON ERROR GO BACK statement, 4–258 to 4–259
- with END statement, 4–87
  - within a handler, 4–259
  - within protected regions, 4–259
- ## ON ERROR GOTO 0 statement, 4–263 to 4–264
- with END statement, 4–87
- ## ON ERROR GOTO statement, 4–260 to 4–262
- with END statement, 4–87



- ON ERROR GOTO statement (Cont.)
  - within a handler, 4-261, 4-264
  - within protected regions, 4-261, 4-264
  - with WHEN blocks, 4-261
- Online documentation, 2-29
- Opening files
  - with USEROPEN clause, 4-280
- OPEN statement, 4-269 to 4-282
  - with STATUS function, 4-378
- Operator precedence, 1-36, 1-50
- Operators
  - arithmetic, 1-36
  - evaluation of, 1-50
  - lexical, 3-4, 3-8, 3-15
  - logical, 1-46
  - numeric operator precedence, 1-50
  - numeric relational, 1-43
  - precedence of, 1-36, 1-50
  - string relational, 1-45
- OPTIONAL
  - with EXTERNAL statement, 4-104
- OPTION statement, 4-283 to 4-287
- ORGANIZATION clause, 4-276
- OTHERWISE clause, 4-265, 4-267
- Output
  - formatting with FORMAT\$ function, 4-127
  - formatting with PRINT USING statement, 4-298 to 4-302
- Output listing
  - cross-reference table, 3-3, 3-18
  - %LIST directive, 3-17
  - %NOLIST directive, 3-20
  - %PAGE directive, 3-21
  - %PRINT directive, 3-22
  - %SBTTL directive, 3-25
  - %TITLE directive, 3-27
- Overflow checking, 4-286
- /OVERFLOW qualifier, 2-16

## P

---

- %PAGE directive, 3-21
- %PRINT directive, 3-22, A-2
- Packed decimal, 1-12
  - See also DECIMAL data type
- Parameter-passing mechanisms
  - DEF\* functions, 4-67
  - DEF statement, 4-61
  - EXTERNAL statement, 4-105
  - FUNCTION statement, 4-135
  - SUB statement, 4-388
- Parameter-passing mechanisms (Cont.)
  - VAX BASIC, 4-11 to 4-13
- Parameters
  - DEF\* functions, 4-66, 4-67
  - DEF statement, 4-60, 4-61
  - EXTERNAL statement, 4-103
  - function, 4-60, 4-66
  - FUNCTION subprograms, 4-134
  - modifiable, 4-14
  - nonmodifiable, 4-14
  - SUB subprograms, 4-386
- Parentheses
  - in array names, 1-19
  - in expressions, 1-36, 1-50
- Passing mechanisms
  - with CALL statement, 4-10
- Percent sign (%)
  - in DATA statement, 1-26, 4-49
  - in DECLARE statement, 4-55
  - in PRINT USING statement, 4-300
  - in variable names, 1-17
  - suffix character, 1-14
- Period (.)
  - in PRINT USING statement, 4-299
  - in variable names, 1-17
- PLACE\$ function, 4-288 to 4-291
  - rounding and truncation values, 4-289 to 4-290
- Plus sign (+)
  - in string concatenation, 1-41
- POS function, 4-292 to 4-293
- Precision
  - in PRINT statement, 4-296
  - in PRINT USING statement, 4-299
  - NUM\$ function, 4-254
  - NUM1\$ function, 4-256
  - of data types, 1-13
  - of numeric strings, 4-74, 4-288, 4-305, 4-314, 4-392
- Predefined constants, 1-34
- PRIMARY KEY clause, 4-277
- PRINT statement, 4-294 to 4-297
  - with TAB function, 4-396
- PRINT USING statement, 4-298 to 4-304
- Print zones
  - in MAT PRINT statement, 4-223
  - in PRINT statement, 4-295
- PROD\$ function, 4-305 to 4-307
  - rounding and truncation values, 4-289 to 4-290
- Program control statements
  - END statement, 4-85
  - EXIT statement, 4-97

## Program control statements (Cont.)

- FOR statement, 4–123
- GOSUB statement, 4–149
- GOTO statement, 4–151
- IF statement, 4–155
- ITERATE statement, 4–175
- ON...GOSUB statement, 4–265
- ON...GOTO statement, 4–267
- RESUME statement, 4–345
- RETURN statement, 4–350
- SELECT statement, 4–365
- SLEEP statement, 4–373
- STOP statement, 4–380
- UNTIL statement, 4–409
- WAIT statement, 4–419
- WHILE statement, 4–426

## Program Design Facility, A–2

## Program documentation, 1–52

## Program elements, 1–1

## Program execution

- continuing, 2–21, 2–49
- initiating with RUN command, 2–48
- stopping, 2–21, 2–49, 4–380
- suspending, 4–373
- waiting for input, 4–419

## Program input

- INPUT LINE statement, 4–166
- INPUT statement, 4–162
- LINPUT, 4–186
- waiting for, 4–419

## Program lines

- automatic sequencing, 2–56
- deleting, 2–22
- displaying, 2–33
- editing, 2–24
- elements of, 1–1
- format of, 1–1
- length of, 1–2
- numbering, 1–2
- order of, 1–10, 2–45
- resequencing, 2–45
- terminating, 1–2, 1–10, 1–11

## Programs

- compiling, 2–10
- continuing, 2–21, 2–49
- debugging, 2–49
- deleting, 2–62
- editing, 2–24
- ending, 4–85
- executing, 2–48
- halting, 2–21, 2–49

## Programs (Cont.)

- merging, 2–5
- naming, 2–38
- renaming, 2–41
- saving, 2–43, 2–51
- stopping, 2–21, 2–49, 4–380
- PROGRAM statement, 1–6, 4–308 to 4–309
- Promotion rules
  - data type, 1–38
  - DECIMAL, 1–39
  - floating-point, 1–38
  - integer, 1–38
- Prompt
  - after STOP statement, 4–380
  - INPUT LINE statement, 4–166
  - INPUT statement, 4–162
  - LINPUT statement, 4–186
  - MAT INPUT statement, 4–218
  - MAT LINPUT statement, 4–221
- PSECT, 4–29, 4–201
- PUT statement, 4–310 to 4–313

## Q

## Qualifiers, 2–10 to 2–40

- /BYTE, 2–13
- /DEBUG, 2–49
- /DECIMAL\_SIZE, 2–14
- /DOUBLE, 1–16, 2–14
- /GFLOAT, 2–15
- /HFLOAT, 2–15
- /INTEGER\_SIZE, 2–11
- /LONG, 2–16
- /[NO]ANALYSIS\_DATA, 2–11
- /[NO]ANSI\_STANDARD, 2–12
- /[NO]AUDIT, 2–13
- /[NO]CHECK, 2–11
- /[NO]CROSS\_REFERENCE, 2–13
- /[NO]DEBUG, 2–14
- /[NO]DEPENDENCY\_DATA, 2–11
- /[NO]DESIGN, 2–11
- /[NO]DIAGNOSTICS, 2–11
- /[NO]FLAG, 2–14
- /[NO]LINES, 1–2, 2–15, 4–89, 4–380
- /[NO]MACHINE\_CODE, 2–16
- /[NO]OBJECT, 2–16
- /[NO]OLD\_VERSION, 2–11
- /[NO]OVERFLOW, 2–16
- /[NO]ROUND, 2–16
- /[NO]SETUP, 2–17
- /[NO]SHOW, 2–18, 3–11

## Qualifiers (Cont.)

- /[NO]SYNTAX\_CHECK, 2–19
- /[NO]TRACEBACK, 2–19
- /[NO]WARNINGS, 2–20
- /REAL\_SIZE, 2–11
- /SCALE, 2–11
- /SINGLE, 2–18
- /TYPE, 1–14
- /TYPE\_DEFAULT, 2–19
- /VARIANT, 2–19, 3–29
- VAX BASIC command, 2–10 to 2–40
- /WORD, 2–20

QUO\$ function, 4–314 to 4–316  
    rounding and truncation values, 4–289 to 4–290

Quotation marks  
    in string literals, 1–27

## R

%REPORT %DEPENDENCY directive, A–2

%REPORT directive, 3–23 to 3–24

Radix–50, 4–317

RAB status, 4–354

RAD\$ function, 4–317 to 4–318

### Radix

- ASCII, 1–31
- binary, 1–31
- decimal, 1–31
- hexadecimal, 1–31
- in explicit literal notation, 1–31
- octal, 1–31

RANDOMIZE statement, 4–319 to 4–320

    See also RND statement

Random numbers, 4–319, 4–357

### Range

- of data types, 1–13
- of subscripts, 1–20

RCTRLC function, 4–321

    See also CTRLC function, 4–321

RCTRLC function, 4–322 to 4–323

READ statement, 4–324 to 4–325

    See also DATA statement  
    with DATA statement, 4–48, 4–49

REAL data type, 1–12

REAL function, 4–326 to 4–327

/REAL\_SIZE qualifier, 2–11

### Record attributes

- MAP clause, 4–274
- RECORDSIZE clause, 4–274, 4–279
- RECORDTYPE clause, 4–278

### Record buffer

### Record buffer (Cont.)

- DATA pointers, 4–343
- MAP DYNAMIC pointers, 4–207, 4–339
- moving data, 4–238
- REMAP pointers, 4–337, 4–339
- setting size, 4–273

RECORD clause, 4–112, 4–140, 4–310, 4–311

Record File Address, 1–12, 4–112, 4–140, 4–147

### RECORD items.

- accessing, 4–330

### Record Management Services

    See RMS

### Record pointers

- after FIND statement, 4–116, 4–117
- after GET statement, 4–144, 4–145
- after PUT statement, 4–311
- after UPDATE statement, 4–411
- REMAP statement, 4–339
- RESTORE statement, 4–344
- WINDOWSIZE clause, 4–281

### Records

- deleting with DELETE statement, 4–70
- deleting with SCRATCH statement, 4–361
- finding RFA of, 4–112, 4–141
- locating by KEY, 4–117, 4–141, 4–145
- locating by RECORD number, 4–140
- locating by RFA, 4–112, 4–117, 4–141, 4–145
- locating randomly, 4–117
- locating sequentially, 4–112, 4–117, 4–140, 4–145
- locating with FIND statement, 4–111
- locating with GET statement, 4–139
- locking, 4–113, 4–114, 4–145, 4–280
- locking with GET statement, 4–141, 4–142
- processing, 4–139, 4–274
- retrieving by KEY, 4–141, 4–145
- retrieving by RECORD number, 4–140
- retrieving by RFA, 4–140, 4–145
- retrieving randomly, 4–145
- retrieving sequentially, 4–140, 4–145
- retrieving with GET statement, 4–139
- size of, 4–310
- stream, 4–277
- unlocking, 4–70, 4–114, 4–129, 4–145, 4–280
- unlocking with UNLOCK statement, 4–407
- writing by RECORD number, 4–310
- writing sequentially, 4–311
- writing with PRINT statement, 4–294
- writing with PUT statement, 4–310
- writing with UPDATE statement, 4–411

RECORDSIZE clause, 4–204, 4–279, 4–310

RECORD statement, 4–328 to 4–332

**RECORD structures**  
     components of, 4-331  
     declaring, 4-331  
**RECORDTYPE clause**, 4-278  
**RECOUNT function**, 4-333 to 4-334  
     after GET statement, 4-145  
     after INPUT LINE statement, 4-168  
     after INPUT statement, 4-164  
     after LINPUT statement, 4-187  
**Recursion**  
     in DEF\* functions, 4-68  
     in DEF functions, 4-62  
     in error handlers, 4-261  
     in subprograms, 4-388  
**Redimensioning arrays**  
     with executable DIM statement, 4-78  
**REGARDLESS clause**  
     with FIND statement, 4-114  
     with GET statement, 4-141  
**Relational expressions**, 1-42  
     compared with logical, 1-46, 1-48  
     definition of, 1-42  
     format of, 1-42  
     in SELECT statement, 4-366  
     numeric, 1-42  
     string, 1-43  
     truth tests, 1-42, 1-44  
**Relational operators**  
     numeric, 1-43  
     string, 1-45  
**Relative files**, 4-277  
     BUCKETSIZE clause, 4-273  
     deleting records in, 4-70  
     finding records in, 4-117  
     opening, 4-276  
     record size in, 4-280  
     retrieving records sequentially in, 4-145  
     updating, 4-412  
     writing records to, 4-311  
**REMAP statement**, 4-337 to 4-341  
     FILL item formats and storage, 4-31 to 4-32  
     with MAP DYNAMIC statement, 4-207  
**REM statement**, 4-335 to 4-336  
     in multi-statement lines, 1-9  
     multi-line format, 1-54, 4-335  
     terminating, 1-55, 4-335  
     transferring control to, 1-55  
**RENAME command**, 2-41 to 2-42  
**REPLACE command**, 2-43 to 2-44  
     with RENAME command, 2-41  
**RESEQUENCE command**, 2-45 to 2-47  
  
**Reserved words**, 1-5  
**RESET statement**, 4-343  
     See also RESTORE statement  
**RESTORE statement**, 4-343 to 4-344  
**Result data types**  
     for DECIMAL data, 1-40  
     GFLOAT and HFLOAT, 1-39  
     mixed-mode expressions, 1-38  
**RESUME statement**, 4-345 to 4-347  
     END statement, 4-87  
     ERL function, 4-89  
     ERN\$ function, 4-91  
     ERR function, 4-93  
     INPUT LINE statement, 4-168  
     INPUT statement, 4-164  
     LINPUT statement, 4-187  
**RETRY statement**, 4-348 to 4-349  
     with FOR...NEXT loops, 4-348  
     with FOR...UNTIL loops, 4-348  
     with FOR...WHILE loops, 4-348  
**RETURN statement**, 4-350 to 4-351  
**RFA clause**, 4-112, 4-140  
**RFA data type**  
     allowable operations, 1-12  
     storage of, 1-12  
**R formatting character**  
     in PRINT USING statement, 4-303  
**RIGHT\$ function**, 4-352 to 4-353  
     See also SEG\$ function  
**Right-justification**  
     PRINT USING statement, 4-303  
     with RSET statement, 4-359  
**RMS (Record Management Services)**  
     accessing records, 4-139  
     deleting records, 4-70  
     locating records, 4-110  
     opening files, 4-269  
     operations, 4-354  
     replacing records, 4-411  
**RMSSTATUS function**, 4-354 to 4-356  
**RND function**, 4-357 to 4-358  
     See also RANDOMIZE statement  
**/ROUND qualifier**, 2-16  
**RSET statement**, 4-359 to 4-360  
**RUN command**, 2-48 to 2-50  
**RUNNH command**, 2-48  
     See also RUN command

# S

- \$ system-command, 2-4
- %SBTTL directive, 3-25 to 3-26
- SAVE command, 2-51 to 2-52
  - with RENAME command, 2-41
- SCALE command, 2-53 to 2-54
- Scale factor
  - setting with OPTION statement, 4-286
  - setting with SCALE command, 2-53
- /SCALE qualifier, 2-11
- SCRATCH command, 2-55
- SCRATCH statement, 4-361 to 4-362
- SEG\$ function, 4-363 to 4-364
- Segmented keys, 4-278
- SELECT
  - transferring control into, 4-265, 4-267
- SELECT statement, 4-365 to 4-367
- Semicolon (:)
  - in INPUT LINE statement, 4-167
  - in INPUT statement, 4-163
  - in LINPUT statement, 4-187
  - in MAT PRINT statement, 4-223
  - in PRINT statement, 4-294
- SEQUENCE command, 2-56 to 2-57
- Sequential files, 4-277
  - deleting records in, 4-361
  - finding records in, 4-117
  - fixed-length, 4-277
  - NOSPAN clause, 4-275
  - opening, 4-276
  - record size in, 4-280
  - retrieving records in, 4-145
  - stream, 4-277
  - updating, 4-412
  - variable-length, 4-277
  - writing records to, 4-294, 4-311
- SET command, 2-58 to 2-59
  - VAX BASIC qualifiers, 2-10 to 2-40
- SET PROMPT statement, 4-368 to 4-369
- /SETUP qualifier, 2-17
- SGN function, 4-370
- SHOW command, 2-60 to 2-61
- /SHOW qualifier, 2-18
  - CDD\_DEFINITIONS, 3-11
  - %INCLUDE directive, 3-11
- Sine, 4-371
- SIN function, 4-371 to 4-372
- SINGLE data type, 1-12
- Single-line
  - DEF\* functions, 4-67
- Single-line (Cont.)
  - DEF statement, 4-61
    - loops, 4-123, 4-409, 4-426
  - /SINGLE qualifier, 2-18
- Single-statement lines, 1-7
- Size
  - of numeric data, 1-13
  - of STRING data, 1-12
- SLEEP statement, 4-373 to 4-374
- Source text
  - copying, 1-9
- SPACE\$ function, 4-375
- SQR function, 4-376 to 4-377
- SQRT function, 4-376
- Square roots, 4-376
- Statement blocks
  - exiting, 4-97
- Statement modifiers
  - FOR statement, 4-123
  - IF statement, 4-155, 4-156
  - UNLESS statement, 4-406
  - UNTIL statement, 4-410
  - WHILE statement, 4-427
- Statements
  - backslash separator, 1-8
  - block, 4-85, 4-98, 4-124, 4-155, 4-329, 4-366
  - components of, 1-4
  - continued, 1-7
  - data typing, 1-15
  - declarative, 4-54
  - empty, 1-53
  - executable, 1-4
  - execution of, 1-8
  - format of, 1-4
  - labeling of, 1-3
  - multi-statement lines, 1-8
  - nonexecutable, 1-4, 1-10, 4-32, 4-49, 4-56, 4-77, 4-105, 4-203, 4-207, 4-335
  - order of, 1-10, 2-45
  - processing of, 1-10
  - single-line, 1-7
- Static
  - arrays, 4-77
  - mapping, 4-201
  - storage, 4-30, 4-201, 4-339
- STATUS function, 4-378 to 4-379
  - VAX BASIC STATUS bits, 4-379
- STEP clause, 4-124
- STOP statement, 4-380 to 4-381
  - See also CONTINUE statement
  - with RUN command, 2-49

## Storage

- allocating for FILL items, 4-239, 4-338
  - allocating for RECORD structures, 4-331
  - allocating for VARIANT fields, 4-331
  - allocating with MAP DYNAMIC statement, 4-205
  - allocating with MAP statement, 4-202
  - allocating with REMAP statement, 4-337
  - COMMON area and MAP area, 4-32, 4-203
  - dynamic, 4-205, 4-337, 4-339
  - for arrays, 4-78
  - for FILL items, 4-31 to 4-32, 4-239
  - for record structures, 4-331
  - in COMMON statement, 4-33
  - in MAP statement, 4-203
  - of data, 1-13
  - of DECIMAL data, 1-12
  - of RFA data, 1-12
  - of STRING data, 1-12
  - shared, 4-29, 4-201
  - static, 4-30, 4-201, 4-339
- STR\$ function, 4-382 to 4-383
- ## Stream
- format, 4-277
  - record, 4-277
- STRING\$ function, 4-384 to 4-385
- ## String arithmetic functions
- DIF\$ function, 4-74
  - PLACE\$ function, 4-288
  - PROD\$ function, 4-305
  - QUO\$ function, 4-314
  - SUM\$ function, 4-392
- ## String constants, 1-27
- processing of, 1-28
- ## String data
- assigning with LSET statement, 4-195
  - assigning with RSET statement, 4-359
- ## STRING data type, 1-12
- length, 1-13
  - storage of, 1-12
- ## String expressions, 1-41
- relational, 1-43
- ## String functions, 4-44
- ASCII function, 4-6
  - EDIT\$ function, 4-83
  - INSTR function, 4-169
  - LEFT\$ function, 4-180
  - LEN function, 4-182
  - MID\$ function, 4-231
  - POS function, 4-292
  - RIGHT\$ function, 4-352
  - SEG\$ function, 4-363

## String functions (Cont.)

- SPACE\$ function, 4-375
  - STRING\$ function, 4-384
  - TRM\$ function, 4-403
  - XLATE\$ function, 4-428
- ## String literals, 1-45
- continuing, 1-7
  - delimiter, 1-27
  - in PRINT USING statement, 4-303
  - processing of, 1-11
  - quotation marks in, 1-27
- ## String relational expressions
- evaluation of, 1-43
  - operators, 1-45
  - padding, 1-44
- ## Strings
- comparing, 1-43, 4-35
  - concatenating, 1-7, 1-36, 1-41
  - converting, 4-22
  - creating, 4-375, 4-384
  - editing, 4-83, 4-403
  - extracting substrings, 4-180, 4-231, 4-352, 4-363
  - finding length, 4-182
  - finding substrings, 4-169, 4-292
  - justifying with FORMAT\$ function, 4-127
  - justifying with LSET statement, 4-195
  - justifying with PRINT USING statement, 4-302
  - justifying with RSET statement, 4-359
  - numeric, 4-35, 4-74, 4-173, 4-288, 4-305, 4-314, 4-326, 4-392, 4-413, 4-415
  - replacing substrings, 4-231
  - suffix character, 1-14
- ## String variables, 1-18
- formatting storage, 4-195, 4-359
  - in INPUT LINE statement, 4-167
  - in INPUT statement, 4-163
  - in LET statement, 4-184
  - in LINPUT statement, 4-187
- ## SUBEND statement, 4-390
- See also END statement
- ## SUBEXIT statement, 4-391
- See also EXIT statement
- ## Subprograms
- calling, 4-10
  - declaring, 4-102
  - ending, 4-85, 4-135, 4-387
  - error handling in, 4-87, 4-98, 4-136, 4-259
  - exiting, 4-97
  - FUNCTION statement, 4-133
  - naming, 4-10, 4-386
  - recursion in, 4-388

- Subprograms (Cont.)
  - returning from, 4-350
  - SUB statement, 4-386
- Subroutines
  - external, 4-103
  - GOSUB statement, 4-149
  - RETURN statement, 4-350
- Subscripted variables, 1-19
  - format of, 1-21
  - range checking, 4-286
  - subscript range, 1-20
- Subscripts, 1-19
  - range of, 1-20
- SUB statement, 4-386 to 4-389
  - parameters, 4-386
  - VAX BASIC parameter-passing mechanisms, 4-11 to 4-13
- Substrings
  - extracting, 4-231, 4-363
  - extracting with LEFT\$ function, 4-180
  - extracting with MID\$ function, 4-231
  - extracting with RIGHT\$ function, 4-352
  - extracting with SEG\$ function, 4-363
  - finding, 4-169, 4-292
  - replacing, 4-231
- Suffix characters
  - integer, 1-14
  - string, 1-14
- SUM\$ function, 4-392 to 4-393
- SWAP% function, 4-394 to 4-395
- /SYNTAX\_CHECK qualifier, 2-19
- SYS\$CURRENCY, 4-302
- SYS\$DIGIT\_SEP, 4-302
- SYS\$RADIX\_POINT, 4-302
- System command, 2-4

## T

---

- %TITLE directive, 3-27 to 3-28
- TAB function, 4-396 to 4-397
- TAN function, 4-398
- Tangent, 4-398
- TEMPORARY clause, 4-280
- Tensor, 1-20
- Terminal
  - printing to, 4-294
- Terminal control functions
  - ECHO function, 4-81
  - NOECHO function, 4-246
  - RCTRL0 function, 4-322
  - TAB function, 4-396

- Terminal-format files, 4-279, 4-280
  - input from, 4-162, 4-166, 4-186, 4-217, 4-220
  - margin, 4-210, 4-248
  - writing records to, 4-223, 4-294
- Text libraries
  - accessing, 3-13
- TIME\$ function, 4-401 to 4-402
- TIME function, 4-399 to 4-400
  - function values, 4-399 to 4-400
- /TRACEBACK qualifier, 2-19
- Trailing minus sign field
  - in PRINT USING statement, 4-300
- Trigonometric functions
  - ATN function, 4-7
  - COS function, 4-39
  - SIN function, 4-371
  - TAN function, 4-398
- TRM\$ function, 4-403
- TRN, 4-215
- Truncation
  - in numeric strings, 4-288, 4-289 to 4-290, 4-305, 4-314
  - in PRINT USING statement, 4-302, 4-303
  - with FIX function, 4-119
- Truth tables, 1-47
- Truth tests
  - in logical expressions, 1-47
  - in relational expressions, 1-42
  - in string relational expressions, 1-44
- /TYPE qualifier, 1-14
- /TYPE\_DEFAULT qualifier, 2-19

## U

---

- UBOUND function, 4-404 to 4-405
- Unconditional branching
  - with GOSUB statement, 4-149
  - with GOTO statement, 4-151
- Unconditional loops, 4-124
- Undefined files, 4-276
- Underscore ( \_ )
  - in PRINT USING statement, 4-301
  - in variable names, 1-17
- UNLESS statement, 4-406
- UNLOCK EXPLICIT clause, 4-113, 4-114, 4-141, 4-280
- UNLOCK statement, 4-407 to 4-408
- UNSAVE command, 2-62
- UNTIL clause, 4-125
- UNTIL loops, 4-244
  - error handling in, 4-346

## UNTIL loops (Cont.)

- exiting, 4-98
- explicit iteration of, 4-175
- transferring control into, 4-149, 4-151, 4-265, 4-267

UNTIL statement, 4-409 to 4-410

UPDATE statement, 4-411 to 4-412

- with UNLOCK statement, 4-407

## Upper bounds

- determining with UBOUND function, 4-404

## Uppercase letters

- in EDIT command, 2-25
- in PRINT USING statement, 4-302
- processing of, 1-11

User-defined functions, 4-59, 4-65

USEROPEN clause, 4-280

# V

---

%VARIANT directive, 3-29 to 3-30

- in %IF directive, 3-8
- in %LET directive, 3-15

VAL% function, 4-415 to 4-416

VAL function, 4-413 to 4-414

## Values

- assigning to array elements, 4-214, 4-217, 4-220, 4-226, 4-325
- assigning to variables, 4-184
- assigning with LET statement, 4-184
- assigning with LINPUT statement, 4-186
- assigning with LSET statement, 4-195
- assigning with MAT INPUT statement, 4-217
- assigning with MAT LINPUT statement, 4-220
- assigning with MAT READ statement, 4-226
- assigning with READ statement, 4-324
- assigning with RSET statement, 4-359
- comparing, 4-115

## Variable names

- in COMMON statement, 4-33
- in MAP DYNAMIC statement, 4-207
- in MAP statement, 4-202
- in REMAP statement, 4-338
- rules for, 1-16

## Variables, 1-16

- assigning values to, 4-162, 4-166, 4-184, 4-186, 4-324
- comparing, 4-115
- declaring, 4-54
- definition of, 1-16
- explicitly declared, 1-19
- external, 4-103

## Variables (Cont.)

- floating-point, 1-18
- implicitly declared, 1-17
- initialization of, 1-22, 4-33, 4-56
- in MOVE statement, 4-238
- in SUB subprograms, 4-388
- integer, 1-18
- lexical, 3-15
- loop, 4-124
- naming, 1-16
- string, 1-18, 4-163, 4-167, 4-184, 4-187
- subscripted, 1-19

VARIANT clause, 4-329

/VARIANT qualifier, 2-19, 3-29

VAX BASIC character set, 1-10

VAX BASIC STATUS bits, 4-379

VAX Language-Sensitive Editor, A-2

VAX Source Code Analyzer, A-2

Vector, 1-20

Version identification, 3-6

## Virtual address

- finding, 4-189

Virtual arrays, 4-56, 4-76, 4-79

- initialization of, 1-23, 4-79

- padding in, 4-79

- with FIELD statement, 4-109

Virtual files, 4-276

- record size, 4-280

- with RESTORE statement, 4-344

VMSSTATUS function, 4-417 to 4-418

# W

---

## WAIT clause

- with GET statement, 4-142
- with INKEY\$ function, 4-158

WAIT statement, 4-419 to 4-420

/WARNINGS qualifier, 2-20

## WHEN blocks

- with GOSUB statement, 4-149
- with GOTO statement, 4-151

## WHEN ERROR constructs

- with DEF\* functions, 4-68
- with DEF functions, 4-62

WHEN ERROR statement, 4-421 to 4-425

- with a detached handler, 4-422
- with an attached handler, 4-422

WHILE clause, 4-125

WHILE loops, 4-244

- error handling in, 4-346
- exiting, 4-98



## WHILE loops (Cont.)

- explicit iteration of, 4-175
- transferring control into, 4-149, 4-151, 4-265, 4-267

WHILE statement, 4-426 to 4-427

### Width

- margin, 4-209, 4-210, 4-248

WINDOWSIZE clause, 4-281

WORD data type, 1-11

/WORD qualifier, 2-20

## X

---

XLATE\$ function, 4-428 to 4-429

## Y

---

! your-comment, 2-2

## Z

---

ZER function, 4-214

### Zero

array element, 1-20, 4-78, 4-215, 4-218, 4-221, 4-224, 4-227, 4-240

blank-if-zero field, 4-300

in PRINT USING statement, 4-300

### Zero-fill field

in PRINT USING statement, 4-300



# How to Order Additional Documentation

---

## Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

## Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

## Telephone and Direct Mail Orders

<b>Your Location</b>	<b>Call</b>	<b>Contact</b>
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	_____	Local Digital subsidiary or approved distributor
Internal <sup>1</sup>	_____	USASSB Order Processing - WMO/E15 <i>or</i> U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

---

<sup>1</sup>For internal orders, you must submit an Internal Software Order Form (EN-01740-07).



# Reader's Comments

VAX BASIC Reference Manual  
AA-HY16B-TE

---

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

<b>I rate this manual's:</b>	<b>Excellent</b>	<b>Good</b>	<b>Fair</b>	<b>Poor</b>
Accuracy (software works as manual says)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Completeness (enough information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Clarity (easy to understand)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization (structure of subject matter)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Figures (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Examples (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Index (ability to find topic)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Page layout (easy to find information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

I would like to see more/less \_\_\_\_\_

What I like best about this manual is \_\_\_\_\_

What I like least about this manual is \_\_\_\_\_

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

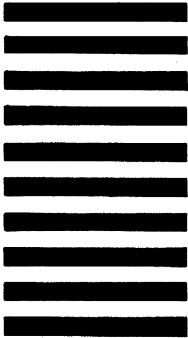
I am using **Version** \_\_\_\_\_ of the software this manual describes.  
Name/Title \_\_\_\_\_ Dept. \_\_\_\_\_  
Company \_\_\_\_\_ Date \_\_\_\_\_  
Mailing Address \_\_\_\_\_  
Phone \_\_\_\_\_

Do Not Tear - Fold Here and Tape

**digital**™



No Postage  
Necessary  
if Mailed  
in the  
United States



**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO. 33 MAYNARD MASS:

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION  
Corporate User Publications—Spit Brook  
ZK01-3/J35  
110 SPIT BROOK ROAD  
NASHUA, NH 03062-9987



Do Not Tear - Fold Here

Cut Along Dotted Line

# Reader's Comments

VAX BASIC Reference Manual  
AA-HY16B-TE

---

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Completeness (enough information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Clarity (easy to understand)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization (structure of subject matter)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Figures (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Examples (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Index (ability to find topic)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Page layout (easy to find information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

I would like to see more/less \_\_\_\_\_  
\_\_\_\_\_

What I like best about this manual is \_\_\_\_\_  
\_\_\_\_\_

What I like least about this manual is \_\_\_\_\_  
\_\_\_\_\_

I found the following errors in this manual:

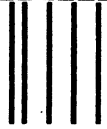
Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

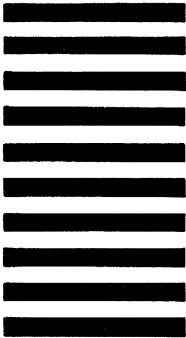
I am using **Version** \_\_\_\_\_ of the software this manual describes.  
Name/Title \_\_\_\_\_ Dept. \_\_\_\_\_  
Company \_\_\_\_\_ Date \_\_\_\_\_  
Mailing Address \_\_\_\_\_  
Phone \_\_\_\_\_

Do Not Tear - Fold Here and Tape

**digital**™



No Postage  
Necessary  
if Mailed  
in the  
United States



**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION  
Corporate User Publications—Spit Brook  
ZK01-3/J35  
110 SPIT BROOK ROAD  
NASHUA, NH 03062-9987



Do Not Tear - Fold Here

Cut Along Dotted Line