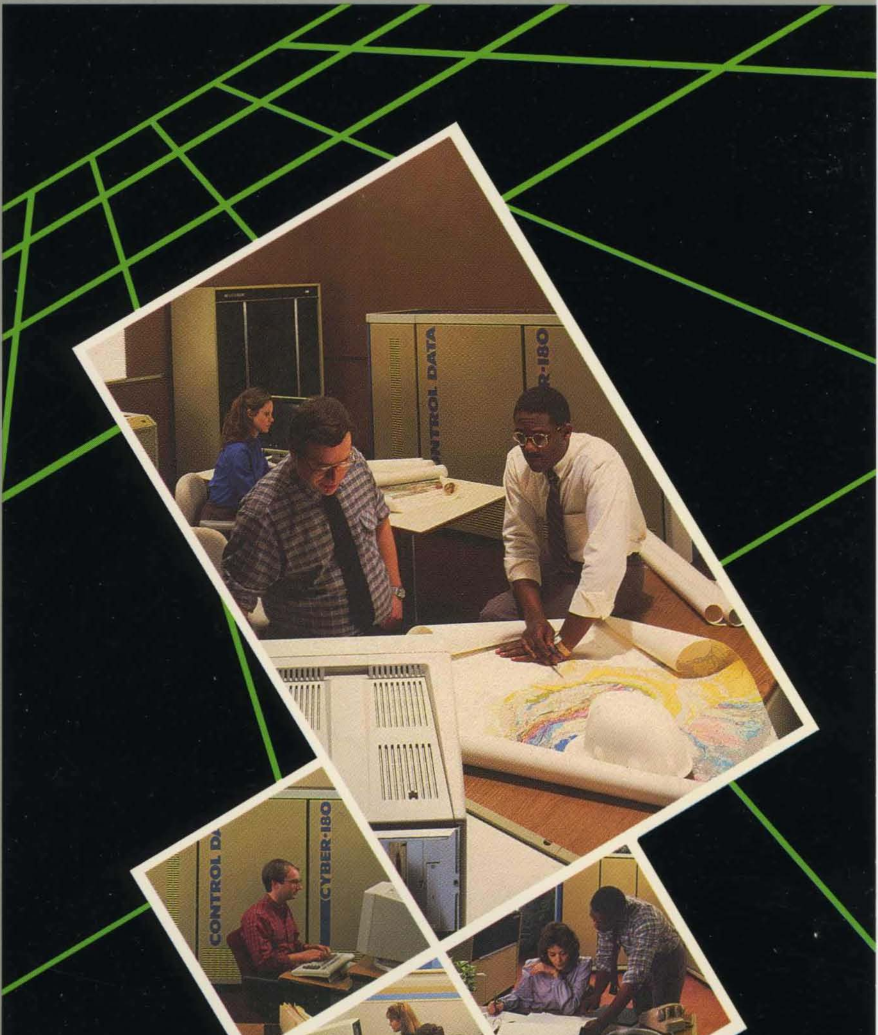
BASIC for NOS/VE

CONTROL DATA



Usage

60486313

BASIC for NOS/VE

Usage

Publication Number 60486313

Related Manuals

The following table lists all manuals that are referenced in this manual or that contain background information:

Manual Title	Publication Number	Online Title
BASIC Manuals:		
BASIC for NOS/VE Usage	60486313	BASIC
BASIC Summary	60486319	
NOS/VE Manuals:		
NOS/VE System Usage	60464014	
NOS/VE Commands and Functions	60464018	SCL
NOS/VE Source Code Management Usage	60464313	
NOS/VE Object Code Management Usage	60464413	
Additional References:		
Debug for NOS/VE Usage	60488213	
Debug for NOS/VE Quick Reference	L60488218	DEBUG
NOS/VE Diagnostic Messages	60464613	MESSAGES
FORTRAN Version 1 for NOS/VE Language Definition Usage	60485913	
CYBIL for NOS/VE System Interface Usage	60464115	
NOS/VE Accounting Analysis System Usage	60463923	

Manual History

Revision	System Version/ PSR level	Product Version	Date
A	1.1.3/644	1.0	October 1985
B	1.1.4/649	1.1	January 1986
C	1.2.1/664	1.2	July 1986
D	1.2.2/678	1.2	April 1987
E	1.3.1/700	1.3	April 1988

This revision:

This manual is revision E printed in April 1988. It documents BASIC for NOS/VE at release level 1.3.1 and at PSR level 700.

One new feature is documented in this revision:

- Unit-measured application accounting using the BCPDAUA subroutine. For a description, see the section titled Unit-measured Application Accounting in chapter 9, Subroutines.

Changed features documented in this revision are the following:

- You can specify permanent file paths in BASIC programs. For a description, see the section titled OPEN Statement in chapter 13, Files, or the section titled Program Execution in chapter 14, Compilation and Execution.

- Loader errors are automatically sent to the standard file $ERRORS. For a description, see the section titled Program Execution in chapter 14, Compilation and Execution.

- The STOP statement no longer automatically invokes the Debug utility. You must now invoke the Debug utility before executing a BASIC program. For a description, see the section titled STOP Statement in chapter 6, Runtime Error Processing.

In addition, the revision includes several new glossary entries and miscellaneous technical and editing corrections.

Contents

About This Manual

About This Manual

This manual describes the the CONTROL DATA® Network Operating System/Virtual Environment (NOS/VE) BASIC language. NOS/VE BASIC was designed to permit easy migration from popular microcomputers to CDC® CYBER 180 computer systems. NOS/VE BASIC conforms to the ANSI standard for minimal BASIC, ANS X3.60-1978, approved January 17, 1978. NOS/VE BASIC does not conform to the new ANSI standard for full BASIC, ANS X3.113-1987, approved January 28, 1987.

Audience

This manual describes the features of NOS/VE BASIC. It assumes that you understand NOS/VE and SCL concepts as presented in the NOS/VE System Usage manual. We expect the audience to form a varied group in terms of programming experience and areas of application. For this reason, the manual is written to accommodate both experienced programmers and casual users.

Organization

This manual is organized by topic into the following chapters:

Chapter 1 presents a brief introduction to NOS/VE BASIC.

Chapters 2 through 7 and 9 through 13 describe the BASIC language specifications. Chapter 2 describes the rules for organizing BASIC statements into executable programs. Chapter 3 describes the fundamental elements used in writing BASIC statements. Chapters 4 through 7 and 9 through 13 describe all of the BASIC statements.

Chapter 8 describes the CDC-supplied functions that allow you to take advantage of certain operating system capabilities.

Chapter 14 describes the system commands used to compile and execute a NOS/VE BASIC program. Descriptions of all parameters and options are included.

Following chapter 14 is a set of appendixes that provide the following supplementary information:

Glossary of terms used in this manual.

Description of the ASCII character set.

Listing of compile-time diagnostics.

Index of library functions.

An introduction to the Debug utility.

Conventions

Certain notational conventions are used throughout this manual with consistent meaning. These conventions are as follows:

UPPERCASE	In statement syntax, an item appearing in all uppercase letters indicates a keyword or character that must be written as shown. Although lowercase letters are interpreted the same as uppercase letters when used in BASIC keywords and symbols, uppercase is used in this manual for consistency.
lowercase	In statement syntax, an item that contains lowercase letters indicates a name, number, or symbol that you must supply. However, to enhance readability, these items are shown in uppercase when occurring in text.
blue	Denotes user examples.
numbers	All numbers in this manual are base 10 unless otherwise noted.
...	In statement syntax, a horizontal ellipsis indicates that a series of similar objects are to be supplied.
. . .	In program examples, a vertical ellipsis (2 or 3 periods) indicates that other BASIC statements or parts of the program have not been shown because they are not relevant to the example.
|	Vertical bars in the margin indicate changes or additions to the text from the previous revision.
●	A dot next to the page number indicates that a significant amount of text (or the entire page) has changed from the previous revision.
spaces	Whenever a space appears in a BASIC statement, any number of spaces can be used. In this manual, extra spaces are used in format descriptions to improve readability.

Ordering Manuals

Control Data manuals are available from your local Control Data sales office. Sites within the U.S. can also order manuals directly from Control Data Literature and Distribution Services at the following address:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

When ordering a manual, please specify the complete manual title and publication number. For example, if you are ordering this manual, specify BASIC for NOS/VE Usage, 60486313.

Submitting Comments

The last page of this manual is a comment sheet. Please use it to give us your opinion of the manual's usability, to suggest specific improvements, and to report technical or typographical errors. If the comment sheet has already been used, you can mail your comments to us at the following address:

Control Data Corporation
Technology and Publications Division
P.O. Box 3492
Sunnyvale, CA 94088-3492

Be sure to include the following information with your comments:

The manual title and publication number (for this manual, BASIC for NOS/VE Usage, 60486313).

The revision letter from the Manual History page indicating the current revision of the manual.

Your name, your company's name and address, your work phone number, and whether you want a reply.

If you have access to SOLVER, the CDC online facility for reporting problems, you can use it to submit comments about this manual. When SOLVER prompts you for the product identifier for your report, please specify BC8 for the BASIC documentation.

In Case of Trouble

Control Data´s CYBER Software Support maintains a hotline to assist you if you have trouble using our products. If you need help beyond that provided in the documentation or find that the product does not perform as described, call us at one of the following numbers and a support analyst will work with you.

From the USA and Canada: (800) 345-9903

From other countries: (612) 851-4131

The preceding numbers are for help on product usage. Address questions about the physical packaging and/or distribution of printed manuals to Literature and Distribution Services at the following address:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

or you can call (612) 292-2101. If you are a Control Data employee, call CONTROLNET® 243-2100 or (612) 292-2100.

Introduction to NOS/VE BASIC 1

This chapter presents a brief introduction to NOS/VE BASIC.

Introduction to NOS/VE BASIC 1

BASIC is an all-purpose programming language that is well-suited for scientific, business, and educational applications.

Features

NOS/VE BASIC is designed to permit easy migration, both of programs and of people, from popular microcomputers to CDC CYBER 180 computer systems and to provide a language that is easy to use, especially for casual users.

In addition, NOS/VE BASIC provides an interface to NOS/VE FORTRAN. This adds power and flexibility to the language by increasing the number of applications that can be readily accessed.

NOS/VE BASIC offers a wide range of capabilities. These capabilities include integer and real arithmetic, block control structures, character string processing, and numerous input/output capabilities.

The NOS/VE BASIC Compiler

The NOS/VE BASIC compiler reads a file containing the NOS/VE BASIC source program, translates that program into an object program consisting of machine instructions, and (optionally) writes the object program to a file. The object program can then be loaded into memory and executed by system commands.

A NOS/VE BASIC source program consists of text lines formatted according to the rules of NOS/VE BASIC syntax. If the compiler detects a syntax error in the source program, it issues a descriptive message describing the nature of the error. The compile-time diagnostics are always written to the list file. The compiler detects errors at different levels of severity. If the errors are severe enough (fatal), the resulting object program cannot be executed; you must correct the errors and recompile. Generally, the diagnostic messages provide enough information to enable you to easily determine the cause of the errors.

In addition to the object program, the NOS/VE BASIC compiler produces an output listing file. This file is optional and is selected by parameters on the BASIC command. The output listing file contains a complete listing of the source program and, optionally, an object listing and a reference map. The reference map provides detailed information about symbolic names and other items used in the NOS/VE BASIC program and is an extremely useful debugging tool.

The NOS/VE BASIC compiler provides a number of other options in addition to those described above. The available compiler options, the formats of the input and output files, and the commands for compiling and executing a NOS/VE BASIC program, are described in chapter 14 of this manual.

The NOS/VE Environment

The NOS/VE operating system provides several software facilities you can use to make the process of creating and maintaining NOS/VE BASIC programs easier and more efficient. These facilities include:

Source Code Utility (SCU)

> Allows you to create and maintain source programs. SCU is especially useful for creating and updating large collections of source programs called source libraries.

Object Code Management utilities (OCM)

> Enables you to create and maintain libraries of compiled object programs (called object libraries). Object libraries are especially useful for programs that are to be shared by other programs. OCM also provides a facility for measuring and analyzing program performance characteristics.

Debug utility

> Enables you to debug a program during execution. You can stop the program at selected points or on the occurrence of an error and request formatted displays of variables and arrays. The Debug utility is described in appendix E of this manual.

NOS/VE BASIC provides an interface to FORTRAN and COBOL using subprogram calls. These subprogram calls can also be used to access other subroutines that conform to the FORTRAN calling sequence. The calls are described in chapter 9 of this manual.

You can also execute NOS/VE System Command Language (SCL) commands from within a NOS/VE BASIC program. The system interface statements are described in chapter 5 of this manual.

Program Structure 2

This chapter describes the fundamental units that are used to form BASIC programs.

Program Structure 2

A program consists of a main program, and zero or more subprograms.

The main program is the only procedure within a program that can be executed by itself. Every program must have a main program.

A subprogram is a procedure that accomplishes a set of tasks for the main program. A subprogram can be compiled by itself, but cannot be executed by itself. A program need not have any subprograms.

This chapter is an overview of the structure of a NOS/VE BASIC program. It describes the components from which a program is built.

A NOS/VE BASIC program contains routines, blocks, lines, statements, and identifiers. These key terms are defined in this chapter and appear throughout the manual.

Routines

A NOS/VE BASIC program is a collection of one or more external routines, one of which is a main program.

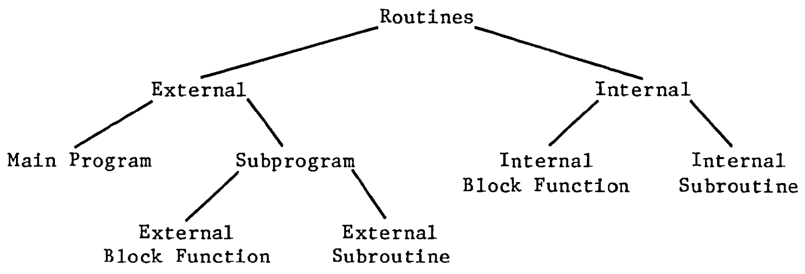
In this manual, the generic term routine applies to main programs, block functions (specified by FUNCTION statements), and subroutines (specified by SUB statements).

Routines are classified as either external or internal.

An external routine is either a main program or a subprogram. A subprogram is either an external block function or an external subroutine. Note that the term subprogram refers to an external routine that is not a main program.

An internal routine is either an internal block function or an internal subroutine.

The following tree diagram illustrates the relationships among the terms just described:



A subprogram begins with a subprogram specification statement (EXTERNAL FUNCTION or EXTERNAL SUB), and ends with the corresponding closing statement (END FUNCTION or END SUB). The compiler needs these statements to determine which external routines are subprograms.

A main program is not explicitly specified. After determining which external routines are subprograms, the compiler determines the main program by default. A main program that is followed by a subprogram must end with an END PROGRAM statement.

An internal routine begins with a specification statement (FUNCTION or SUB), and ends with the corresponding closing statement (END FUNCTION or END SUB). A routine is declared to be internal by default when the keyword EXTERNAL does not appear in its specification statement.

If an external routine is followed by a subprogram, the closing statement (END PROGRAM, END FUNCTION, or END SUB) of the external routine must be immediately followed by the subprogram specification statement.

If any lines intervene, even blank lines or comments, the compiler interprets them as a main program. This might cause the compiler to interpret your program as if it contains several main programs. If this happens, a warning is issued. Usually, other errors result from this interpretation.

An external routine:

- Can be compiled as a separate program unit.
- Cannot be contained within another external routine, but can contain embedded internal routines.
- Shares data with other external routines through the COMMON statement or the passing of parameters.

An internal routine:

- Cannot be compiled as a separate program unit.
- Must be contained within a host external routine, but cannot contain embedded routines.
- Has access to all the data of its host external routine.

A declarative statement provides information about how data is to be processed.

The BASIC declarative statements are: COMMON, DECLARE FUNCTION, DECLARE SUB, DEF, DEFDBL, DEFINT, DEFSNG, DEFSTR, EXTERNAL FUNCTION, EXTERNAL SUB, and OPTION BASE.

The declarative statements in an external routine do not apply to other external routines. They do apply to all embedded internal routines.

The declarative statements within an internal routine also apply to the host external routine.

Blocks and Lines

A routine can be thought of as a collection of blocks.

A block is a group of logically or physically related statements or lines. For example, internal routines and looping structures are blocks.

Small blocks are placed inside of larger blocks to build more complex structures. A block IF construction is an example of a block that contains other blocks.

From a global perspective, external routines are blocks. At the local level, even single unstructured BASIC statements qualify as blocks.

A NOS/VE BASIC line can contain at most 255 characters and spaces. It consists of:

- An optional label, provided by the programmer.
- An optional series of one or more BASIC statements that are separated by colons.
- An optional tail comment.

Line Format:

```
    label  statement1 : statement2 : ... : statementN  ´ comment
```

A blank line within a routine is permitted since all three line components are optional. However, do not use blank lines between external routines. Such lines cause the compiler to interpret what follows as a main program.

You can begin each program line with a positive integer of at most six digits. This integer is called a label. A label can be used to reference a line during program execution. Leading zeros in a label are insignificant.

Each label in an external routine must be greater than the preceding label within that routine.

A label is required for any line that is the destination of a branch via a GOSUB, GOTO, ON-GOSUB, ON-GOTO, or RESUME statement.

A label is also used in conjunction with the RESTORE statement, which sets the pointer for an interior data set.

Every statement in a labeled line is associated with the label of the host line.

A statement in an unlabeled host line is associated with the label of the nearest labeled line that precedes the host line. If the host line precedes all the labeled lines of an external routine, the statement is associated with the default value 0.

Examples

The following statements associate the host line with the nearest labeled line.

```
  Statement A : Statement B : Statement C
1 Statement D : Statement E
  Statement F : Statement G
2 Statement H : Statement I : Statement J
```

Statements A, B, and C are associated with the default value 0.

Statements D, E, F, and G are associated with the label 1.

Statements H, I, and J are associated with the label 2.

The time during which a program is being executed is called runtime. Diagnostic messages that result from runtime errors specify error location in terms of associated labels or the default value 0.

Note that 0 is not a valid label. If a BASIC statement references a label, that label must be positive. The default value 0 is associated with a statement only in the example described previously.

BASIC labels are not the same as BASIC line numbers.

The programmer supplies BASIC labels as addresses for lines that are referenced from within a program during program execution.

The compiler assigns each program line a BASIC line number (sometimes called a compiler sequence number) to denote the physical position of the line within a program. The program cannot use a line number to reference a line during program execution.

The time during which a program is compiling is called compile-time. Diagnostics that result from compile-time errors specify error location in terms of BASIC line numbers. Line numbers are also used in conjunction with the Debug utility.

This distinction between labels and line numbers is carefully adhered to throughout this manual.

After the optional label, a line contains an optional series of one or more BASIC statements. Statements are separated by a colons or an end of a line.

Note that a single line can contain more than one BASIC statement, however, BASIC does not allow line continuation. A continuation line is a source line that contains a continuation of the statement that appears in the previous source line.

The end of a line delimits the last statement in the line. If a colon is the last nonblank character in a line, the compiler acts as if the last statement is an empty statement.

Consecutive colons in a line are also permitted (think of them as delimiters for an empty statement).

You can explain the purpose of a line by ending it with a tail comment. This comment is ignored by the compiler.

The apostrophe, whenever it appears outside a quoted string, marks the end of the significant portion of a line.

Examples

- In the following examples the apostrophe separates the last BASIC statement of a line from a tail comment, and tells the compiler to ignore what follows.

```
´ This entire line is a tail comment.

100 READ MIN,MAX ´ set lower/upper bounds

500 LET COUNT% = 0 : STEP% = 1 ´ Initialize Counters

IF A$ = "YES" THEN GOSUB 1000 ´ Handle Information
```

Statements and Identifiers

A statement is an optional series of tokens.

A token is a set of characters that the compiler recognizes as identifying a single entity or word. Constants, identifiers, and special characters are tokens. Frequently, spaces are used to designate the beginning and the end of a token.

Any number of spaces can be used between tokens. Spaces inside a string constant are not treated as token separators because a string constant is itself a token. The compiler uses punctuation and context to properly interpret such spaces.

An identifier is a token that names a program component, or specifies some action or attribute within a program. BASIC identifiers are used to name variables, functions, and subroutines.

A keyword is an identifier that has a preassigned meaning when it is used in a specific context. Keywords appear in BASIC statements, and as names for library functions and supplied string variables.

A NOS/VE BASIC identifier is either a plain name, or a name whose last character is a symbol that specifies data type. An identifier can be no more than 31 characters long, including any type specification symbol.

A plain name consists of a letter followed by a series of letters, digits, and periods. A period is the only special character allowed in a plain name.

An identifier cannot contain a space because a space would effectively split the identifier into two tokens. However, periods can be used to make a name more readable.

Examples

- The following are examples of plain names.

```
X                        BUBBLESORT

TEMP1                    ROW.TOTAL

Y.1985.NET.PROFIT        GAMMA.FUNCTION
```

Identifiers with type specification symbols are discussed in chapter 3 of this manual. For now, note that such names consist of a plain name of at most 30 characters, followed by one of the four type specification symbols: % , ! , # , $.

Reserved Words

A reserved word is a keyword that is reserved exclusively for program or system use. You are not permitted to use such an identifier for your own purpose. Not all keywords are reserved. The following list contains the reserved words for NOS/VE BASIC:

AND	DATE$	EQV	IF	NOT	RESTORE	THEN
APPEND	DECLARE	ERASE	IMP	ON	RESUME	TIME$
AS	DEF	ERROR	INPUT	OPEN	RETURN	TO
BASE	DEFDBL	EXIT	LBOUND	OPTION	RSET	UBOUND
BEEP	DEFINT	EXTERNAL	LEN	OR	RUN	USING
CALL	DEFSNG	FIELD	LET	OUTPUT	SCL	WEND
CALLX	DEFSTR	FOR	LINE	PRINT	SPC	WHILE
CHAIN†	DIM	FUNCTION	LPRINT	PROGRAM	STEP	WIDTH
CLEAR	ELSE	GET	LSET	PUT	STOP	WRITE
CLOSE	ELSEIF	GO	MID$	RANDOMIZE	SUB	XOR
COMMON	END	GOSUB	MOD	READ	SWAP	
DATA	ENDIF	GOTO	NEXT	REM	TAB	

†Has been reserved for possible future use.

The names of most library routines are not reserved. If you use these identifiers to name objects within your program, the following rules apply:

In each external routine:

- If the first use of an identifier is consistent with the format for referencing a library routine, then the identifier is always interpreted as a library routine name.
- If the first use of an identifier is inconsistent with the format for referencing a library routine, then the identifier is always interpreted as the name of an object within your program.
- Each use of an identifier must be consistent with the first use or a compile-time error results.

Examples

The first use of the identifier ASC (in the line labeled 10) is consistent with a reference to the ASC library function. Hence, this identifier is interpreted as a library function reference. The second use (in the line labeled 20) is inconsistent with this interpretation. A compile-time error results.

```
10 L = ASC(S$)
20 FUNCTION ASC(S$)
      ASC = LEN(S$) + 5
   END FUNCTION
```

This program fragment would be legal if the first line were made the last line. In this case, the identifier ASC would refer to the user-defined ASC function.

BASIC Character Set

Only a subset of the standard ASCII character set is actually used to form NOS/VE BASIC statements. The table that follows lists the NOS/VE BASIC characters along with their primary functions or areas of usage.

Characters	Primary Functions or Areas of Usage
Uppercase Letters A through Z	Identifiers, String Constants.
Lowercase Letters a through z	Identifiers, String Constants.
Digits 0 through 9	Identifiers, Labels, Numbers.
' (apostrophe)	Commentary.
:	Delimiter, Substring/Dimension Format.
( ) " space	Delimiters.
, ;	Delimiters, Output Format.
%	Data Type, Output Format Overflow.
$!	Data Type, Output Format.
#	Data Type, Output Format, Channels.
_ (underscore)	Output Format.
&	Number Base, Output Format.
.	Identifiers, Numbers, Output Format.
+ - * ^ \	Arithmetic Operators, Output Format.
/	Arithmetic Operator.
=	Assignment, Relational Operator.
< >	Relational Operators.

BASIC Character Set

With one exception, you can use whatever letter case you want. For example, the following statements have all the same meaning to the compiler.

```
LET A = 5

Let A = 5

let a = 5
```

The BASIC compiler produces a listing of your program exactly as typed. However, it internally translates all lowercase letters outside of quoted strings and DATA statements to uppercase. Thus, identifiers used in diagnostics are displayed in uppercase no matter how they appear in your program listing.

Letter case is significant only when writing quoted or unquoted string constants. Here, the compiler does not make the lowercase-to-uppercase conversion. The constant is stored exactly as typed.

Examples

The two string constants below are not equivalent.

```
"BASIC Statements"

"Basic Statements"
```

They also have different meanings (BASIC, the acronym for Beginner's All-purpose Symbolic Instruction Code, versus Basic, as in Fundamental).

Termination Statements

If a main program is followed by a subprogram, the main program must end with an END PROGRAM statement. This statement has the format:

```
END PROGRAM
```

The END PROGRAM statement must be the last statement in the last line of the main program. Execution of this statement terminates the program.

The specification statement of any subsequent subprogram must immediately follow the END PROGRAM statement. No lines can intervene, not even blank lines or comments.

The END PROGRAM statement is optional if the main program is the last or only external routine in a program. If the last statement of the main program does not transfer control, the program is terminated.

The END and REM statements are also fundamental to a program.

The END statement terminates program execution. When this statement is executed, any open files are closed, and control is returned to system command level. Any number of END statements can appear in an external routine.

Examples

Depending on the value of A, the following program fragment can terminate at the statement labeled 40 or the statement labeled 60.

```
10  IF A = 2 THEN 50
20  LET A = A + 2
30  PRINT A
40  END
50  PRINT A
60  END
```

The REM statement, in addition to a tail comment, provides a way to include comments (remarks) within a program. If the keyword REM begins a BASIC statement, the compiler ignores the rest of the line and continues with the next one.

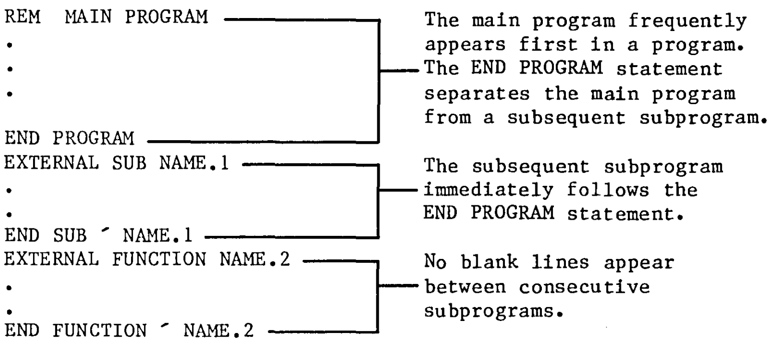
Summary and Sample

A NOS/VE BASIC program is a collection of one or more external routines, one of which is a main program.

Structurally, each external routine is a collection of blocks, some of which might be internal routines. Each block is a group of related lines. Each line is a series of BASIC statements. Each statement is a sequence of tokens.

If an external routine is followed by a subprogram, the closing statement (END FUNCTION, END PROGRAM, or END SUB) of the external routine must be immediately followed by the subprogram specification statement. If any lines intervene, even blank lines or comments, the compiler interprets them as a main program.

The following illustrates general program structure and the use of the END PROGRAM statement:



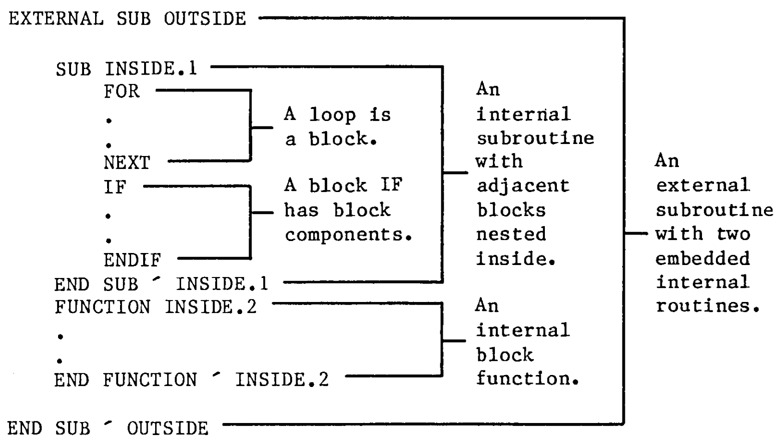
```
REM  MAIN PROGRAM
•
•
•

END PROGRAM
EXTERNAL SUB NAME.1
•
•
END SUB ' NAME.1
EXTERNAL FUNCTION NAME.2
•
•
END FUNCTION ' NAME.2
```

The main program frequently appears first in a program. The END PROGRAM statement separates the main program from a subsequent subprogram.

The subsequent subprogram immediately follows the END PROGRAM statement.

No blank lines appear between consecutive subprograms.

The following illustrates the block structure of an external routine:



```
EXTERNAL SUB OUTSIDE

    SUB INSIDE.1
        FOR
        .
        .
        NEXT
        IF
        .
        .
        ENDIF
    END SUB ' INSIDE.1
    FUNCTION INSIDE.2
    .
    .
    END FUNCTION ' INSIDE.2

END SUB ' OUTSIDE
```

A loop is a block.
A block IF has block components.
An internal subroutine with adjacent blocks nested inside.
An internal block function.
An external subroutine with two embedded internal routines.

Summary and Sample

The following program illustrates the END and REM statements:

```
      REM  This program computes factorials of integers
      REM  between 1 and 20, inclusive.  The number whose
      REM  factorial is computed is denoted by N.
      '-------------------------------------------------------
   10 PRINT "ENTER A POSITIVE INTEGER NO LARGER THAN 20."
      INPUT N
      REM ----------- CHECK FOR ACCEPTABLE INPUT ------------
      IF N <> INT(N) THEN PRINT "ENTER AN INTEGER" : GO TO 10
      IF N < 1 OR 20 < N THEN PRINT "OUT OF RANGE" : GO TO 10
      REM ----- N FACTORIAL IS STORED UNDER THE NAME F ------
      LET F = 1 ' INITIALIZATION
      FOR I = 1 TO N
          LET F = F*I
      NEXT I
      PRINT N; " FACTORIAL IS "; F
      END
```

Language Fundamentals 3

This chapter describes the elements used to write BASIC statements.

Language Fundamentals 3

Constants and variables are the elements of a programming language. They are the data objects that a program processes.

This chapter describes NOS/VE BASIC constants and variables. It also discusses how data types are associated with identifiers.

Constants

A constant is a value that must remain fixed during program execution.

In NOS/VE BASIC, there are two kinds of numeric constants, integer and real, and two kinds of string (or character) constants, quoted and unquoted. BASIC does not allow named constants.

Integer Constants

An integer constant is a signed whole number written without a decimal point. Leading zero digits are ignored. The plus sign for positive integers is optional. You cannot use commas to group the digits of a numeric constant.

Examples

- The following are examples of integer constants.

```
+275210

    44

    07

 -8396
```

The magnitude of an integer constant must be less than 2^63 (approximately 9.2*10^18) or a compile-time error results.

(The circumflex (^) used above denotes the exponentiation operator.)

In addition to the usual decimal form for integers, you can express integers in hexadecimal (base 16) and octal (base 8) form.

A hexadecimal integer constant is expressed by typing an ampersand, followed by an H, followed by one or more hexadecimal digits (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F).

Similarly, an octal integer constant is expressed by typing an ampersand, optionally followied by O, followed by one or more octal digits (0,1,2,3,4,5,6,7).

Notice that no spaces are used in either format.

Examples

- The following are examples of hexadecimal and octal integer constants.

```
&H21D  (equivalent to  541 in base 10)

&H2C   (equivalent to   44 in base 10)

&O54   (equivalent to   44 in base 10)

&273   (equivalent to  187 in base 10)
```

Real Constants

A NOS/VE BASIC real constant is one of the following:

- A number written in decimal (fixed point) format, optionally followed by either an exclamation point (!) or a number sign (#).
- An integer followed by either an exclamation point or a number sign.
- A number written in exponential (floating point) format.

The magnitude of a real constant must be less than 2^4095 (approximately 5.2*10^1232) or a compile-time error results.

Examples

- The following are examples of valid NOS/VE BASIC real constants.

```
-9.86!      -0.0003     44.     44!

-9.86#       -.0003     +44.0   44#
```

The real constant (r * 10^s) can be written in exponential format using one of two forms:

```
rEs
rDs
```

r	Integer or real number.
e or E	Exponent.
d or D	Exponent.
s	Integer.

The D notation is provided for compatibility with popular microcomputer versions of BASIC. BASIC treats the E and D notations identically; double-precision data types are not supported. For information about double-precision data and BASIC, see the section titled Double-Precision Vestiges later in this chapter.

Note that no spaces are used in exponential format.

Examples The following are examples of real constants written in

56.0E+4	(equivalent to 560000.0)
7.3218E-2	(equivalent to 0.073218)
-3.79651D3	(equivalent to -3796.51)
8D5	(equivalent to 800000.0)

Quoted String Constants

A quoted string constant is a sequence of characters or spaces that is enclosed by quotation marks. These outside marks delimit the string constant but are not part of it. A quotation mark embedded within a quoted string constant is denoted by two successive quotation marks.

The length of a quoted string constant is limited only by the number of characters that can fit on a line.

Examples

- This quoted string constant uses several special characters.

  ```
  "EMPLOYEE #   PART-TIME %   SALARY ($/unit)"
  ```

- This quoted string uses embedded quotation marks.

  ```
  PRINT "STAN ""THE MAN"" MUSIAL"
  ```

  The output from this PRINT statement appears below.

  ```
  STAN "THE MAN" MUSIAL
  ```

Unquoted String Constants

An unquoted string constant is a sequence of characters or spaces that:

- Contains no apostrophes, colons, or commas.
- Does not begin with a quotation mark.
- Does not begin or end with a space.

Unquoted string constants are used only in data statements and INPUT replies. An unquoted string constant in a DATA statement is limited by the length of the BASIC source line (255 characters). Response to an interactive input request is limited to 128 characters.

Examples The following example is an unquoted string.

```
GROSS WEEKLY EARNINGS (in $)
```

Data Type

Some identifiers carry with them an associated data type. The data type of an identifier establishes the kind of values that are stored under that name. Variable names and function names always have an associated data type. Subroutine names never have an associated data type.

The three NOS/VE BASIC data types are: integer, real, and string. A numeric identifier is either type integer or real. A string identifier is type string.

Data type is established either by using a data type specification symbol as the last character of an identifier, or by a type declaration statement.

Data Type Specification Symbols

The last character of an identifier can be used to specify data type.

An integer name is an identifier whose last character is a percent sign (%). An integer name is a type integer identifier.

A real name is an identifier whose last character is either an exclamation point (!) or a number sign (#). A real name is a type real identifier.

A string name is an identifier whose last character is a dollar sign ($). A string name is a type string identifier.

In general, an identifier is either a plain name, an integer name, a real name, or a string name. Remember that an identifier can have at most 31 characters, including the type specification character.

Examples The following are examples of integer names, real names, and string names.

Integer Names	Real Names	String Names
N%	X!	S$
ROW.TOTAL%	Y#	STATUS$
NO.OF.CARS%	CURRENT.GPA!	DEPT.NAME$
EXAM.1.SCORE%	TYPE.747.MPG#	MESSAGE.10$

Type Declaration Statements

Purpose	A type declaration statement specifies the data type (if appropriate) of any plain name whose first letter appears in a letter list. A type declaration statement affects plain names only. It has no affect on integer names, real names, or string names.
Format	DEFxxx letlist

xxx	Replaced by either INT, SNG, DBL, or STR, as appropriate.
letlist	List of letters and letter ranges that are separated by commas. A letter range is written L1-L2, where L1 and L2 are letters, and L2 does not alphabetically precede L1. The letter range L1-L2 is equivalent to listing all the letters between L1 and L2, inclusive. Any plain name that begins with a letter appearing in the letter list has the specified data type.

Remarks

- The four NOS/VE BASIC type declaration statements are as follows:

  - DEFINT, which declares type integer.
  - DEFSNG, which declares type real.
  - DEFDBL, which is equivalent to DEFSNG.
  - DEFSTR, which declares type string.

- By default, the data type (if any) of a plain name is real. This can be confirmed formally with a DEFDBL or DEFSNG statement, or overridden with a DEFINT or DEFSTR statement.

- Type declaration statements in an external function also apply to the function name and formal parameters. In an external subroutine, such statements apply to the formal parameters. (A subroutine name has no associated data type.)

Examples

- In an external routine, a letter can be specified in only one type declaration statement.

  ```
  DEFINT A,B,I-N,Y,Z

  DEFSNG D-H,V,W

  DEFDBL C-C,X

  DEFSTR O-U
  ```

- In general, a type declaration statement must precede every plain name that the statement affects. Thus, the following statements are out of order:

  ```
  LET A = 4  :  DEFINT A
  ```

  The variable A is type real by default, and is assigned the value 4.0. The subsequent attempt to alter the type results in a compile-time error.

- There is only one exception to the general rule. A type declaration statement that affects an external function name can follow the function specification statement. Thus, the order of the following statements is correct:

  ```
  REM MAIN PROGRAM
  DEFSTR G      'Type declaration for main program
  DECLARE EXTERNAL FUNCTION GREAT
  PRINT GREAT
  END PROGRAM
  EXTERNAL FUNCTION GREAT
  DEFSTR G      'Type declaraction for this function
  LET GREAT = "STRING"
  END FUNCTION
  ```

Data Type Compatibility

With one exception, type integer and type real data are compatible.

This means that:

- Integer and real data can be combined through arithmetic operations.
- An attempt to assign an integer value to a variable or function of type real is permitted. The value is converted to type real before being stored.
- An attempt to assign a real value to a variable or function of type integer is permitted. The value is rounded to the nearest integer before being stored.

Integer and real data are not compatible with type string data.

The one exception is the passing of data to user-defined routines through parameters. Here, an integer value cannot be passed to a real formal parameter. A real value cannot be passed to an integer formal parameter.

Variables

A variable is a named memory location. Different values can be stored inside the memory location at different times during program execution. A reference to the variable name accesses the value that is currently stored.

A variable name can be any valid identifier. The name carries with it an associated data type (integer, real, or string). The data type establishes the kind of values that are stored in the variable, and determines which operations can be performed on these values.

This section describes NOS/VE BASIC variables.

Typed Variables

An integer variable can only store integer values; a real variable can only store real values; and a string variable can only store string values.

Each variable is assigned an initial value before the expression involving that variable is evaluated. All numeric variables are assigned the value zero and all string variables are assigned the null string.

A NOS/VE BASIC integer variable can store any integer value n in the range (- 2^63 <= n <= 2^63 - 1), which is approximately the range (- 9.2*10^18 <= n <= 9.2*10^18). An attempt to store an integer value outside this range results in a runtime error.

An integer variable is named with either an integer name, or a plain name typed in a DEFINT statement.

A NOS/VE BASIC real variable can store any real value whose magnitude is less than 2^4095, which is approximately 5.2*10^1232. An attempt to store a real value whose magnitude is too large results in a runtime error.

A real variable is named with either a real name, or a plain name. The default data type is real. A DEFSNG or DEFDBL statement can be used to confirm the default.

A NOS/VE BASIC string variable can store a string value with at most 65,535 characters.

A string variable is named with either a string name, or a plain name typed by a DEFSTR statement.

A substring is a string variable consisting of zero or more consecutive character positions within a given string variable. A NOS/VE BASIC substring is expressed using colon-substring notation or a MID$ reference.

Supplied String Variables

NOS/VE BASIC supplies two string variables at runtime: TIME$ and DATE$.

TIME$

Purpose

TIME$ is an 8-character string variable whose default value is the current time as kept by NOS/VE.

Format

"hh:mm:ss"

hh	Hours in the range 00 through 23.
mm	Minutes in the range 00 through 59.
ss	Seconds in the range 00 through 59.

Remarks

- The value of TIME$ can be set within a program. If user-defined, its current value is the last assigned value plus the time elapsed since the assignment. TIME$ cannot be set in an INPUT or READ statement.

- If you specify only a single digit for a TIME$ component, a leading zero is provided. If you omit a component, then the default value 00 is provided.

- A runtime error results if an impossible time value, such as "24:00:00", is assigned.

Examples

- The following examples show different values for TIME$.

```
TIME$ = "12:15:30"

TIME$ = "4:7"         (set to "04:07:00")

TIME$ = "11"          (set to "11:00:00")

TIME$ = "23:59:59"    (last second of the day)

TIME$ = "0:0:0"       (midnight)

TIME$ = "::"          (midnight)

TIME$ = ":"           (midnight)

TIME$ = "10::32"      (set to "10:00:32")
```

DATE$

Purpose

DATE$ is a 10-character string variable whose default value is the current date as kept by NOS/VE.

Format

"mm-dd-yyyy"

mm	Month in the range 00 through 12.
dd	Day in the range 00 through 31.
yyyy	Year in the range 00 through 9999.

Remarks

- The value of DATE$ can be set within a program. If user-defined, slashes can replace hyphens. However, the value of DATE$ is always printed using hyphens. DATE$ cannot be set in an INPUT or READ statement
- The value of DATE$ advances when the value of TIME$ passes "00:00:00".
- If you specify only a single digit for the month or day component, a leading zero is provided.
- If you specify only two digits for the year component, then digits in the range:
  - 00 through 77 are interpreted as 2000 through 2077.
  - 78 through 99 are interpreted as 1978 through 1999.
- A runtime error results if an impossible date value, such as "09-31-1985", is assigned.

Examples

- The following examples show different values for DATE$.

```
DATE$ = "09-01-1965"

DATE$ = "7-4-1776"      (set to "07-04-1776")

DATE$ = "4/17/62"       (set to "04-17-2062")

DATE$ = "4/17/80"       (set to "04-17-1980")
```

Subscripted Variables

A variable that is not a substring can be either scalar or subscripted.

A scalar variable associates a name with a single memory location. In contrast, a subscripted variable shares its name with other members of a larger structure. This structure is called an array.

An array is a group of variables with the same data type that are referenced by a single name. This name is called the array name.

A specific variable in the array is accessed using the array name and a sequence of numbers called subscripts. The subscripts identify the variable by its position within the array. This variable is called a subscripted variable, or an array element.

A subscripted variable acts just like a scalar variable, but uses a more complex reference format. The naming and data type rules discussed in this chapter apply to both scalar and subscripted variables (array elements). Limits on the values of subscripted variables are the same as those on scalar variables of like data type.

An external routine can contain an array with the same name as a scalar variable because they have different reference formats.

Remember that the term variable applies to both scalar and subscripted variables, and that a substring is neither scalar nor subscripted.

For more information about arrays, see chapter 11 of this manual.

Double-Precision Vestiges

NOS/VE BASIC has no double-precision data type because real data in NOS/VE BASIC is approximately as precise as double-precision data in many microcomputer versions of BASIC.

However, some traces of the format of double-precision do occur in NOS/VE BASIC. Features with such traces are provided so that existing microcomputer BASIC programs can be used on NOS/VE with minimal changes.

The traces of the format of double-precision are found in:

- Constants that are followed by either an exclamation point (!) or a number sign (#).
- The exponential format that uses the letter D rather than the letter E to separate the mantissa from the exponent.
- Identifiers whose last character is either an exclamation point (!) or a number sign (#).
- Related library functions whose names contain an SNG, S, DBL, or D designation.

These formats differentiate between single- and double-precision in many microcomputer versions of BASIC, but are equivalent in NOS/VE BASIC.

Expressions and Assignment

This chapter describes the ways in which expressions are written and evaluated. This chapter also describes assignment statements. Assignment statements are executable statements that use expressions to define or redefine the values of variables.

Expressions and Assignment 4

Expressions are built by applying operators to constants, variables, and function references.

The four kinds of BASIC expressions are: arithmetic, string, relational, and logical. Each kind has its own set of operators and evaluation rules.

This chapter discusses how to construct and evaluate expressions. It also describes how assignment statements are used to store their values.

Expressions

An expression is one or more constants, variables, or function references that are linked by operators. Subexpressions occurring within an expression can be enclosed by parentheses.

A system of precedence determines the order in which the operations in an expression are performed.

A subexpression enclosed by parentheses is treated as a single operand. Its value must be computed before an operator can be applied to it. This means that parentheses always have the highest precedence. If parentheses are nested, the innermost expression is evaluated first.

Each kind of expression has its own set of permissible operations. The operations in each set are assigned precedences. For a given set, operations with a precedence of 1 are performed first, operations with a precedence of 2 are performed second, and so forth.

Evaluation of operators with equal precedence is performed left-to-right, although the evaluation of operands of equal precedence is not guaranteed. This may present a problem when the operands are functions that have side effects. Where such programming is done, the expression should be carefully parenthesized.

Expressions are classified as either numeric or string. Arithmetic, logical, and relational expressions are called numeric because they produce numeric values. String expressions produce string values.

When a BASIC numeric expression is evaluated, intermediate results are assigned to temporary storage. These assignments are subject to the same constraints and liable to the same errors as ordinary assignments. Possible errors are: numeric overflow, numeric underflow, and divide fault.

Each of these errors is fatal and induces the runtime error processing explained in chapter 6. An error can sometimes be avoided by parenthesizing an expression.

Examples

- This expression induces a numeric overflow when evaluated.

```
(2.0^2500.0) * (2.0^2500.0)/(2.0^2500.0)
```

- This expression is the algebraic equivalent and does not induce a numeric overflow.

```
(2.0^2500.0) * ((2.0^2500.0)/(2.0^2500.0))
```

A runtime error results if the value of an expression or subexpression falls outside the range allowed for variables of the corresponding data type.

Note that an expression can be a single constant, variable, or function reference.

Arithmetic Expressions

An arithmetic expression is one or more numeric constants, numeric variables, or numeric function references that are linked by arithmetic operators. Subexpressions occurring within an arithmetic expression can be enclosed by parentheses.

A unary operator (+ or -) can be the first token in an arithmetic expression. Two arithmetic operators cannot appear consecutively unless the second is a unary operator. Arbitrarily long sequences of consecutive unary operators are permitted.

The precedence for NOS/VE BASIC arithmetic operations appears below. Remember that parentheses always have the highest precedence.

Precedence	Operator	Operation
1	^	Exponentiation
2	+, -	Identity or negation (unary)
3	*, /	Multiplication or real division
4	\	Integer division
5	MOD	Modulo arithmetic
6	+, -	Addition or subtraction

For integer division, denoted by the reverse slant (\) operator, the operands, are rounded to the nearest integer. The division is then performed and the quotient is truncated to an integer.

Examples

- The operands are rounded, yielding (14/5). Division produces 2.8. Truncation results in the value 2.

  ```
  13.8\5.3
  ```

- Modulo arithmetic is denoted by the MOD operator. The operands are rounded to the nearest integers, and division is performed. The result is the integer remainder of this division.

  ```
  -13.8 MOD 5.3
  ```

  The operands are rounded, yielding (-14/5). Division yields -2 with a remainder of -4. The result is -4.

The circumflex (^) operator, the up-arrow on some keyboards, denotes exponentiation. A runtime error results if zero or a negative number is raised to a negative power.

The slant (/) operator denotes division. A runtime error results if division by zero is attempted.

For exponentiation and division, the operands are converted to type real, and real results are produced.

For all other arithmetic operations:

- If the operands are of like type, no operand conversions are performed. The result has the same type as the operands.

- If the operands are of different types, the integer operand is converted to type real. A real result is produced.

Examples

- Exponentiations are performed left-to-right. The value 2.0 is raised to the third power. The result, 8.0, is squared. The result, 64.0, is negated. The value of the expression is -64.0. In contrast, the expression (-2.0)^3^2 has the value 64.0.

  ```
  -2.0^3^2
  ```

- The order in which operations are performed is shown underneath the following sample expression:

  ```
                18 - 200 / 20 * 10 - 8 MOD 7 + 37

  Order --->     4     1    2    5     3    6
  ```

  The result is the value -46.

- The expression X - Y/X^2 + Y^2 is equivalent to (X - (Y/(X^2))) + (Y^2). If X = 2 and Y = 4, the value of this expression is 17. Compare the following expressions to to X- Y/X^2 + Y^2:

  ```
  (X - Y)/(X^2 + Y^2)

  (X - Y)/ X^2 + Y^2

  X - Y /(X^2 + Y^2)
  ```

  Using X = 2 and Y = 4, these expressions have the values -.1, 15.5, and 1.8, respectively.

String Expressions

A string expression is one or more quoted string constants, string variables, or string function references that are separated by plus signs. Subexpressions occurring within a string expression can be enclosed by parentheses.

In this context, the plus sign is called the concatenation operator. Concatenation, the only string operation, joins two string operands. The length of the string produced is the sum of the lengths of the operands.

The string expression (A$ + B$) is read "A is concatenated with B".

Examples

- If A$ = "START" and B$ = "UP", the expression:

  B$ + A$ has the value "UPSTART".

  A$ + B$ has the value "STARTUP".

- Consider the following assignment statements:

  ```
  LET C$ = "WORK:  555-1212"

  LET C$ = C$(1:4) + " PHONE" + C$(5:15)
  ```

  These statements insert the string " PHONE" in an appropriate place in C$. Thus, C$ is assigned the new value "WORK PHONE: 555-1212".

For more information about strings, see chapter 12.

Relational Expressions

Purpose	A relational expression compares the values of expressions with compatible data types.

Format

```
exp1  relop  exp2
```

exp1, exp2	Expressions that are both arithmetic or both string.
relop	Relational operator.

Remarks

- The operand expressions are evaluated and then compared. A value of 0 results if the comparison is false. A value of -1 results if the comparison is true.

Operator	Comparison
=	Equal To
<> or ><	Not Equal To
<	Less Than
<= or =<	Less Than Or Equal To
>	Greater Than
>= or =>	Greater Than Or Equal To

- Relational operators all have the same precedence. Remember that parentheses take precedence over all other operations.

- A character-by-character comparison is made when the operands of a relational expression are string expressions. Decisions are based on the sequence of ASCII character codes defined in the ANSI standard ASCII character set (see appendix B). This sequence is designed so that letter comparisons are made based on alphabetical order.

Examples

- The following expression is false because S follows D in the alphabet. The value 0 results.

  ```
  "FALETTI, STEVE" <= "FALETTI, DAN"
  ```

- This next expression is true because (512.0 > 64.0). The value -1 results.

  ```
  2^(3^2) > (2^3)^2
  ```

NOTE

Since a relational expression produces a numeric value (either 0 or -1), a compound expression such as (1 < 7 < 4) is allowed. However, such an expression does not have the usual mathematical interpretation.

The expression (1 < 7 < 4) results in the value -1, for true, even though the inequality is mathematically false. Evaluating the expression left-to-right, the comparison (1 < 7) yields the value -1. The comparison becomes (-1 < 4). This is true, so the value -1 results.

Testing compound mathematical inequalities requires use of the logical AND operator. Logical expressions are discussed next.

The values of the ASCII character codes for uppercase letters are less than the values of the codes for all the lowercase letters. When sorting string data, it is often important first to convert all the data to the same case. (See the LCASE and UCASE functions)

Logical Expressions

Purpose

A logical expression is typically used to make compound relational comparisons.

Format

```
expl  logop  exp2
```

expl, exp2	Numeric expressions. Although arithmetic expressions are permitted, these operand expressions are usually relational or logical.
logop	Logical operator.

Remarks

- After the operand expressions are evaluated, the logical operations are performed according to priority. If a 0 value results, the expression is considered false. If a nonzero value results, the expression is considered true.

Precedence	Operator	Operation
1	NOT	Logical Negation
2	AND	Logical Conjunction
3	OR	Logical Inclusive Disjunction
4	XOR	Logical Exclusive Disjunction
5	EQV	Logical Equivalence
6	IMP	Logical Implication

- A logical operator is applied bit-by-bit to its 64-bit numeric operands. Each bit of the result is set to either 0 or 1, based on the definition of the operator. In general, bit k of the result depends on the kth bit of each operand.

- If all bits are set to zero, the result is considered false. In this case, the value 0 results. If at least one bit is set to 1, the result is considered true. In this case, the decimal equivalent of the binary representation results.

Remarks (cont)

- If both operands have values that are either 0 or -1, the logical operation produces a value of either 0 or -1.

- The unary NOT operator can be the first token in a logical expression. Two logical operators cannot appear consecutively unless the second is the unary NOT operator. Arbitrarily long sequences of consecutive unary NOT operators are permitted.

- The precedence for NOS/VE BASIC logical operations appears in the following table.

- The corresponding bits of the operands of a logical operation must be in one of four states. The following truth table shows the results of applying each logical operation to each state. This defines each of the logical operations. The letters p and q denote corresponding bits.

p	q	NOT p	p AND q	p OR q	p XOR q	p EQV q	p IMP q
1	1	0	1	1	0	1	1
1	0	0	0	1	1	0	0
0	1	1	0	1	1	0	1
0	0	1	0	0	0	1	1

Examples

- This statement causes a branch to line 100 provided A falls between 1 and 5, inclusive.

  ```
  IF 1 <= A AND A <= 5 THEN GOTO 100
  ```

- This statement assigns the value "EXEMPT" to C$ if A$ has the value "UNEMPLOYED", or B$ has the value "CHILD", or both A$ and B$ have these respective values.

  ```
  IF A$ = "UNEMPLOYED" OR B$ = "CHILD" THEN LET C$ =
  "EXEMPT"
  ```

- This expression is always true regardless of the values of P and Q.

  ```
  NOT(P AND Q)  EQV  (NOT P  OR  NOT Q)
  ```

- The number 26 is 11010 binary; 9 is 1001 binary. Application of the AND operator results in a word with all bits, except the fourth bit from the right, set to 0 (result: 000...0001000). The logical expression is considered true. Y% is assigned the value 8, which is 1000 binary.

  ```
  LET Y% = 26 AND 9
  ```

- The number 26 is 11010 binary; 9 is 1001 binary. Application of the XOR operator results in a word with all bits, except the first, second, and fifth bits from the right, set to 0 (result: 000...00010011). The logical expression is considered true. Z% is assigned the value 19, which is 10011 binary.

  ```
  LET Z% = 26 XOR 9
  ```

Assignment Statements

An assignment statement assigns a value to a variable.

In BASIC, the equal sign is used to denote both the assignment operator and a relational operator. This can cause confusion. When reading an equal sign that denotes assignment, use the words "is assigned the value of", "receives the value of", or "becomes". This distinguishes the assignment usage from the comparison usage.

This section discusses the NOS/VE BASIC assignment statements: LET and SWAP. It also describes the CLEAR statement, which clears the variables in an external routine.

LET Statement

Purpose	Assigns the value of an expression to a variable.
Format	LET var = exp

LET	Optional keyword. This statement is the only one that can be written without a keyword.
var	Name of the variable being assigned a value.
exp	Expression whose value is compatible with the data type of VAR.

Remarks

- The value of EXP is assigned to the variable VAR, destroying any previous value. If VAR is a numeric variable, the value of EXP is converted to the appropriate data type. Thus, if VAR is integer, the value of EXP is rounded to the nearest integer and then stored. If VAR is real, the value of EXP is converted to a real number and then stored.

- Each identifier for a variable or a function is given a default initial value.

- The default initial value for:

  - A type integer identifier is 0 (integer zero).
  - A type real identifier is 0.0 (real zero).
  - A type string identifier is "" (the null string).

Examples

- The rounded value -4 is assigned to the variable A%.

  ```
  LET A% = -3.5
  ```

- The value of X is incremented by one. This example illustrates the importance of distinguishing an assignment from a comparison. If the above were used in the context of a relational expression, it would always be false.

  ```
  X = X + 1
  ```

- The string variable S$ is assigned the result of the concatenation of A$ with a substring of B$.

  ```
  LET S$ = A$ + B$(2:6)
  ```

- If Z has the value zero, then this statement assigns the value (0.0 + 3.0) to the variable Z, where Z is real by default.

  ```
  LET Z = Z + 3
  ```

SWAP Statement

Purpose Exchanges the values of two variables.

Format SWAP var1 , var2

var1, var2 Variables having compatible data types.

Remarks This single statement is equivalent to the three statements:

```
LET TEMP = VAR1
LET VAR1 = VAR2
LET VAR2 = TEMP
```

TEMP is a variable that has the same data type as VAR1.

Examples The fourth line of this program fragment exchanges the values of A$ and B$.

```
1 LET A$ = "TO DO"
2 LET B$ = "TIME"
3 PRINT "SO MUCH ";A$;" AND SO LITTLE ";B$;"!"
4 SWAP A$,B$
5 PRINT "SO MUCH ";A$;" AND SO LITTLE ";B$;"!"
```

The output produced appears below.

```
SO MUCH TO DO AND SO LITTLE TIME!
SO MUCH TIME AND SO LITTLE TO DO!
```

CLEAR Statement

Purpose

The CLEAR statement discards the data of the external routine in which it appears. This includes all the data of any embedded internal routines.

Format

CLEAR

Remarks

- When a CLEAR statement in an external routine is executed:
  - All numeric scalar variables within that external routine are set to zero. All string scalar variables are set to the null string.
  - Both the lower and upper bounds of each dimension of each array within that external routine are set to zero or one, depending on the option base. The value of the single remaining element is set to either zero or the null string, as dictated by data type.
- A CLEAR statement in an external routine:
  - Also clears the variables and arrays that the external routine shares with other external routines through the COMMON statement.
  - Does not clear variables and arrays that are passed to that external routine as actual parameters.
- The ERASE statement, used with arrays, is related to the CLEAR statement. For more information, see chapter 11.

Examples

When the CLEAR statement in this program fragment is executed, B and C% are assigned the value zero, and D$ is assigned the value of the null string.

```
LET B = 2.0
LET C% = 44
LET D$ = "BATTERIES"
CLEAR
```

Decision and Branching 5

Decision and branching provides the means of altering the normal sequential flow of execution.

Decision and Branching 5

Statements are executed one at a time in the order they appear unless a control statement overrides this sequential execution.

A control statement establishes conditions for altering sequential execution and passes control to a specified statement when these conditions are met.

Possible results include a transfer of control under all circumstances or the execution of an additional block of statements before sequential execution is resumed.

Control statements that pertain to error processing are discussed in the Runtime Error Processing chapter. Control statements involving interaction between routines are discussed in the User-Defined Functions chapter and the Subroutines chapter.

This chapter discusses NOS/VE BASIC control statements that are used within a single routine to control the order of execution under normal (error-free) circumstances.

GOTO Statements

This section discusses the two NOS/VE BASIC GOTO statements. The unconditional GOTO statement causes a branch under all circumstances. The ON-GOTO statement causes a branch to one of several possible locations based on the value of an index expression.

A GOTO statement cannot be used to branch into or out of a routine (whether external or internal). The keyword GOTO and the two-word sequence GO TO are interchangeable in NOS/VE BASIC.

Unconditional GOTO Statement

Purpose Transfers control to the line with the specified label.

Format GOTO label

label Label of the line to which execution control passes.

Examples This statement causes an unconditional branch to the line labeled 100.

```
GOTO 100
```

ON-GOTO Statement

Purpose	Transfers control to a line whose label is among a group of specified labels.

Format

```
ON test GOTO label1 , label2 , ... , labeln
```

test	Numeric expression the value of which determines the position in the list of the labels used. The label then specifies the line to which execution control passes.
labeln	List of labels giving the destinations of the branch.

Remarks

The value of TEST is rounded to the nearest integer K, and control passes to the line labeled LABELK. In other words, an unconditional GOTO LABELK is executed. If K is less than one or greater than N, a runtime error results.

Examples

The following statement ON S% GOTO 200, 600, 500 transfers control to the line labeled:

200 if S% has the value 1.

600 if S% has the value 2.

500 if S% has the value 3.

If S% has any value greater than or less than the range of values 1 through 3, a runtime error results.

GOSUB Statements

NOS/VE BASIC provides two GOSUB statements. The unconditional GOSUB statement causes a branch under all circumstances. The ON-GOSUB statement causes a branch to one of several possible locations based on the value of an index expression.

These two statements operate like the unconditional GOTO and ON-GOTO statements, respectively, with one additional feature. They provide a way of returning to the statement following the one that caused the branch. This is accomplished by the RETURN statement.

A GOSUB statement cannot be used to branch into or out of a routine (whether external or internal). The keyword GOSUB and the two-word sequence GO SUB are interchangeable in NOS/VE BASIC.

This section discusses the unconditional GOSUB, ON-GOSUB, and RETURN statements. It includes a brief description of the data structure called a stack, which controls the branch and return process.

Branch and Return Process

A GOSUB statement and a RETURN statement work together to effect a branch and return. To understand this process, it might help to know a few things about the data structure known as a stack.

A stack is a list that allows insertions and deletions at the top only. Items in a stack are processed on a "Last In First Out" basis.

Inserting an item on top of a stack is referred to as pushing because you can visualize this action pushing the other stack items down one position. Deleting an item from the top of a stack is referred to as popping because you can visualize this action causing the other stack items to pop up one position.

A stack of plates in a cafeteria is an excellent model for the workings of this data structure.

When an unconditional GOSUB or an ON-GOSUB statement is executed, the address of the statement that follows it is pushed onto a stack. When a RETURN statement is reached, the address that was pushed onto the stack previously is now popped off the stack and control passes to the statement with this address.

The RETURN statement has the format:

```
    RETURN
```

The RETURN statement can appear any number of times in an external routine.

Each routine has its own stack. When control is returned to a calling routine from a called routine, the items in the stack of the called routine are discarded. Hence, each time a routine is called, it begins with an empty stack. A runtime error results if a RETURN statement is executed when the stack is empty.

Unconditional GOSUB Statement

Purpose	Transfers control to the line with the specified label. In addition, this statement provides a way of returning to the statement following the GOSUB statement.
Format	GOSUB label
	label Label of the line to which execution control passes.
Remarks	The RETURN statement is used to return control to the statement following the unconditional GOSUB statement.
Examples	In this program fragment, the GOSUB statement transfers control to the line labeled 500. When the RETURN statement is reached, control passes to the PRINT statement in the line labeled 100, and execution continues.

```
    GOSUB 500
100 PRINT "BACK FROM DESTINATION BLOCK"
    .
    .
500 REM Destination Block Begins Here
    .
    .
    RETURN
```

ON-GOSUB Statement

Purpose

Transfers control to a line whose label is among a group of specified labels. In addition, this statement provides a way of returning to the statement following the ON-GOSUB statement.

Format

```
ON  test  GOSUB  label1 , label2 , ...  , labeln
```

test	Numeric expression whose value determines the position in the list of the labels used. The label then specifies the line to which execution control passes.
labeln	List of labels giving the destinations of the branch.

Remarks

- The value of TEST is rounded to the nearest integer K. Control passes to the line labeled LABELK. In other words, an unconditional GOSUB LABELK is executed. If K is less than one or greater than N, a runtime error results.

- The RETURN statement is used to return control to the statement immediately following the ON-GOSUB statement.

Examples

The statement ON SGN(A) + 2 GOSUB 800, 400, 500 transfers control to the line labeled:

800 if (SGN(A) + 2) is 1.

400 if (SGN(A) + 2) is 2.

500 if (SGN(A) + 2) is 3.

The function reference SGN(A) returns the value -1, 0, or 1 depending on whether A is negative, zero, or positive, respectively.

If the expression (SGN(A) + 2), when rounded to the nearest integer, has any value other than 1, 2, or 3, a runtime error results.

Line IF Constructions

A line IF construction is a multi-statement decision and branching structure that is confined to a single line.

A line IF construction has three components:

- An IF condition, which determines where control is transferred.
- A THEN clause, which is executed when the IF condition is true.
- An optional ELSE clause, which is executed when the IF condition is false. If no ELSE clause has been provided, control passes to the next line.

Line IF Constructions

Purpose

Creates a multi-statement decision and branching structure that is confined to a single line.

Format

```
IF  condition  THEN  clause1  ELSE  clause2
IF             GOTO  label    ELSE  label
```

condition	Expression whose value directs the flow of execution. This expression is usually relational or logical, but it can be arithmetic.
clause1	Series of BASIC statements that are separated by colons.
clause2	Optional series of BASIC statements that are separated by colons. If omitted, the keyword ELSE is also omitted.
label	Series of BASIC statements that are separated by colons.

Remarks

- Remember that the entire line IF construction is confined to a single line.

- An IF condition is a numeric expression (usually relational or logical) whose value directs the flow of execution.

- If the value of the IF condition is:

  - Zero (representing false in a logical context), the THEN clause is ignored, and control passes to the first statement of the ELSE clause. If no ELSE clause has been provided, control passes to the next line.

  - Nonzero (representing true in a logical context), control passes to the first statement of the THEN clause.

Remarks (cont)

- If the end of a clause is reached (a branching statement might prevent this), control passes to the line following the IF construction.

- Consider the following line IF construction:

  ```
  IF  condition  THEN  GOTO  label  ELSE  clause2
  ```

  The following two special constructions are equivalent to the one above:

  ```
  IF  condition  THEN  label  ELSE  clause2
  IF  condition  GOTO  label  ELSE  clause2
  ```

  The first special form allows you to omit the keyword GOTO when the THEN clause contains a single unconditional GOTO statement.

  The second special form allows you to omit the keyword THEN when the THEN clause contains a single unconditional GOTO statement.

- Consider the following line IF construction:

  ```
  IF  condition  THEN  clause1  ELSE  GOTO  label
  ```

  The following special construction is equivalent to the one above:

  ```
  IF  condition  THEN  clause1  ELSE  label
  ```

  This special form allows you to omit the keyword GOTO when the ELSE clause contains a single unconditional GOTO statement.

  The special forms described apply only in conjunction with the unconditional GOTO statement in line IF constructions. They cannot be generalized.

Examples

- If B$ has the value "YES", then C is incremented by one. Otherwise, D is incremented by one. Execution continues with the next line.

```
IF B$ = "YES" THEN C = C+1 ELSE D = D+1 PRINT C, D
```

- These two statements are equivalent. They illustrate the use of a logical expression in the IF condition.

```
IF (Y<1 XOR 5<Y) THEN PRINT "YES" ELSE PRINT "NO"

IF (1<=Y AND Y<=5)THEN PRINT "NO" ELSE PRINT "YES"
```

- If A% has the value 0, then "FALSE" is printed. Otherwise, "TRUE" is printed. Execution continues with the next line.

```
IF A% THEN PRINT "TRUE" ELSE PRINT "FALSE"
LET A% = A% + 1
```

- The PRINT and LET statements are executed only if X is positive. Execution continues with the next line.

```
IF X > 0 THEN PRINT "X IS POSITIVE" : LET S$ = "ON"
LET X = X + 1
```

Examples (cont)

- If A% has the value 1, control passes to the line labeled 100. A RETURN statement corresponding to the GOSUB statement transfers control back to the PRINT statement in this line IF construction.

  ```
  IF A% = 1 THEN GOSUB 100 : PRINT "BACK"
  ```

- The following three statements are equivalent:

  ```
  IF S$ = "YES" THEN GOTO 100
  ```

  ```
  IF S$ = "YES" GOTO 100
  ```

  Each of these statements causes a branch to the line labeled 100 if S$ has the value "YES". Otherwise, execution continues with the next line.

  ```
  IF S$ = "YES" THEN 100
  ```

- If C% has the value 0, control passes to the line labeled 200. Otherwise, control passes to the line labeled 400.

  ```
  IF C% = 0 THEN 200 ELSE 400
  ```

Block IF Constructions

A block IF construction is a multi-statement decision and branching structure that is not confined to a single line.

A block IF construction has five components:

- An IF condition, which initially directs the flow of execution.
- An initial THEN block, which is executed when the IF condition is true.
- An optional series of one or more ELSEIF constructions, the first of which is executed when the IF condition is false.
- An ELSEIF construction consists of an ELSEIF condition followed by a THEN block.
- An optional ELSE block, which is executed when the IF condition and all the ELSEIF conditions are false.
- An ENDIF statement, which denotes the physical end of the block IF construction.

Block IF Constructions

Purpose	Creates a multi-line decision and branching structure.

Format

```
IF  incond  THEN
  inblock
ELSEIF  condl  THEN
  blockj
ELSE
  elblock
ENDIF
```

incond	Expression whose value initially directs the flow of execution.
inblock	THEN block.
condJ	Expression of the Jth ELSEIF construction where (1 <= J <=N).
blockJ	THEN block of the Jth ELSEIF construction, where, (1 <= J <= N).
elblock	Optional ELSE block. If omitted, the preceding ELSE is also omitted.

Remarks

- An IF condition is an expression whose value initially directs the flow of execution. This expression is usually relational or logical, but it can be arithmetic. If the value of the IF condition is:

  - Zero (denoting false in a logical context), the initial THEN block is ignored, and control passes to the first ELSEIF construction. If no ELSEIF constructions are provided, control passes to the first statement of the ELSE block. If no ELSE block is provided, control passes to the statement following the ENDIF statement.

  - Nonzero (denoting true in a logical context), control passes to the first statement of the initial THEN block.

- If the end of a THEN block or an ELSE block is reached (a branching statement might prevent this), control passes to the statement following the ENDIF statement.

Remarks (cont)

- An ELSEIF construction is an ELSEIF condition followed by a THEN block. An ELSEIF condition is an expression whose value further directs the flow of execution. This expression is usually relational or logical, but it can be arithmetic. If the value of an ELSEIF condition is:

  - Zero, the corresponding THEN block is ignored, and control passes to the next ELSEIF construction. If no ELSEIF construction follows, control passes to the first statement of the ELSE block. If no ELSE block is provided, control passes to the statement following the ENDIF statement.

  - Nonzero, control passes to the first statement of the corresponding THEN block.

- The format presented for block IF constructions is standard, but not required. In particular, line feeds can be replaced by colons. Thus, the following line is a legal format for a block IF construction.

  ```
  IF condition THEN : block1 : ELSE : block2 : ENDIF
  ```

  This single line format is impractical, since a line IF construction would be equally effective with less typing. However, formats other than the standard one can be devised using this as a base.

- The ELSEIF construction is optional.

Examples

- This block IF construction computes total purchase cost. The cost per item is one of two values, depending on the amount purchased. If the value of QUANTITY% is less than 100, the lines labeled 10 and 20 are executed, and control passes to the line labeled 50. If the value of QUANTITY% is 100 or larger, the lines labeled 30 and 40 are executed, and control passes to the line labeled 50.

```
   IF QUANTITY% < 100 THEN
10   PRINT "NORMAL COST PER ITEM IS:  $";COST1
20   LET TOTAL.COST = QUANTITY% * COST1
   ELSE
30   PRINT "SPECIAL COST PER ITEM IS:  $";COST2
40   LET TOTAL.COST = QUANTITY% * COST2
   ENDIF
50   PRINT
```

- This block IF construction has no ELSE block. If the value of S$ is "YES", a receipt request is handled. A RETURN statement corresponding to the GOSUB statement transfers control to the line labeled 90. If the the value of S$ is not "YES", no receipt is issued, since control passes directly to the line labeled 100.

```
   IF S$ = "YES" THEN
     LET SWITCH$ = "ON"
     PRINT "YOU HAVE REQUESTED A RECEIPT."
     PRINT "IT WILL BE ISSUED IN A MOMENT."
     GOSUB 500 ' Branch to print receipt.
90   PRINT "YOUR RECEIPT HAS BEEN ISSUED."
   ENDIF
100  PRINT
```

Examples (cont)

- This block IF construction prints the letter grade that goes with a 100-point exam score, using a straight percentage grading scale.

```
 5  PRINT "YOUR LETTER GRADE IS:  ";
10  IF     SCORE >= 90 THEN
15      PRINT "A"
20  ELSEIF SCORE >= 80 THEN
25      PRINT "B"
30  ELSEIF SCORE >= 70 THEN
35      PRINT "C"
40  ELSEIF SCORE >= 60 THEN
45      PRINT "D"
50  ELSE
55      PRINT "F"
60  ENDIF
65      PRINT "THANK YOU"
```

If SCORE is greater than or equal to 90, the grade is A.

If SCORE is greater than or equal to 80 and less than 90, the grade is B.

If SCORE is greater than or equal to 70 and less than 80, the grade is C.

If SCORE is greater than or equal to 60 and less than 70, the grade is D.

If SCORE is less than 60, the grade is F.

NOTE

If a block IF construction has many ELSEIF conditions, place the cases that are most likely to be true near the top. This speeds up execution.

Truncation errors occur during real arithmetic because the computer can use only a limited number of digits to express a decimal number. Usually such errors affect only the least significant digits, and are negligible from a practical standpoint. They can, however, affect tests for equality involving real numbers. You might find the use of inequalities more appropriate than equalities for real number comparisons.

Looping Structures

A looping (or iterative) structure provides for repeated execution of a group of statements. The statements themselves remain the same, but the data involved is allowed to change.

NOS/VE BASIC provides two looping structures: the FOR-NEXT loop, and the WHILE-WEND loop.

In general, the FOR-NEXT loop is appropriate when the desired number of repetitions is known upon loop entry. The WHILE-WEND loop is generally used when the desired number of repetitions is undetermined on loop entry.

This section discusses these two looping structures.

FOR-NEXT Loops

The FOR-NEXT loop is a multi-line structure that causes a group of statements to be executed a specified number of times. This structure is appropriate when the desired number of repetitions is known before the loop is entered.

The FOR-NEXT loop has three components: the FOR statement, the loop body, and the NEXT statement.

```
FOR statement
   loop body
NEXT statement
```

FOR Statement

Purpose	Marks the beginning of the loop and controls the number of times the loop is executed.
Format	FOR counter = initial TO limit STEP size

counter	Numeric scalar variable whose value controls the looping process. This variable is called the control variable or the counter.
initial	Numeric expression whose value is the initial value of the counter.
limit	Numeric expression whose value is the limit value for the counter.
size	Optional numeric expression whose value determines the increment value for the counter. If omitted, the keyword STEP is also omitted and the default value 1 is assumed.

Remarks

- Execution of the FOR statement on entry to the loop:
  - Establishes the limit and increment values for the loop counter.
  - Assigns an initial value to the loop counter.
  - Performs the first loop exit test on the loop counter to determine whether or not the loop body is to be executed.
- The loop body is an optional block that contains the statements to be executed.
- The NEXT statement marks the physical end of the loop. Its execution increments the value of the loop counter and passes control back to the loop exit test in the FOR statement. This test determines whether or not the loop body is reexecuted.

```
    FOR  counter = initial  TO  limit  STEP  size
```

  When control first reaches the FOR statement, the values of INITIAL, LIMIT, and SIZE are computed. The control variable COUNTER is then assigned the value of INITIAL.

  The following statements set the limit value at 15 before assigning the initial value 1 to the control variable N.

```
    LET N = 15
    FOR N = 1 TO N
```

Remarks (cont)

- The loop exit test, which determines whether the loop body is executed, uses the following criteria.

  The loop body is executed as long as the test expression is nonnegative.

  ```
  (LIMIT - COUNTER) * SGN(SIZE)
  ```

  The SGN function reference returns the value -1, 0, or 1, depending on whether the value of its argument is negative, zero, or positive, respectively.

  When the test expression is negative, control passes to the statement following the NEXT statement.

- The expressions for LIMIT and SIZE are evaluated only when the FOR statement is executed upon entry to the loop. Changes made within the loop body to variables in these expressions do not affect the number of times the loop is executed. However, changes made within the loop body to COUNTER do affect the number of times the loop is executed.

  You might want to know in advance the number of times a loop will be executed. Assuming the loop body does not modify the value of COUNTER, and does not cause an early loop exit.

  ```
  MAX( (LIMIT + SIZE - INITIAL) SIZE , 0)
  ```

  This expression computes the number of times a FOR-NEXT loop will be executed. The MAX function reference returns the maximum of the values of its two arguments.

- A branch into the middle of a loop body without execution of the FOR statement is permitted but not recommended. When control is passed to the FOR statement, the loop exit test uses whatever values are currently stored in the specified variables. If the variables did not previously exist, default zero values are provided. Many problems, such as an endless loop, can result from such an ill-advised branch.

NEXT Statement

Purpose	Marks the physical end of the loop. Its execution increments the value of the loop counter and passes control back to the loop exit test in the FOR statement.
Format	NEXT counter

counter	Optional appearance of the name of the control variable specified in the FOR statement. If COUNTER is omitted, the NEXT statement corresponds to the nearest preceding FOR statement that does not have a corresponding NEXT statement.

Remarks

- When the NEXT statement is reached, the value of COUNTER is incremented by the value of SIZE (which can be negative). Control passes back to the loop exit test.

- Looping continues until the loop exit test stops it, or until control is transferred out of the loop by some other means. If control passes to the FOR statement directly from within the loop body, bypassing the NEXT statement, the loop is reinitialized.

FOR-NEXT Examples

Examples

- The two program fragments below each compute the sum of the first N terms of the series:

  1 + (1/3) + (1/5) + ... + (1/J) + ...

  ```
  'FRAGMENT #1
  DEFINT J,N
  INPUT N
  LET SUM = 0.0
  FOR J = 1 TO N
    SUM = SUM + 1 / (2 * J - 1)
  NEXT J
  ```

  ```
  'FRAGMENT #2
  DEFINT J,N
  INPUT N
  LET SUM = 0.0
  FOR J = 1 TO 2 * N - 1 STEP 2
    SUM = SUM + 1 / J
  NEXT
  ```

  In fragment #1, a default increment size of 1 is assumed. In fragment #2, the control variable J is omitted from the NEXT statement.

  Note that the loop exit test for a FOR-NEXT loop occurs at the top of the loop. Thus, it is possible that the loop body is never executed.

- On entry to a loop that begins with this FOR statement, the loop exit test immediately passes control to the statement following the corresponding NEXT statement. The loop body is never executed.

  ```
  FOR K% = 1 TO 10 STEP -3
  ```

Examples

- The FOR-NEXT loop is first illustrated below and then is expressed again using a line IF construction.

```
    FOR X = 1.0 TO 10.0 STEP 0.5
        .
        .
    NEXT X
90  REM


    LET X = 1.0
50  IF 10.0 - X < 0 THEN GOTO 90
        .
        .
    LET X = X + 0.5
    GOTO 50
90  REM
```

- FOR-NEXT loops can be nested inside of other FOR-NEXT loops. Note that the loop body of the inside loop must be completely contained within the loop body of the outside loop.

```
FOR I = N TO 1 STEP -1
    FOR J = 1 TO I
        PRINT "*";
    NEXT J
    PRINT
NEXT I
```

This program fragment prints an inverted right triangle of asterisks when N is a positive integer. The output when N has the value 4 appears below.

```
****
***
**
*
```

Special Formatted NEXT

Purpose	Permits a special format for the NEXT statement to facilitate the case of nested loops.
Format	NEXT conlist

conlist	Optional ordered list of control variables that are separated by commas. The control variables listed must appear in the order of the nesting, with the control variable for the innermost loop listed first, and that of the outermost loop listed last.

Remarks — This format is a shortcut for a series of consecutive NEXT statements. The order of the statements corresponds to the order of the list.

Examples — The following program fragments are equivalent.

```
'FRAGMENT #1                'FRAGMENT #2
FOR I = 3 TO 7 STEP 2       FOR I = 3 TO 7 STEP 2
    FOR J = 2 TO 4              FOR J = 2 TO 4
    PRINT I * J;                    PRINT I * J;
NEXT J, I                       NEXT J
                            NEXT I
```

Each fragment generates the output below:

```
6 9 12 10 15 20 14 21 28
```

WHILE-WEND Loops

The WHILE-WEND loop is a multi-line structure that causes a group of statements to be repeatedly executed. This structure is appropriate when the desired number of repetitions is undetermined on loop entry.

The WHILE-WEND loop has three components: the WHILE statement, the loop body, and the WEND statement.

```
WHILE  statement
    loop body
WEND
```

WHILE Statement

Purpose	Marks the beginning of the loop, and performs a loop exit test to determine whether or not the loop body is executed.
Format	WHILE condition

condition	Numeric expression whose value determines whether the loop body is executed. This expression is usually relational or logical, but it can be arithmetic.

Remarks

- If the value of CONDITION is:
  - Nonzero (representing true in a logical context), the loop body is executed.
  - Zero (representing false in a logical context), control passes to the statement following the WEND statement.
- The loop exit test determines whether or not the loop body is reexecuted.
- The loop exit test for a WHILE-WEND loop occurs at the top of the loop. Thus, it is possible that the loop body is never executed.

WEND Statement

Purpose	Marks the physical end of the loop. Its execution passes control back to the WHILE statement.
Format	WEND
Remarks	For information on the loop body see the FOR Statement section.

WHILE-WEND Example

Examples

This program fragment computes the number of terms needed for the product to become less than the value of LOWER.BOUND. If LOWER.BOUND is given the value 0.13, the value 3 is printed.

```
    LET N% = 1 : LET PRODUCT = 1
    INPUT LOWER.BOUND ' Set lower bound for the product.
    'Eliminate unacceptable lower bounds.
    IF LOWER.BOUND <= 0 THEN PRINT "INFINITE" : END
    WHILE PRODUCT >= LOWER.BOUND
        LET PRODUCT = PRODUCT * (0.5)^N%
        LET N% = N% + 1
    WEND
90  PRINT "NUMBER OF TERMS: "; N%
```

System Interface

You can execute NOS/VE commands from within a BASIC program with the RUN and SCL statements.

This section discusses these two system interface statements.

RUN Statement

Purpose

Executes a NOS/VE command, then terminates the BASIC program.

Format

RUN command

command — String expression whose value is the NOS/VE command to be executed.

Remarks

- When the RUN statement is executed, all BASIC files are closed, and the value of COMMAND is passed to NOS/VE for processing as a separate task. After the command is processed, the BASIC program is terminated, and control is transferred to the environment from which the NOS/VE BASIC program was invoked. Any error status resulting from execution of the command becomes the status of the NOS/VE BASIC program.

- The RUN command initiates a new task while the original task, the BASIC program, still exists.

- The number of concurrent tasks you can run is limited. The default maximum is twenty concurrent tasks; be careful not to exceed this limit. Your site administrator can change the default number of tasks by changing the TASK_LIMIT validation attribute associated with your user name.

- When you specify the NOS/VE command in the COMMAND parameter, you should omit the STATUS parameter from the NOS/VE command. If you include the STATUS parameter and an error occurs while the NOS/VE command executes, the BASIC program terminates normally; you see no indication of the error condition. By omitting the STATUS parameter, any abnormal status condition resulting from execution of the command becomes the status of the BASIC program.

Examples

This statement deletes the file SCRATCH from the working catalog and terminates the BASIC program. For more information about the DELETE_FILE (DELF) command, see the NOS/VE System Usage manual.

```
RUN "DELF SCRATCH"
```

SCL Statement

Purpose	Transfers control to NOS/VE so that a specified NOS/VE command can be executed.
Format	SCL command
	command: String expression whose value is the NOS/VE command to be executed.

Remarks

- When the SCL statement is executed, all BASIC files are left open, and the value of COMMAND is passed to NOS/VE for processing. After the command is processed, control returns to the program, and execution continues with the next statement. If an error occurs during processing of the command, a BASIC runtime error results.

- The number of concurrent tasks you can run is limited. The default maximum is twenty concurrent tasks; be careful not to exceed this limit. Your site administrator can change the default number of tasks by changing the TASK_LIMIT validation attribute associated with your user name.

- You should not use the SCL statement to attach a file you later open with an OPEN statement. The OPEN attaches the file internally; if the file is already attached, the OPEN might fail due to a share mode conflict. For information about attaching files, see the NOS/VE System Usage manual.

- The SCL command initiates a new task while the original task, the BASIC program, still exists. Up to eleven separate tasks can run concurrently; be careful not to exceed this limit.

- When you specify the NOS/VE command in the COMMAND parameter, you should omit the STATUS parameter from the NOS/VE command. If you include the STATUS parameter and an error occurs while the NOS/VE command executes, the BASIC program continues running normally; you see no indication of the error condition. By omitting the STATUS parameter, any abnormal status condition resulting from execution of the command becomes the status of the NOS/VE BASIC program, resulting in a BASIC runtime error.

Examples

The SCL statement deletes the file SCRATCH from the working catalog and returns control to the BASIC program. Execution continues with the line labeled 100. For more information about the DELETE_FILE (DELF) command, see the NOS/VE System Usage manual.

```
     SCL "DELF SCRATCH"
100  .
     .
     .
```

Runtime Error Processing 6

This chapter describes the statements and library functions used to process runtime errors.

Runtime Error Processing 6

Runtime refers to the time during which a program is being executed.

Runtime errors need not cause program termination. Instead, you can choose to handle and clear these errors from within your program.

Runtime error diagnostics (error messages) for uncleared errors are written to the NOS/VE standard file $ERRORS. The default connection for the standard error file is the listing file OUTPUT. For interactive mode, this means that diagnostics appear at the terminal.

A complete listing of the NOS/VE BASIC runtime error diagnostics appears in the Diagnostic Messages for NOS/VE manual.

This chapter discusses the NOS/VE BASIC statements and library functions used to process runtime errors.

Error Processing Overview

When execution control first reaches a routine, default error handling is in effect. You can override the default and take control of error handling with the ON ERROR statement. Errors can then be cleared with the RESUME statement.

This section provides an overview and model to help you visualize the dynamics of error processing.

Introduction to Error Handling

If a runtime error occurs, a diagnostic describing the error is saved. The occurrence of these events is denoted by the phrase "an error/diagnostic results". How an error is processed depends on whether default or user error handling is in effect.

When an error occurs, it is located in one of two environments:

- In an internal routine.
- In the portion of an external routine that is outside of all embedded internal routines.

With this in mind, a program can be thought of as a collection of environments. Within each environment, you can choose whether default or user error handling is in effect. However, an error can be cleared only through user handling. If an error is cleared, its corresponding diagnostic is deleted without being printed. Hence, a diagnostic that is printed always corresponds to an uncleared error.

Default Error Handling

An error in the portion of the main program that is outside of all embedded internal routines is located at the highest possible level. This environment is referred to as the top level environment. Any other environment is referred to as a low level environment.

If default error handling is in effect when a runtime error/diagnostic occurs in the top level environment:

- The diagnostics that have not been deleted are written to the NOS/VE standard file $ERRORS.

- The program is terminated.

If default error handling is in effect when a runtime error/diagnostic results in a low level environment:

- Control returns to the place where the routine was called (the call site).

- Another error/diagnostic results at the call site because of the return from a called routine with an uncleared error.

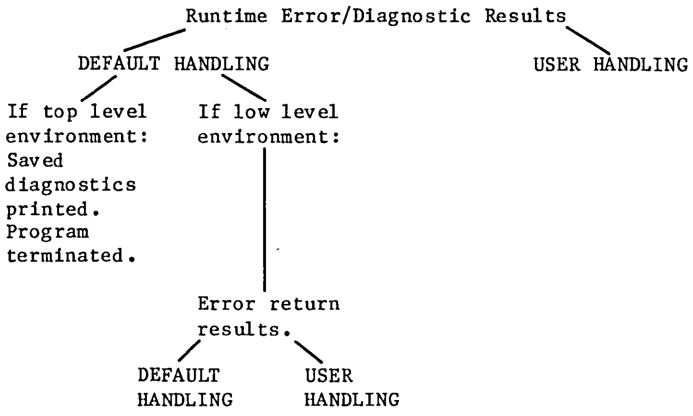
For simplicity, the return and resulting error/diagnostic are collectively referred to as an error return. Note that diagnostics are saved in chronological order.

Error handling now continues in the calling routine. The next action depends on whether default or user handling is in effect in the new environment.

If default error handling remains in effect in each new environment, control is transferred upward from call site to call site.

This series of error returns continues until user handling is in effect in some environment, or until the top level environment is reached with default handling still in effect.

All the errors in such a series, including the original error, are associated with the final error in the series. If this final error is cleared through user handling, all the associated errors are also cleared and the corresponding diagnostics are deleted.


Runtime Error/Diagnostic Results
DEFAULT HANDLING
USER HANDLING
If top level environment: Saved diagnostics printed. Program terminated.
If low level environment:
Error return results.
DEFAULT HANDLING
USER HANDLING

User Error Handling

The ON ERROR statement is used to activate user error handling within an environment.

This means that:

- An ON ERROR statement in an internal routine applies only to that internal routine.

- An ON ERROR statement in the portion of an external routine that is outside of all embedded internal routines does not apply to those embedded routines.

The ON ERROR statement specifies where control is to be transferred if a runtime error occurs in its environment. Ideally, the statements next executed either eliminate or bypass any problems resulting from the error that has occurred. Note that these statements can simply ignore the error, although this could cause further problems.

Suppose that user error handling is in effect when a runtime error/diagnostic results.

If a previous error in the current environment has not yet been cleared:

- Another error/diagnostic results because concurrent errors in an environment cannot be handled.

- The diagnostics that have not been deleted are written to the NOS/VE standard file $ERRORS.

- The program is terminated.

Otherwise, control passes to the line specified in the governing ON ERROR statement, and one of three outcomes eventually occurs.

The following are the possible outcomes:

- A RESUME statement is reached. If a RESUME statement is executed:

  - The error in the current environment, and all of its associated errors are cleared.

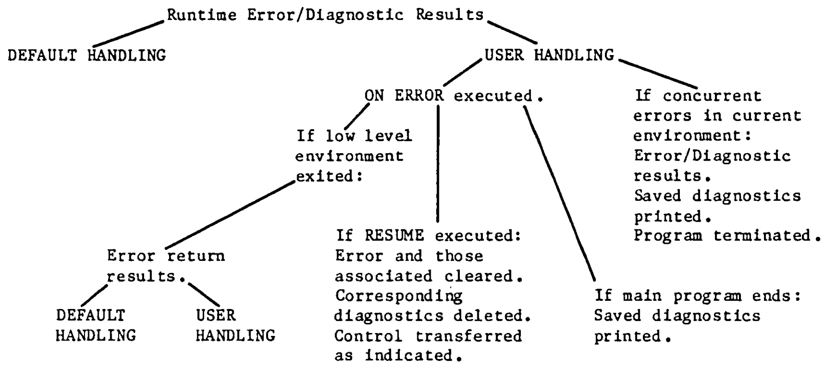
  - The corresponding diagnostics are deleted.

  - Control is transferred to one of three places, depending on the specific form of the RESUME statement.

- The program terminates with an uncleared error. If the program terminates before an error is cleared, the diagnostics that have not been deleted are written to the NOS/VE standard file $ERRORS.

- An error return occurs. If control returns from an environment (by an EXIT FUNCTION, END FUNCTION, EXIT SUB, or END SUB statement) without clearing an error, an error return occurs. Error handling now continues in the calling routine. The next action depends on whether default or user error handling is in effect in the new environment.

User Error Handling Process Model


Runtime Error/Diagnostic Results

DEFAULT HANDLING

USER HANDLING

ON ERROR executed.

If low level environment exited:

Error return results.

DEFAULT HANDLING

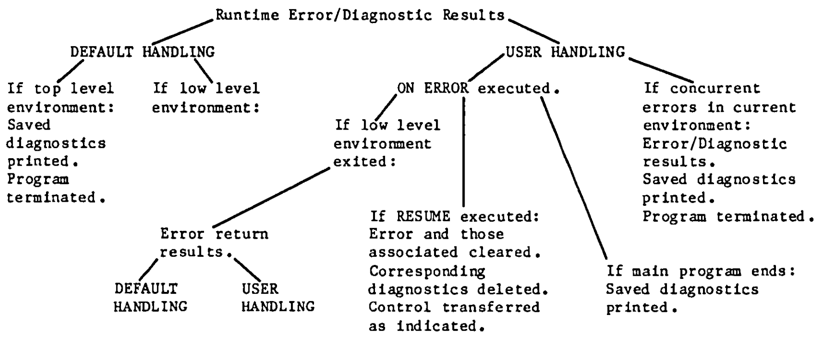
USER HANDLING

If RESUME executed: Error and those associated cleared. Corresponding diagnostics deleted. Control transferred as indicated.

If main program ends: Saved diagnostics printed.

If concurrent errors in current environment: Error/Diagnostic results. Saved diagnostics printed. Program terminated.

Error Processing Model


Runtime Error/Diagnostic Results
DEFAULT HANDLING
If top level environment: Saved diagnostics printed. Program terminated.
If low level environment:
Error return results.
DEFAULT HANDLING
USER HANDLING
USER HANDLING
ON ERROR executed.
If low level environment exited:
If RESUME executed: Error and those associated cleared. Corresponding diagnostics deleted. Control transferred as indicated.
If main program ends: Saved diagnostics printed.
If concurrent errors in current environment: Error/Diagnostic results. Saved diagnostics printed. Program terminated.

Sample Error Processing

```
   REM Main Program
     :
10 ON ERROR GOTO 40
20 CALL A
     :
   SUB A
     :
30 ***** 'Error #1
   'Error Return
     :
   END SUB 'A
40 'Error Handling
     :
50 RESUME NEXT
     :
```

Error/Diagnostic #1 results at the line labeled 30.
Under default handling, an error return (error #2) occurs as control passes to the line labeled 20.

Under user handling in the new environment, which was activated by the ON ERROR statement (labeled 10), control is transferred to the line labeled 40.
When the RESUME statement (labeled 50) is executed, error #2 and the associated error #1 are cleared. The corresponding diagnostics are deleted.

Control passes to the statement following the one that caused the final error in the series. That is, control passes to the line following the line labeled 20.

ERL Function

Purpose

Returns the label associated with the statement whose execution caused the error in the current environment.

Format

ERL

ERL has no parameters. The value returned is always an integer. If no error exists in the current environment, the default value 0 is returned.

Remarks

If a program has no labels, every statement is associated with the default value 0. The ERL function always returns the value 0, whether an error exists or not.

Examples

- Suppose A receives the value 0.0 through the INPUT statement. The resulting division by zero in the line labeled 30 causes an error. The value 30 (the label associated with the IF-THEN statement) is returned by the ERL function reference and printed.

```
    ON ERROR GOTO 70
    INPUT A
30  IF B / A > 0 THEN 500
    .
    .
70  PRINT ERL
```

- In this example, the error is division by zero. The line causing the error has no label associated with it and the ERL function returns the value 0.

```
    ON ERROR GOTO 70
    A = 0 : B = 5
    IF B / A > 0 THEN 500
    .
    .
70  PRINT ERL
```

- In the example, the error is also division by zero. The ERL function returns the value 20 which is the statement label of the line nearest the preceeding label to the line causing the error.

```
    ON ERROR GOTO 70
    A = 0
20  B = 5
    IF B / A > 0 THEN 500
    .
    .
70  PRINT ERL
```

ERR Function

Purpose

The ERR function returns the number that identifies the uncleared error (if any) in the current environment.

Format

ERR

ERR has no parameters. The value returned is always an integer. The value 0 is returned if no error exists in the current environment.

Remarks

- If an error in the current environment:
  - Is a NOS/VE BASIC runtime error, the ERR function returns the value of the 4-digit status condition code for the error.
  - Was induced with the ERROR statement. The ERR function returns the value of the error number specified in the ERROR statement.
- After an error in the current environment is cleared with the RESUME statement, the ERR function returns the value 0.

Examples

- Suppose A receives the value 0.0 through the INPUT statement. The resulting division by zero in the line labeled 30 causes an error. The ERR function reference returns the value 5003, the status condition code for a divide fault. This value is printed.

```
    ON ERROR GOTO 70
    INPUT A
30  IF B / A > 0 THEN 500
    :
70  PRINT ERR
```

- The error was induced in this example using the ERROR statement at label 20. The next ERR function reference at label 70 returns the value 1 which is the error number specified in the ERROR statement.

```
    ON ERROR GOTO 70
10  LET A = 10
20  IF A = 10 THEN ERROR 1
    .
    .
70  PRINT ERR
```

Runtime Diagnostic Format

Purpose A NOS/VE BASIC runtime diagnostic has the format:

Format

```
--ZZZ--  ERR = ###, ERL = NNN in module ***: error
description
```

ZZZ	Replaced by FATAL or CATASTROPHIC, depending on the severity of the error. A catastrophic error cannot be cleared with user error handling.
###	Replaced by the status condition code identifying the error. This code is a 4-digit integer of the form xxxx.
NNN	Replaced by the NOS/VE BASIC line label associated with the statement that caused the error.
***	Replaced by the name of the external routine containing the error. The name $MAIN is specified for the main program.
error description	Provides a brief description of the error that occured.

Remarks For a complete listing of the NOS/VE BASIC runtime error diagnostics, see the NOS/VE Diagnostic Messages manual.

Examples

- This short program causes the runtime diagnostic below to be issued.

```
10   LET A = -2
20   LET B = A^(0.5)
    END PROGRAM
```

```
--FATAL--  ERR = 5251, ERL = 20 in module $MAIN:
Negative number raised to nonintegral power.
```

- In this short program, label 20 is omitted and the associated label 10 appears in the diagnostic below. If a program has only a few labels, the associated label (or default value 0) provided in a runtime diagnostic does less to pinpoint the location of the error.

```
10  LET A = -2
LET B = A^(0.5)
END PROGRAM
```

```
--FATAL--  ERR = 5251, ERL = 10 in module $MAIN:
Negative number raised to nonintegral power.
```

ON ERROR Statement

Purpose

Specifies where control is to be transferred if a runtime error occurs in the current environment.

Format

```
ON ERROR  GOTO  0
ON ERROR  GOTO  label
```

0	Default error handling is activated.
label	Label of the line to which control is transferred if an error occurs in the current environment.

Remarks

- If 0 is specified, default error handling is activated, overriding any user error handling that is in effect because of a previously executed ON ERROR statement.

- If a label is specified, user error handling is activated, and an error causes control to pass to the line with the specified label.

Examples

User error handling is activated by the ON ERROR statement. If DIVISOR receives the value 0 through the INPUT statement, the resulting division by zero on the next statement causes an error. Control passes to the line labeled 300 for user error handling.

```
ON ERROR GOTO 300
INPUT DIVISOR
LET RECIPROCAL = 1 / DIVISOR
```

Ideally, the statements executed after a branch with the ON ERROR statement either eliminate or bypass any problems resulting from the error that has occurred. Note that these statements can simply ignore the error, although this could cause further problems.

The ERL and ERR library functions are provided as aids to error handling. These functions help determine the location and cause of an error.

RESUME Statement

Purpose	Clears the error and then transfers control to the specified statement.

Format

```
RESUME
RESUME  0
RESUME  NEXT
RESUME  label
```

0	Control returns to the statement that caused the error in the current environment. This statement is reexecuted.
NEXT	Control passes to the statement following the one that caused the error in the current environment.
label	Label of the line to which control is transferred.

Remarks

- The RESUME statement:
  - Clears the runtime error in the current environment, and all of its associated errors.
  - Deletes the corresponding diagnostics without printing them.
  - Transfers control as indicated by the specific form used.
- A runtime error results if a RESUME statement is executed when no error exists in the current environment.

Examples

Suppose the function reference G(X) in the line labeled 150 causes an error. Control passes to the line labeled 300 and user error handling begins. The RESUME statement clears the error, deletes the corresponding diagnostic, and transfers control as specified by ***. If 0 is specified, control returns to the line labeled 150. An infinite loop could result if the error handling does not eliminate the problem. If NEXT is specified, control passes to the line labeled 160. If a label is specified, control transfers to that label.

```
     ON ERROR GOTO 300
150  IF G(X) > 7.0 THEN END ' Causes an error.
160  REM Branch to here if RESUME NEXT is executed.
       :
300  REM Begin user error handling.
       :
     RESUME ***
```

ERROR Statement

Purpose Simulates the occurrence of a specified error.

Format ERROR errnum

errnum Numeric expression whose value, when rounded to the nearest integer, specifies the number of the error to be simulated.

Remarks If the specified error number is a 4-digit status condition code for a NOS/VE BASIC runtime error, the corresponding error is induced. Otherwise, an error/diagnostic results because an unrecognized error has occurred. In either case, a subsequent ERR function reference returns the specified error number.

Examples

- This simulates the occurrence of an error that is unknown to NOS/VE BASIC. A subsequent ERR function reference returns the value 1.

```
ERROR 1
```

- This simulates the occurrence of the error resulting from division by zero. A subsequent ERR function reference returns the value 5003.

```
ERROR 5003
```

- In this program, error 0001 is induced in the line labeled 100. Error/Diagnostic 0001 results. User error handling calls subroutine HANDLE. Error 0002 is induced in the line labeled 220. Error/Diagnostic 0002 results. User handling in the new environment clears error 0002, deletes diagnostic 0002, and passes control to the EXIT SUB statement. Control then passes to the END statement and the program terminates. Since error 0001 was never cleared, diagnostic 0001 is written to the NOS/VE standard file $ERRORS.

```
100 ON ERROR GOTO 150 : ERROR 0001
150 CALL HANDLE
    END
    SUB HANDLE
220   ON ERROR GOTO 240 : ERROR 0002
      EXIT SUB
240   RESUME NEXT
    END SUB 'HANDLE
```

STOP Statement

Purpose

Stops program execution and returns control to the environment or utility from which you executed the BASIC program.

Format

STOP

The STOP statement can occur any number of times in a program.

Example

The following example shows how to use the STOP statement.

```
PRINT " Enter C to continue, S to stop."
INPUT A$
IF (A$ = "S") or (A$ = "s") THEN STOP
    .
    .
    .
```

User-Defined Functions 7

This chapter describes the two types of user-defined functions: expression functions and block functions.

User-Defined Functions 7

A user-defined function is a procedure that returns a single value.

This chapter describes the two kinds of NOS/VE BASIC user-defined functions: expression functions and block functions. Expression functions are simple single-statement functions. Block functions are more powerful multi-statement structures.

Function Overview

A function is a procedure that returns a value to the place in an expression where the procedure was called. The returned value is usually computed from the values of actual parameters, which are supplied when the function is called.

Every user-defined function has two components:

- The function specification.
- The function body.

The function specification stipulates that a function is being defined and provides a function name. A list of formal parameters might also be included.

The function body computes the returned value.

A formal parameter is a variable or array that acts as a placeholder for an actual parameter. A formal parameter is used within the function body to show how the corresponding actual parameter is involved in producing the returned value.

Any formal parameters used in the function body are also listed in the function specification component. The formal and actual parameter lists must be in one-to-one correspondence. The number of parameters is limited only by the NOS/VE maximum line length.

NOTE

The result of a lack of correspondence between formal and actual parameter lists depends on the specific case. Possible results include a compile-time error, a runtime diagnostic that seems inappropriate because it comes from the loader, or incorrect computations without notification.

A function is called by referencing its name and providing a list of actual parameters (if any). Each actual parameter is passed to its corresponding formal parameter in the function body, where it can be used in computing the returned value.

The specific manner in which an actual parameter is passed to a formal parameter is referred to as parameter passing.

For block functions, parameter passing style is important because it determines whether a change to a formal parameter affects the corresponding actual parameter.

The returned value is substituted for the function reference that made the call. This value has the same data type as that of the function name.

Expression Functions

Purpose	Specifies the function name, optional parameter list and function body.

Format

```
DEF  funname  fplist  =  exp
```

funname	Identifier naming the function. The data type of the function name establishes the data type of the returned value.
fplist	Optional formal parameter list whose format is discussed below.
exp	Expression whose value is returned by the function. This expression can contain both formal parameters and other variables. Its value must be compatible with the data type established by the function name.

A formal parameter list for an expression function has the format:

```
(fp1 , fp2 , ... , fpN)
```

fpJ	Variable denoting the Jth formal parameter, where (1 <= J <= N). The data type of this formal parameter must be compatible with the data type of the corresponding actual parameter. However, an integer value can be passed to a real formal parameter. A real value can be passed to an integer formal parameter.

Format (cont)	An expression function is called by referencing its name and providing an actual parameter list (if appropriate). The returned value is substituted for the function reference and has the data type of the function name.

An expression function reference has the format:

```
funname  aplist
```

funname	Name of the function.
aplist	Optional actual parameter list used only if a formal parameter list appears in the function specification statement.

An expression functions actual parameter list has the format:

```
(ap1 , ap2 , ... , apN)
```

apJ	Expression denoting the Jth actual parameter, where (1 <= J <= N). The value of this expression must be compatible with the data type of the corresponding formal parameter.

Remarks

- The DEF statement defining an expression function must be executed before the function can be called.

  An expression function:

  - Cannot be defined recursively.
  - Can be used to define other expression functions, provided each function is defined before it is first referenced.
  - Cannot be passed a whole array as an actual parameter.
  - Is known only to the external routine that contains it.

Examples

- The expression function TRAP.AREA computes the area of a trapezoid from the height H and the lengths of the bases B1 and B2. The function reference in the PRINT statement returns the value 36.0.

```
DEF TRAP.AREA(H,B1,B2) = 0.5*H*(B1 + B2)
PRINT TRAP.AREA(4.0,10.0,8.0)
```

- The expression function CHAPTER$ constructs a chapter heading from the chapter number N% and the chapter title S$. The library function STR$ is used in the string construction. The function reference in the PRINT statement returns the value " 8. INVESTMENT STRATEGY".

```
DEF CHAPTER$(N%,S$) = STR$(N%) + ". " + S$
PRINT CHAPTER$(8,"INVESTMENT STRATEGY")
```

- The expression function FNA has no parameters. The function output is computed using the current value of X.

```
DEF FNA = X^2 + SIN(X)
```

- This program fragment prints the value 5.35.

```
DEF FNA(X) = 10.0*X^3 + 4.0
LET A(1) = 0.3
PRINT FNA(A(1))
```

NOTE

It is harmless to have a formal parameter in an expression function with the same name as another variable in your program. Changing one will not change the other. Also, different expression functions within the same external routine can use the same formal parameter names.

Block Function Structure

A block function is a multi-statement user-defined function whose function body is a block. A routine can supply data to a block function through parameters. In addition, data can be shared between a routine and a block function through variables that are accessible to both routines.

Unlike an expression function, a block function can be defined recursively and can take a whole array as an actual parameter.

This section describes block function structure.

Block Function Specification

Purpose	Specifies that a function is being defined, provides a function name, and lists the formal parameters (if any).

Format

```
EXTERNAL FUNCTION funname fplist
```

EXTERNAL	Optional keyword used only to specify an external function. If omitted, the function specified is internal.
funname	Identifier naming the function. The data type of the function name establishes the data type of the returned value.
fplist	Optional formal parameter list whose format is discussed below.

A formal parameter list for a block function has the format:

```
(fp1 , fp2 , ... , fpN)
```

fpJ	Variable or formal array (defined below) denoting the Jth formal parameter, where (1 <= J <= N). The data type of this formal parameter must be the same as that of the corresponding actual parameter. An integer value cannot be passed to a real formal parameter. A real value cannot be passed to an integer formal parameter.

Remarks

A formal array is an array name followed by parentheses that contain zero or more commas. The number of dimensions is one more than the number of commas supplied. The formal array dimension bounds are established by the actual array being passed to it.

Examples

- An external block function named LIST$ is specified. Its only formal parameter is a two-dimensional string array. LIST$ returns a result of type string.

  ```
  EXTERNAL FUNCTION LIST$(S$(,))
  ```

- An internal block function named SAMPLE is specified. There are two parameters: integer variable and one-dimensional real array. SAMPLE returns a result of type real.

  ```
  FUNCTION SAMPLE(N%,B())
  ```

- An internal block function with no parameters specified.

  ```
  FUNCTION NO.PARAMETERS
  ```

Block Function Body

Purpose	Contains the statements that perform the tasks for the calling routine and compute the returned value. The body of a block function follows the function specification statement. Within the function body, the value to be returned is assigned to the function name.
Format	LET funname = xxx

funname	Name of the function.
xxx	Expression whose value must be compatible with the data type established by the function name.

Remarks	If no such assignment is made, the default initial value (zero or the null string, as appropriate) is returned, and a warning is issued. As in all assignment statements, the keyword LET is optional.

END FUNCTION Statement

Purpose	Designates the physical end of the function, and follows the function body. Every block function must end with an END FUNCTION statement.
Format	END FUNCTION The END FUNCTION can appear only once in a block function.
Remarks	• The END FUNCTION statement for an external function must be the last statement of the routine's last line. • The END FUNCTION statement transfers control to the function reference which made the function call. The returned value is then substituted for the function reference. • A runtime error results if a block function is exited while it contains an uncleared error. • For more information about clearing runtime errors, see chapter 6.

EXIT FUNCTION Statement

Purpose Transfers control to the function reference which made the function call. The returned value is then substituted for the function reference.

Format

```
EXIT FUNCTION
```

The EXIT FUNCTION can appear any number of times within a function body.

Remarks

- A runtime error results if a block function is exited while it contains an uncleared error.
- For more information about clearing runtime errors, see chapter 6.

Examples

The following shows an example using the EXIT FUNCTION statement.

```
EXTERNAL FUNCTION SAMPLE$(N)
' This function converts positive integer arguments
' to single-letter codes using modulo arithmetic.
'
' --------------- The Function Body ---------------
'
  DEFINT N,X
' Return the null string if input is nonpositive.
  IF N <= 0 THEN LET SAMPLE$ = "" : EXIT FUNCTION
' Convert input to range 0 through 25.
  LET X = (N + 25) MOD 26
' Convert to ASCII range code for uppercase letters.
  LET Y = X + 65
' Return string containing single-letter.
  SAMPLE$ = CHR$(Y)
' -------------- End of Function Body --------------
END FUNCTION
```

External vs. Internal Functions

Block functions are classified as either external or internal. An external function:

- Is an external routine that performs tasks for a calling routine and returns a single value.
- Is declared to be external by including the keyword EXTERNAL in the function specification statement.
- Can be compiled as a separate program unit.
- Cannot be contained within another external routine, but can contain embedded internal routines.
- Shares data with other external routines through the COMMON statement (the next topic) or the passing of parameters.

Declarative statements in an external function apply to all embedded internal routines.

An internal function:

- Is an internal routine that performs tasks for a calling routine and returns a single value.
- Is declared internal, by default, when the keyword EXTERNAL is omitted from its subroutine specification statement.
- Cannot be compiled as a separate program unit.
- Must be contained within a host external routine, and cannot contain embedded routines.
- Has access to all the data of its host external routine.

Declarative statements within an internal function apply to the entire host external routine.

The external and internal classifications apply only to routines.

Examples

- An external function named SAMPLE% is specified. It has three formal parameters: integer variable, real variable, and two-dimensional string array. SAMPLE% returns a result of type integer.

  ```
  EXTERNAL FUNCTION SAMPLE%(N%,X,T$(,))
  ```

- An internal function named TEST is specified. It has two formal parameters: one-dimensional integer array and integer variable. TEST returns a result of type real.

  ```
  FUNCTION TEST(S%(),Y)
  ```

COMMON Statement

Purpose

Allows scalar variables and arrays to be shared among external routines.

Format

COMMON objlist

objlist	Nonempty list of scalar variables and formal arrays that are separated by commas. The listed objects are made accessible to all external routines.

Remarks

- A COMMON statement must precede the first reference to any variable or array that it specifies as a common object.

- A variable that is made accessible to other external routines is not necessarily shared. A variable is shared among a group of external routines only when it appears in a COMMON statement in each of the routines.

- Common arrays that are dimensioned differently in separate external routines will acquire the size specified by the declarations of the first module seen by the loader to which it is visible and the shape declared by the last module seen by the loader to which it is visible. This is an artifact of the way common areas are handled by the loader. Warning errors are likely, but not certain, in the loapmap for such a program.

Examples

- The scalar variables D and C, and the two-dimensional array B are made accessible to all external routines.

```
COMMON D,C,B(,)
```

- Order is not important in a COMMON statement. Thus, the following two statements are equivalent.

```
COMMON X,Y

COMMON Y,X
```

Function Name Declaration

Purpose	Declares names to be those of block functions.

Format

```
DECLARE  EXTERNAL  FUNCTION  fnlist
```

EXTERNAL	Optional keyword EXTERNAL used to declare names as those of external functions. If omitted, specified names are declared to be those of internal functions.
fnlist	List of function names that are separated by commas. Names that appear in this list are declared to be those of block functions.

Remarks

- This statement cannot be used to declare names for expression functions.
- It might be necessary to reference an internal function before it is defined. The function declaration statement makes such a reference possible.

Examples

- This statement designates the name STATS as that of an external function.

```
DECLARE EXTERNAL FUNCTION STATS
```

- This statement designates the names REQUEST and RECEIPT as those of internal functions.

```
DECLARE FUNCTION REQUEST,RECEIPT
```

Block Function Calls

Purpose

A block function is called by referencing its name, and providing an actual parameter list (if appropriate). The value returned is substituted for the function reference, and has the data type of the function name.

Format

funname aplist

funname	Name of the function.
aplist	Optional actual parameter list used only if a formal parameter list appears in the function specification statement.

A block function's actual parameter list has the format:

(ap1 , ap2 , ... , apN)

apJ	Expression or actual array (defined below) denoting the Jth actual parameter, where (1 <= J <= N). The data type of this actual parameter must be the same as that of the corresponding formal parameter. An integer value cannot be passed to a real formal parameter. A real value cannot be passed to an integer formal parameter.

Remarks

- An actual array is an array name followed by parentheses that contains zero or more commas. The number of dimensions is one more than the number of commas supplied. The formal array dimension bounds are established by the actual array being passed to it.

- If execution control reaches the FUNCTION statement of an internal function without using a function call:

  - The statements in the internal function are not executed.

  - Control passes to the statement following the function's END FUNCTION statement. If this END FUNCTION statement is the last statement of the main program, the program is terminated.

Examples

- This reference calls the function SURVEY using the one-dimensional real array FORM as the actual parameter.

  ```
  SURVEY(FORM())
  ```

- For the internal block function below, the function reference:

SPLIT.DEF(1.0,2)	Returns the value 1.5.
SPLIT.DEF(3.0,2)	Returns the value 33.5.

```
   FUNCTION SPLIT.DEF(X,N%)
     RESTORE 90 : READ A,B
     IF X <= N% THEN
       LET SPLIT.DEF = A*X + B
     ELSE
       LET SPLIT.DEF = A*X^2 + B
     ENDIF
 90  DATA 4.0,-2.5
   END FUNCTION
```

Block Function Parameters

Scalar variables or whole arrays can be used as actual parameters. When a change is made to the corresponding formal parameter, the actual parameter is also changed.

Hence, if a block function modifies:

- a formal array
- a formal scalar variable that was passed the value of an actual scalar variable

then the corresponding actual parameter is also modified. For arrays, this includes modifications made with the DIM and ERASE statements.

Actual parameters can also be constants, single array elements, substrings, or nontrivial expressions. When a change is made to the corresponding formal parameter, the actual parameter is not changed.

Block Function Parameters

The presence of parentheses does not protect an actual parameter from modification in the calling routine. The function references F(X) and F((X)) are equivalent.

However, use of a nontrivial expression, such as the one used in the function reference F(X+0.0), does protect the actual parameter from modification in the calling routine.

NOTE

Remember that for block functions, integer values cannot be passed to real formal parameters. Real values cannot be passed to integer formal parameters. For expression functions, the mixing of type integer and type real data is permitted.

If you are using external routines check the load map for errors.

The following similar program fragments shows when actual parameters are modified.

```
'FRAGMENT #1            'FRAGMENT #2
DEFINT A,X              DEFINT A,X
FUNCTION ADD(X)         FUNCTION ADD(X)
  LET X = X + 1           LET X = X + 1
  LET ADD = X              LET ADD = X
END FUNCTION 'ADD       END FUNCTION 'ADD
LET A = 3               LET A(1) = 3
PRINT A, ADD(A), A      PRINT A(1), ADD(A(1)), A(1)
```

In fragment #1, the scalar A is passed to the scalar X. When X is incremented in the function, so is A. The values 3, 4, and 4 are printed.

In fragment #2, the array element A(1) is passed to the scalar X. When X is incremented in the function, A(1) is not altered. The values 3, 4, and 3 are printed.

Mathematical Library Functions 8

The chapter describes the NOS/VE BASIC supplied functions that perform various mathematical operations.

Mathematical Library Functions 8

NOS/VE BASIC provides many library (or built-in) functions.

This chapter describes the mathematical library functions. They have been divided into groups of related functions. The functions within each category are described in alphabetical order. The RANDOMIZE statement is also discussed because it relates to the use of the RND function.

The string library functions are described in the String Processing chapter. Each of the other library functions is discussed in a topic that relates to the specific use of that function.

An alphabetical list of all the NOS/VE BASIC library functions appears in the Library Functions Index appendix. It includes a categorical cross-reference to help you visualize each function in context.

Exponential Functions

The Exponential Functions include exponential, logarithmic, and hyperbolic functions.

COSH Function

Purpose	Returns the hyperbolic cosine of the value of an argument.
Format	COSH(number) number — Numeric expression whose value x can be either integer or real. The magnitude of x must be less than 4095 * LOG(2).
Remarks	The value returned is ((EXP(x) + EXP(-x))/2.0). The result is always real.
Examples	● The following are examples of the COSH function.

COSH(0)	Returns the value 1.0.
COSH(LOG(2.0))	Returns the value 1.25.

EXP Function

Purpose	Returns the power of the irrational number (e) specified by the value of an argument. This function is the inverse of the LOG function.
Format	EXP(number) number Numeric expression whose value x can be either integer or real. The magnitude of x must be less than 4095 * LOG(2) or a runtime error results.
Remarks	The value returned is (e^x). The result is always real.
Examples	• The following are examples of the EXP function. EXP(0) — Returns the value 1.0. EXP(LOG(4.8)) — Returns the value 4.8.

LOG Function

Purpose	Returns the base (e) logarithm of the value of an argument. This function is the inverse of the EXP function.
Format	LOG(number) number Numeric expression whose value x can be either integer or real, and must be positive.
Remarks	The value returned is that number y, such that (e^y = x). The result is always real.
Examples	• The following are examples of the LOG function.

```
LOG(1)              Returns the value 0.0.

LOG( EXP(3.2) )     Returns the value 3.2.
```

LOG10 Function

Purpose	Returns the base ten logarithm of the value of an argument.
Format	LOG10(number)
	number Numeric expression whose value x can be either integer or real, and must be positive.
Remarks	The value returned is that number y, such that (10^y = x). The result is always real.
Examples	• The following are examples of the LOG10 function.

LOG10(1000)	Returns the value 3.0.
LOG10(1.0E5)	Returns the value 5.0.
LOG10(0.01)	Returns the value -2.0.

SINH Function

Purpose	Returns the hyperbolic sine of the value of an argument.
Format	SINH(number) number Numeric expression whose value x can be either integer or real. The magnitude of x must be less than 4095 * LOG(2).
Remarks	The value returned is ((EXP(x) - EXP(-x))/2.0). The result is always real.
Examples	• The following are examples of the SINH function.

SINH(0)	Returns the value 0.0.
SINH(LOG(4.0))	Returns the value 1.875.

TANH Function

Purpose	Returns the hyperbolic tangent of the value of an argument.
Format	TANH(number)
	number Numeric expression whose value x can be either integer or real. There are no restrictions on x.
Remarks	The value returned is (EXP(x) - EXP(-x))/(EXP(x) + EXP(-x)). The result is always real.
Examples	• The following are examples of the TANH function.

TANH(0)	Returns the value 0.0.
TANH(LOG(2.0))	Returns the value 0.6.

Trigonometric Functions

The Trigonometric Functions include trigonometric, inverse trigonometric, and angle conversion functions.

ACOS Function

Purpose	Returns the inverse cosine of the value of an argument. This function complements the COS function.
Format	ACOS(number)
	number — Numeric expression whose value x can be either integer or real. The magnitude of x must be less than or equal to one.
Remarks	The function returns the radian measure of the angle y, with (0 <= y <= PI), whose cosine is x. The result is always real.
Examples	• The following are examples of the ACOS function.

ACOS(1)	Returns the value 0.0.
ACOS(COS(1.5))	Returns the value 1.5.

ASIN Function

Purpose	Returns the inverse sine of the value of an argument. This function complements the SIN function.
Format	ASIN(number)
	number Numeric expression whose value x can be either integer or real. The magnitude of x must be less than or equal to one.
Remarks	The function returns the radian measure of the angle y, with (-PI/2 <= y <= PI/2), whose sine is x. The result is always real.
Examples	• The following are examples of the ASIN function.

ASIN(0)	Returns the value 0.0.
ASIN(SIN(0.5))	Returns the value 0.5.

ATN Function

Purpose	Returns the inverse tangent of the value of an argument.
Format	ATN(number)
	number Numeric expression whose value x can be either integer or real. There are no restrictions on x.
Remarks	Returns the radian measure of the angle y, with (-PI/2 < y < PI/2), whose tangent is x. The result is always real.
Examples	• The following are examples of the ATN function.

ATN(0)	Returns the value 0.0.
ATN(TAN(1.0))	Returns the value 1.0.

COS Function

Purpose	Returns the cosine of the value of an argument. This function complements the ACOS function.	
Format	COS(radians)	
	radians	Numeric expression whose value x is an angle measured in radians, and can be either integer or real. The magnitude of x must be less than 2^47.
Remarks	The value returned is always real.	

Examples

- The following are examples of the COS function.

COS(0)	Returns the value 1.0.
COS(ACOS(0.4))	Returns the value 0.4.
COS(RAD(180))	Returns the value -1.0.

DEG Function

Purpose	Converts the value of an argument from radians to degrees. This function is the inverse of the RAD function.
Format	DEG(radians)

radians	Numeric expression whose value x is an angle measured in radians, and can be either integer or real.

Remarks The value returned is the degree measure of x. The result is always real.

Examples

- The following are examples of the DEG function.

DEG(ACOS(-1.0))	Returns the value 180.0.
DEG(RAD(135))	Returns the value 135.0.

RAD Function

Purpose	Converts the value of an argument from degrees to radians. This function is the inverse of the DEG function.
Format	RAD(degrees)
	degrees — Numeric expression whose value x is an angle measured in degrees, and can be either integer or real.
Remarks	The value returned is the radian measure of x. The result is always real.
Examples	• The following are examples of the RAD function.

```
SIN( RAD(90.0) )    Returns the value 1.0.

RAD( DEG(3.6) )     Returns the value 3.6.
```

SIN Function

Purpose	Returns the sine of the value of an argument. This function complements the ASIN function.
Format	SIN(radians)

radians	Numeric expression whose value x is an angle measured in radians, and can be either integer or real. The magnitude of x must be less than 2^47.

Remarks The value returned is always real.

Examples

- The following are examples of the SIN function.

SIN(0)	Returns the value 0.0.
SIN(ASIN(0.9))	Returns the value 0.9.
SIN(RAD(-90))	Returns the value -1.0.

TAN Function

Purpose	Returns the tangent of the value of an argument. This function complements the ATN function.
Format	TAN(radians)
	radians — Numeric expression whose value x is an angle measured is radians and can be either integer or real. The magnitude of x must be less than 2^47.
Remarks	The value returned is always real.
Examples	• The following are examples of the TAN function.

TAN(0)	Returns the value 0.0.
TAN(ATN(0.3))	Returns the value 0.3.
TAN(RAD(45))	Returns the value 1.0.

Number Characteristic Functions

The Number Characteristic Functions include functions that change numeric data type, and manipulate the whole, fractional, and sign components of numbers.

ABS Function

Purpose	Returns the absolute value of the value of an argument.
Format	ABS(number)
	number — Numeric expression whose value x can be either real or integer.
Remarks	The value x is returned if x is nonnegative. The value (-x) is returned if x is negative. In other words, the value of the argument is made positive. The data type of the result is the same as that of the argument.

Examples

- The following are examples of the ABS function.

ABS(4.5)	Returns the real value 4.5.
ABS(3)	Returns the integer value 3.
ABS(-2)	Returns the integer value 2.
ABS(-3.0	Returns the real value 3.0.

CDBL Function

Purpose	Returns a real representation of the value of an argument. This function is equivalent to the CSNG function.
Format	CDBL(number)
	number — Numeric expression whose value x can be either real or integer.
Remarks	This function converts x to type real.
Examples	• The following are examples of the CDBL function.

CDBL(-5)	Returns the value -5.0.
CDBL(4.7)	Returns the value 4.7.

CEIL Function

Purpose	Returns the smallest integer that is at least as large as the value of an argument.
Format	CEIL(number) number Numeric expression whose value x can be either real or integer.
Remarks	The ceiling value returned is the smallest integer whose location on the real number line is either at, or to the right of x. The result is always integer.
Examples	• The following are examples of the CEIL function.

CEIL(5.9)	Returns the value 6.
CEIL(2.0)	Returns the value 2.
CEIL(-1)	Returns the value -1.
CEIL(-3.2)	Returns the value -3.

CINT Function

Purpose	Returns the value of an argument rounded to the nearest integer.
Format	CINT(number)

number	Numeric expression whose value x can be either real or integer.

Remarks	The value returned is always integer.

Examples

- The following are examples of the CINT function.

CINT(5.7)	Returns the value 6.
CINT(1.5)	Returns the value 2.
CINT(-2.5)	Returns the value -3.
CINT(-8.4)	Returns the value -8.

CSNG Function

Purpose	Returns a real representation of the value of an argument. This function is equivalent to the CDBL function.
Format	CSNG(number)
	number — Numeric expression whose value x can be either real or integer.
Remarks	This function converts x to type real.
Examples	• The following are examples of the CSNG function.

CSNG(-4)	Returns the value -4.0.
CSNG(3.6	Returns the value 3.6.

FIX Function

Purpose	Returns the value of an argument truncated to an integer.
Format	FIX(number)
	number — Numeric expression whose value x can be either real or integer.
Remarks	This function deletes all the digits of x that are to the right of the decimal point. The result is always integer.

Examples

- The following are examples of the FIX function.

FIX(8.9)	Returns the value 8.
FIX(-1.2)	Returns the value -1.
FIX(-3.6)	Returns the value -3.
FIX(-5)	Returns the value -5.

FP Function

Purpose	Returns the fractional part of the value of an argument.
Format	FP(number)
	number — Numeric expression whose value x can be either real or integer.
Remarks	This function returns the digits of x that are to the right of the decimal point. The result is always real. If x is an integer, has no digits to the right of a decimal point, or has a magnitude that is greater than 10^18, then a zero value is returned. A zero value is always returned with no sign. Otherwise, the returned value has the same sign as x does.

Examples

- The following are examples of the FP function.

FP(7.9)	Returns the value 0.9.
FP(5.), FP(-4)	Return the value 0.0.
FP(-6.2)	Returns the value -0.2.

INT Function

Purpose	Returns the greatest integer that is no larger than the value of an argument.
Format	INT(number) number Numeric expression whose value x can be either integer or real.
Remarks	The floor value returned is the largest integer whose location on the real number line is either at, or to the left of x. The result is always integer.
Examples	• The following are examples of the INT function.

INT(5.6)	Returns the value 5.
INT(-3)	Returns the value -3.
INT(-5.8)	Returns the value -6.
INT(-8.1)	Returns the value -9.

SGN Function

Purpose	Returns an integer that represents the sign of the value of an argument.
Format	SGN(number) number Numeric expression whose value x can be either real or integer.
Remarks	The value 1 is returned if x is positive. The value 0 is returned if x is zero. The value -1 is returned if x is negative. The result is always integer.
Examples	• The following are examples of the SGN function.

SGN(4)	Returns the value 1.
SGN(0.0)	Returns the value 0.
SGN(-5.6)	Returns the value -1.

Miscellaneous Functions

The Miscellaneous Functions include functions not addressed in the previous categories.

MAX Function

Purpose Returns the largest of the values of two arguments.

Format MAX(num1 , num2)

num1, num2	Numeric expressions whose values can be either integer or real.

Remarks If either num1 or num2 is real, the value returned is real. If both num1 and num2 are integer, the value returned is integer.

Examples

- The following are examples of the MAX function.

MAX(5.0,3.2)	Returns the real value 5.0.
MAX(-4,-5.2)	Returns the real value -4.0.
MAX(-8,-2)	Returns the integer value -2.

MIN Function

Purpose	Returns the smallest of the values of two arguments.
Format	MIN(num1, num2)
	num1, num2 Numeric expressions whose values can be either integer or real.
Remarks	If either num1 or num2 is real, the value returned is real. If both num1 and num2 are integer, the value returned is integer.
Examples	• The following are examples of the MIN function.

MIN(5.0,3.2)	Returns the real value 3.2.
MIN(-4,-5.2)	Returns the real value -5.2.
MIN(-8,-2)	Returns the integer value -8.

RND Function

Purpose	Returns a random number between zero and one, exclusive.
Format	RND RND(seed)
	seed — Numeric expression whose value x is a seed for the random number generator. This value can be either integer or real.
Remarks	If called without an argument, or if x is positive, the next value of the pseudo-random sequence is returned. If x is zero, the most recently returned value is repeated. If x is negative, the random number generator is reseeded. A given negative value always produces the same pseudo-random sequence.

SQR Function

Purpose	Returns the principal square root of the value of an argument.
Format	SQR(number)
	number Numeric expression whose value x can be either integer or real, and must be nonnegative.
Remarks	The value returned is (x^0.5). The result is always real.
Examples	• The following are examples of the SQR function.

SQR(9)	Returns the value 3.0.
SQR(16.0)	Returns the value 4.0.
SQR(1.44)	Returns the value 1.2.

RANDOMIZE Statement

The NOS/VE BASIC random number generator produces a sequence of numbers that appears to be randomly generated.

After the initial random number is generated, each subsequent number is derived from the previous. For this reason, these numbers are more accurately referred to as pseudo-random numbers. Billions of such numbers are generated before the sequence repeats.

By default, the same pseudo-random sequence is generated each time a program is run. To create a different sequence, you must specify a seed. This numeric value generates a different initial random number, thereby producing a new sequence. However, a given seed always produces the same pseudo-random sequence.

The random number generator can be reseeded using either the RANDOMIZE statement or the RND function.

RANDOMIZE Statement

Purpose — Reseeds the random number generator. This statement has the format:

Format

```
RANDOMIZE  seed
```

seed — Optional numeric expression whose value x is the seed for the random number generator, and can be either integer or real.

Remarks

- If SEED is omitted in interactive mode, the system asks you to input a seed value. The following prompt is displayed:

  ```
  Random number seed?
  ```

  If SEED is omitted in batch mode, the system reads a line of data from the file $INPUT.

- A given sequence can be repeated by reseeding with a constant numeric expression.

- An easy way to produce a seed that changes with each program run is to use the last two digits of the system supplied variable TIME$. This variable accesses the NOS/VE internal clock. The following RANDOMIZE extracts these two digits, converts them to a numeric value, and reseeds the random number generator with this value.

  ```
  RANDOMIZE  VAL( RIGHT$( TIME$,2) )
  ```

- The random number generator can also be reseeded by calling the RND library function with a negative argument.

Subroutines 9

Subroutines 9

A subroutine is a procedure that handles specific tasks for another routine. The results of these tasks might be needed repeatedly by a single program or commonly needed by many programs.

Code accessed through a GOSUB statement is not considered a subroutine in NOS/VE BASIC. (It is in some earlier versions of BASIC.) The GOSUB/RETURN construct does provide a branch and return, but it does not define a structured program unit nor provide for the passing of information through parameters.

This chapter discusses NOS/VE BASIC subroutines. It describes how FORTRAN or COBOL subprograms can be accessed from within a NOS/VE BASIC program. It also describes how to enable application usage billing based on application units.

Subroutine Overview

A NOS/VE BASIC subroutine is a routine that performs specific tasks for a calling routine. To perform these tasks, a subroutine usually requires the values of actual parameters, which are supplied by the calling routine when the subroutine is called.

A subroutine can return data to the calling routine through the values of actual parameters. In addition, data can be shared between a routine and a subroutine through variables that are accessible to both routines.

Every subroutine has three components:

- The subroutine specification.
- The subroutine body.
- The END SUB statement.

The subroutine specification stipulates that a subroutine is being defined and provides a subroutine name. A list of formal parameters might also be included.

The subroutine body performs the tasks for the calling routine. Any values that are to be returned to the calling routine as actual parameters are computed in the subroutine body.

The END SUB statement designates the physical end of the subroutine.

A formal parameter is a variable or array that acts as a placeholder for an actual parameter.

The role of a formal parameter within the subroutine body depends on the purpose of its corresponding actual parameter. When an actual parameter is used to:

- Supply data to the subroutine from the calling routine, the corresponding formal parameter is used to show how this data is involved in performing the subroutine tasks.
- Return data to the calling routine from the subroutine, the corresponding formal parameter is used to store the value to be returned.

It is possible for a single formal parameter to play both of these roles.

Any formal parameters used in the subroutine body are also listed in the subroutine specification component. The formal and actual parameter lists must be in one-to-one correspondence. The number of parameters is limited only by the NOS/VE maximum line length.

NOTE

The result of a lack of correspondence between formal and actual parameter lists depends on the specific case. Possible results include a compile-time error, a runtime diagnostic that seems inappropriate because it comes from the loader, or incorrect computations without notification. For error checking be sure to check the loap map.

A subroutine is called by referencing its name in a CALL statement and providing a list of actual parameters (if any). When the call is made, each actual parameter is associated with its corresponding formal parameter in the subroutine body. Values passed to the subroutine from the calling routine can then be used in performing the subroutine tasks.

The specific manner in which an actual parameter is passed to a formal parameter is referred to as parameter passing.

Parameter passing is important because it determines whether a change to a formal parameter affects the corresponding actual parameter.

When the subroutine is exited (with an END SUB or EXIT SUB statement), any values to be returned are passed from formal parameters in the subroutine to the corresponding actual parameters in the calling routine.

Subroutine Structure

This section describes subroutine structure.

Subroutine Specification

Purpose	Specifies that a subroutine is being defined, provides a subroutine name, and lists the formal parameters (if any). A subroutine begins with a subroutine specification statement.

Format

```
EXTERNAL SUB subname fplist
```

EXTERNAL	Optional keyword, used only to specify an external subroutine. If omitted, the subroutine specified is internal.
subname	Plain name identifying the subroutine. A subroutine name has no data type associated with it (even if its first letter is referenced in a type declaration statement).
fplist	Optional formal parameter list whose format is discussed below.

A formal parameter list for a subroutine has the format:

```
(fp1 , fp2 , ... , fpN)
```

fpJ	Variable or formal array (defined below) denoting the Jth formal parameter, where (1 <= J <= N). The data type of this formal parameter must be the same as the corresponding actual parameter. An integer value cannot be passed to a real formal parameter. A real value cannot be passed to an integer formal parameter.

A formal array is an array name followed by parentheses that contain zero or more commas. The number of dimensions is one more than the number of commas supplied. Dimension bounds of the formal array are the same as those of the actual array that are passed to the formal array.

Examples

- An external subroutine named QUESTIONS is specified. Its only parameter is a two-dimensional string formal array.

  ```
  EXTERNAL SUB QUESTIONS(T$(,))
  ```

- An internal subroutine named REARRANGE is specified. It has two formal paramters. X is a real variable and R%() is a one-dimensional integer array.

  ```
  SUB REARRANGE(X,R%())
  ```

- An internal subroutine with no parameters is specified.

  ```
  SUB NO.PARAMETERS
  ```

Subroutine Body

The body of a subroutine follows the subroutine specification statement. A subroutine body is a block containing the statements that perform the tasks for the calling routine. Any values that are to be returned to the calling routine as actual parameters are computed in the subroutine body and assigned to the appropriate formal parameters.

END SUB Statement

Purpose

Designates the physical end of the subroutine and follows the subroutine body. Every subroutine must end with an END SUB statement.

Format

END SUB

The END SUB can appear only once in a subroutine.

Remarks

- The END SUB statement for an external and an internal subroutine must be the last statement of the routine's last line.
- The END SUB statement transfers control to the statement following the CALL statement that made the subroutine call. This makes available to the calling routine any returned values.
- A runtime error results if a subroutine is exited while it contains an uncleared error.
- For more information about clearing runtime errors, see chapter 6.

EXIT SUB Statement

Purpose

Transfers control to the statement following the CALL statement that made the subroutine call. This makes available to the calling routine any returned values.

Format

```
EXIT SUB
```

The EXIT SUB can appear any number of times within a subroutine body.

Remarks

- A runtime error results if a subroutine is exited while it contains an uncleared error.
- For more information about clearing runtime errors, see chapter 6.

Examples

The following subroutine shows an example using the EXIT SUB statement.

```
EXTERNAL SUB TRIANGLE(SIDE1,SIDE2,SIDE3,PERIMETER,AREA)
' This subroutine computes the area (using Heron's
' formula) and perimeter of a triangle from the lengths
' of the sides.
'
' --------------- The Subroutine Body --------------
'
  IF (SIDE1 < 0) OR (SIDE2 < 0) OR (SIDE3 < 0) THEN
    LET PERIMETER = 0 : LET AREA = 0 : EXIT SUB
  ENDIF
  LET PERIMETER = SIDE1 + SIDE2 + SIDE3
  LET S = 0.5*PERIMETER
  LET TEMP = S*(S - SIDE1)*(S - SIDE2)*(S - SIDE3)
  LET AREA = SQR(TEMP)
'
' -------------- End of Subroutine Body ------------
END SUB
```

External vs. Internal Subroutines

Subroutines are classified as either external or internal. An external subroutine:

- Is an external routine that performs tasks for a calling routine.
- Is declared to be external by including the keyword EXTERNAL in the subroutine specification statement.
- Can be compiled as a separate program unit.
- Cannot be contained within another external routine, but can contain embedded internal routines.
- Shares data with other external routines through the COMMON statement or the passing of parameters.

Declarative statements in an external subroutine apply to all embedded internal routines.

An internal subroutine:

- Is an internal routine that performs tasks for a calling routine.
- Is declared internal, by default, when the keyword EXTERNAL is omitted from the subroutine specification statement.
- Cannot be compiled as a separate program unit.
- Must be contained within a host external routine, and cannot contain embedded routines.
- Has access to all the data of its host external routine.

Declarative statements within an internal subroutine apply to the entire host external routine.

Examples

- An external subroutine named MATH is specified. It has two parameters. X is a real variable and Y(,) is a two-dimensional real formal array.

  ```
  EXTERNAL SUB MATH(X,Y(,))
  ```

- An internal subroutine named PROCESS is specified. It has three parameters: N%, an integer variable, A, a real variable, and R$(), a one-dimensional string formal array.

  ```
  SUB PROCESS(N%,A,R$())
  ```

COMMON Statement

Purpose

Shares scalar variables and arrays with external routines through the COMMON statement.

Format

COMMON objlist

objlist	Nonempty list of scalar variables and formal arrays that are separated by commas. The listed objects are made accessible to all external routines.

Remarks

- A COMMON statement must precede the first reference to any variable or array that it specifies as a common object.

- A variable that is made accessible to other external routines is not necessarily shared. A variable is shared among a group of external routines only when it appears in a COMMON statement in each of the routines.

Examples

- The scalar variables D and C, and the two-dimensional array B are made accessible to all external routines.

```
COMMON D,C,B(,)
```

- Order is not important in a COMMON statement. Thus, the following statements are equivalent.

```
COMMON X,Y

COMMON Y,X
```

Subroutine Name Declaration

Purpose	Declares names to be those of subroutines.

Format

```
DECLARE  EXTERNAL  SUB  snlist
```

EXTERNAL	Optional keyword EXTERNAL used to declare names as those of external subroutines. If omitted, specified names are declared to be those of internal subroutines.
snlist	List of subroutine names that are separated by commas. Names that appear in this list are declared to be those of subroutines.

Remarks

A subroutine declaration statement that declares an external subroutine must precede the first call to that subroutine.

Examples

- This statement designates the name ADDRESS.LIST as that of an external subroutine.

  ```
  DECLARE EXTERNAL SUB ADDRESS.LIST
  ```

- This statement designates the names CHECK and SCHEDULE as those of internal subroutines.

  ```
  DECLARE SUB CHECK,SCHEDULE
  ```

Subroutine Calls

Purpose	Call a NOS/VE subroutine.

Format

CALL subname aplist

subname	Name of the subroutine.
aplist	Optional actual parameter list used only if a formal parameter list appears in the subroutine specification statement.

An actual parameter list for a subroutine has the format:

(ap1 , ap2 , ... , apN)

apJ	Expression or actual array (defined below) denoting the Jth actual parameter, where (1 <= J <= N). The data type of this actual parameter must be the same as that of the corresponding formal parameter. An integer value cannot be passed to a real formal parameter. A real value cannot be passed to an integer formal parameter.

Remarks

- An actual array is an array name followed by parentheses that contain zero or more commas. The number of dimensions is one more than the number of commas supplied. The formal array dimension bounds are established by the actual array being passed to it.

- If execution control reaches the SUB statement of an internal subroutine without using a CALL statement:

  - The statements in the internal subroutine are not executed.

  - Control passes to the statement following the subroutine's END SUB statement. If this END SUB statement is the last statement of the main program, the program is terminated.

Examples

- This statement calls the subroutine PAYROLL using the two-dimensional string array NAMES$ as the actual parameter.

  ```
  CALL PAYROLL(NAMES$(,))
  ```

- The following CALL statement CALL TRIANGLE (3.0,4.0,5.0,P,A) calls the subroutine below to compute the perimeter and area of a triangle with sides of lengths 3.0, 4.0, and 5.0. The actual parameters P and A receive the returned values 12.0 and 6.0, respectively.

```
EXTERNAL SUB TRIANGLE(SIDE1,SIDE2,SIDE3,PERIMETER,AREA)
' This subroutine computes the area (using Heron's
' formula) and perimeter of a triangle from the lengths
' of the sides.
  IF (SIDE1 < 0) OR (SIDE2 < 0) OR (SIDE3 < 0) THEN
    LET PERIMETER = 0 : LET AREA = 0 : EXIT SUB
  ENDIF
  LET PERIMETER = SIDE1 + SIDE2 + SIDE3
  LET S = 0.5*PERIMETER
  LET TEMP = S*(S - SIDE1)*(S - SIDE2)*(S - SIDE3)
  LET AREA = SQR(TEMP)
END SUB
```

Subroutine Parameters

Actual parameters that are scalar variables or whole arrays can be passed. A dynamically dimensioned array can be passed as a parameter to an external routine compiled with statically dimensioned arrays. A statically dimensioned array can be passed as a parameter to an external routine compiled with dynamically dimensioned arrays. When a change is made to the corresponding formal parameter, the actual parameter is also changed. A runtime diagnostic results if an external routine attempts to redimension a statically dimensioned array parameter.

Actual parameters that are constants, single array elements, substrings, or nontrivial expressions will not be modified even if the corresponding formal parameter is modified.

(A nontrivial expression is one that involves at least one operation or function reference.)

Hence, if a subroutine modifies:

- a formal array
- a formal scalar variable that was passed the value of an actual scalar variable

then the corresponding actual parameter is also modified. For arrays, this includes modifications made with the DIM and ERASE statements. A subroutine compiled with statically dimensioned arrays cannot redimension an array parameter.

If a subroutine modifies a formal scalar variable that is passed:

- a constant
- a single array element
- a substring
- a nontrivial expression

then the corresponding actual parameter is not modified.

The presence of parentheses does not protect an actual parameter from modification in the calling routine. The subroutine calls CALL SUBROUTINE(X) and CALL SUBROUTINE((X)) are equivalent.

However, use of a nontrivial expression, such as the one used in the subroutine call CALL SUBROUTINE(X+0.0), does protect an actual parameter from modification in the calling routine.

NOTE

Remember that for subroutines, integer values cannot be passed to real formal parameters. Real values cannot be passed to integer formal parameters.

The following similar program fragments contrast when actual parameters are modified.

```
'FRAGMENT #1                'FRAGMENT #2
DEFINT A,X                  DEFINT A,X
SUB ADD5(X)                 SUB ADD5(X)
  LET X = X+5                 LET X = X+5
END SUB                     END SUB
LET A = 3      : PRINT A    LET A(1) = 3      : PRINT A(1)
CALL ADD5(A) : PRINT A      CALL ADD5(A(1)) : PRINT A(1)
```

In fragment #1, the scalar A is passed to the scalar X. When X is incremented in the subroutine, so is A. The values 3 and 8 are printed.

In fragment #2, the array element A(1) is passed to the scalar X. When X is incremented in the subroutine, A(1) is not altered. The values 3 and 3 are printed.

CALLX Statement

Purpose	Provides an interface to subroutines written in languages that conform to the FORTRAN calling sequence.

Format

```
CALLX  fsubname  aalist
```

fsubname	Name of the FORTRAN (or other) subroutine being called. The name must be a plain name and cannot contain periods.
aalist	Optional list of actual arguments.

The actual argument list in a CALLX statement has the format:

```
(aa1 , aa2 , ... , aaN)
```

aaJ	Expression or actual array (defined below) denoting the Jth actual argument, where (1 <= J <= N). The data type of this actual argument must be the same as that of the corresponding formal argument in the FORTRAN subroutine. Actual string arrays cannot be passed through the CALLX statement.

Remarks

An actual array is an array name followed by a pair of parentheses that contains zero or more commas. The number of dimensions of the actual array is one more than the number of commas supplied. There is no mechanism by which a FORTRAN (or other) subroutine can alter the dimension bounds of an actual array argument.

Examples

The BASIC program (left half of example below) assigns values to a one-dimensional array, prints the array using the internal subroutine, and then calls a FORTRAN subroutine. The array is passed to the FORTRAN subroutine as an actual parameter.

The FORTRAN subroutine (right half of example below) replaces the Jth array element by the sum of the values of all elements whose subscripts are less than or equal to J, where (1 <= J <= M).

The BASIC program then reprints the array.

```
REM BASIC PROGRAM
DEFINT A,I,L,U : DIM A(-5:5)
LET LB = LBOUND(A) : UB = UBOUND(A)
FOR I = LB TO UB
  LET A(I) = I
NEXT I
CALL ARRAY.PRINT("BEFORE:")
CALLX RUNSUM(A(),UB - LB + 1)
CALL ARRAY.PRINT("AFTER:")
END
SUB ARRAY.PRINT(S$)
  PRINT S$ : PRINT
  FOR I = LB TO UB
    PRINT A(I);
  NEXT I
  PRINT : PRINT
END SUB
END PROGRAM
```

```
C      FORTRAN SUBROUTINE
       SUBROUTINE RUNSUM(A,M)
       INTEGER A(M)
       DO 10 J = 2,M
       A(J) = A(J) + A(J - 1)
10     CONTINUE
       RETURN
       END
```

Suppose that the binary object programs for the BASIC program and the FORTRAN subroutine are in the $LOCAL files LGO and FLGO, respectively. If the working catalog is $LOCAL, the BASIC program can be executed with the following SCL command.

```
EXECUTE_TASK (LGO,FLGO)
```

The output from this program appears below.

```
BEFORE:

-5 -4 -3 -2 -1  0  1  2  3  4  5

AFTER:

-5 -9 -12 -14 -15 -15 -14 -12 -9 -5  0
```

Unit-measured Application Accounting

The BCPDAUA subroutine enables application usage billing based on application units.

You, the programmer, define the units to measure. For example, you might want to measure the number of calls to a particular function. In that case, the function call is an application unit.

To count application units, you first set up an array of integers. Each element of the array represents a unit to be counted. You then call the BCPDAUA subroutine to tell NOS/VE the location of the array of counters.

As the program executes, you update the array. For example, if the first counter represents a call to a particular function, every time that function is called, you increment the first counter.

When your task terminates, NOS/VE accesses the array and emits the values to the job account log as an application unit statistic.

BCPDAUA Subroutine

Purpose: Begins the process of counting application units by telling NOS/VE the location of the array of integers.

Format: CALLX BCPDAUA (array, size, status)

array	A single-dimension array of 1 to 63 integers. All elements in the array must be zero or positive. Each element represents an event to be counted while the program is executing, such as a call to a particular function.
size	An integer from 1 to 63 specifying the size of the array.
status	String variable to receive the status resulting from this CALLX BCPDAUA statement. The string variable must be 256 characters in length; you must set all 256 characters to blank before calling BCPDAUA.

If a status of "NO ERROR." is returned, there were no errors. Otherwise, the status contains the complete error message.

Remarks

- The BCPDAUA call must be in the program unit for which application units are recorded.

- When you call the BCPDAUA subroutine, BCPDAUA in turn calls the CYBIL procedure CLP$DEFINE_APPLIC_UNIT_ARRAY. Application usage billing is based on the CYBIL statistic AVC$APPLICATION_UNITS (AV11). For details on application accounting and the CLP$DEFINE_APPLIC_UNIT_ARRAY procedure, see the CYBIL System Interface and the NOS/VE Accounting Analysis System manuals.

- You can use the Debug utility to execute a program containing a CALLX BCPDAUA statement.

- The message text of any error is returned as the status. The ON ERROR statement does not detect errors generated by the BCPDAUA subroutine.

Examples

The following example shows how to call the BCPDAUA subroutine. The NOS/VE Accounting Analysis System manual describes how to display the resulting statistics.

```
option base 1
dim application.array%(3)
let stat$ = space$(256)
callx bcpdaua (application.array%(),3,stat$)
if mid$(stat$,1,9) <> "NO ERROR." then
  print stat$
endif
application.array%(2) = 2
```

Input and Output 10

Input and Output 10

Data that is supplied to a program for processing is called input. Data that is printed or stored as a result of program execution is called output.

The input and output processes are collectively referred to as Input/Output, abbreviated I/O.

This chapter discusses how a NOS/VE BASIC program receives input from the terminal, accesses input from an interior data set, and sends output to the terminal.

The discussion is based on the assumption that the default connections for the standard files $INPUT and $OUTPUT are the NOS/VE local files INPUT and OUTPUT. For interactive mode, this means that input is received from the terminal, and output appears at the terminal.

Specific details concerning I/O operations for arrays and files appear in the Arrays and Files chapters.

Interactive Input

Data that is supplied to a program from the terminal during run time is called interactive input.

This section discusses the two BASIC statements which provide for interactive input.

INPUT Statement

Purpose	Inputs data into an executing program from the terminal.
Format	INPUT ; prompt varlist

;	Optional semicolon, which serves no purpose in NOS/VE BASIC. This option is provided for compatibility with popular microcomputer versions of BASIC.
prompt	Optional message that can be used to prompt the user for input.
varlist	List of variables that are separated by commas. This input variable list contains the variables that receive values from the terminal.

If the PROMPT parameter is omitted, the system supplies the string "? " when an INPUT statement is executed. This default prompt indicates that data is expected. You can specify a more elaborate prompt with the PROMPT parameter. There are two formats:

```
prompt ;
prompt ,
```

prompt	Quoted string constant containing the message you want printed.
;	Appends the system prompt "? " to the message you have provided.
,	Specifies that the system prompt should not be appended to the message.

Remarks

- Only the first 31 characters of a user prompt (message combined with optional system prompt) is displayed.

- You can enter constants when the input prompt appears at the terminal. Commas are used to separate values. All the data for a given INPUT statement is entered after the prompt.

- The number of characters that you can enter on an input line interactively after the prompt cannot exceed 128 characters. Commas used to separate data items and spaces within quoted strings are counted as part of the 128 characters. An attempt to enter more than 128 characters results in the message:

  ```
  Error in INPUT reply.  Please respecify.
  ```

- When you press RETURN, the values in the prompt line are assigned to their corresponding variables in the input variable list. There must be a one-to-one correspondence between values in the input reply and variables in the input variable list.

- A numeric variable can only be assigned a numeric value. Mixing of integer and real data types is handled exactly as it is handled in an assignment statement. Thus, an integer input for a real variable is converted to type real. A real input for an integer variable is rounded to the nearest integer.

- If a data value in a reply to an INPUT statement begins with a quote, the value is assumed to be a quoted string constant.

- When a string begins with a quote (") it must also contain a closing quote. The actual string is between the quotes. If there is no quote at the beginning, a quote within a string is treated as part of the string.

- Modified unquoted string constants (defined below) can be entered as interactive input.

Remarks (cont)

- A modified unquoted string constant is an unquoted string constant that can contain an apostrophe or a colon. Since the compiler never sees an input reply, the restriction that these two characters be used only as delimiters (when outside of a quoted string) can be relaxed.

- A comma in the data supplied in response to an input prompt is interpreted as a separator. For example, if the response to the statement INPUT a$,b$ is only a comma (,), the response is interpreted as two null strings (a$ and b$) separated by a comma.

- You can enter a carriage return as an acceptable null statement response to an INPUT statement requiring only one data item. If more than one data item is expected, as with INPUT a$,b$, then a carriage return results in the error:

  ```
  Error in INPUT reply.  Please respecify.
  ```

- If erroneous data is entered (including too few or too many values), NOS/VE BASIC attempts to recover. The prompt:

  ```
  Error in INPUT reply.  Please respecify.
  ```

  is issued, and the system waits for the entire input reply to be reentered.

Examples

- In each example below, the second line shows what is displayed at the terminal when the INPUT statement in the first line is executed. Trailing spaces of the prompt will be included.

```
INPUT A(1),F$(2:5),Z
?

INPUT "HOW MANY TRIALS"; NUMBER.OF.TRIALS
HOW MANY TRIALS?

INPUT "ENTER NAME AND ID NUMBER:  ", N$,ID%
ENTER NAME AND ID NUMBER:

INPUT; "ENTER TWO POSITIVE INTEGERS:  ", J%,K%
ENTER TWO POSITIVE INTEGERS:
```

Examples (cont)

- The following example shows the value on the right.

```
INPUT A,B,C
? 100,,200              B = 0
INPUT A$,B$
? HELLO,                B$ = null string.
INPUT A$,B$             A$ = A"BC
? A"BC,ABC"             B$ = ABC"
```

Consider the following interactive sessions.

- The string "GRADE POINT AVERAGE" is assigned to T$. The value 3.41 is assigned to X.

```
INPUT T$,X
? "GRADE POINT AVERAGE", 3.41
```

- The string "MAMA´S FAMOUS PIZZA: VARIETY #" is assigned to Q$. The value 4 is assigned to N%.

```
INPUT Q$,N%
? MAMA´S FAMOUS PIZZA:  VARIETY #,4
```

- The string "10. CHAPTER" is assigned to W$.

```
INPUT W$
? 10. CHAPTER
```

- Since the commas to separate data items in a string are included in one input buffer, the number of a´s and b´s in the example cannot exceed 127.

```
INPUT A$,B$
? aaaaaaaaaa . . . . . . . . . . . . . . . . . . . aaaa
aaaa . . . . . aaaa,bbbbbbbbb . . . . . . . . . . b
```

LINE INPUT Statement

Purpose	Assigns an entire line of data to a single string variable during interactive input.
Format	LINE INPUT ; prompt strvar

;	Optional semicolon permitted for compatibility with popular microcomputer versions of BASIC.
prompt	Optional message that can be used to prompt the user for input.
strvar	String variable that receives the input line.

If the PROMPT parameter is omitted, the system supplies the string "? " when a LINE INPUT statement is executed. This default prompt indicates that data is expected. You can specify a more elaborate prompt with the PROMPT parameter. There are two formats:

```
prompt ;
prompt ,
```

prompt	Quoted string constant containing the message you want printed in the prompt.
;	Appends the system prompt "? " to the message you have provided.
,	Specifies that the system prompt should not be appended to the message.

Remarks

- Only the first 31 characters of a user prompt (message combined with optional system prompt) is displayed.

- You can enter text when the input prompt appears at the terminal. All the input for a given LINE INPUT statement must be entered on the prompt line. When you press RETURN, the entire input reply is assigned to the variable specified in the LINE INPUT statement.

- An input reply to a LINE INPUT statement has no delimiters. Everything from the end of the prompt to the carriage return is assigned to the variable specified in the LINE INPUT statement.

- Leading and trailing spaces can be included by using the space bar. A quotation mark is treated exactly like any other character, even if it is the first character.

Examples

- In each example below, the second line shows what is displayed at the terminal when the LINE INPUT statement in the first line is executed. Note trailing spaces of the prompt will be included.

```
 LINE INPUT T$(1)
?

LINE INPUT "WOULD YOU LIKE A RECEIPT"; R$
WOULD YOU LIKE A RECEIPT?

LINE INPUT "ENTER MESSAGE HERE:  ", M$
ENTER MESSAGE HERE:

LINE INPUT; "ANSWER:  ", S$
ANSWER:
```

- Assume that the space bar is pressed one time after the last L of STARGELL is typed, and then a carriage return is issued. The following interactive assignment is equivalent to the statement below.

```
LINE INPUT "PLAYER'S NAME:  ", P$
PLAYER'S NAME:  "POPS" STARGELL
LET P$ = """POPS"" STARGELL "
```

Note the existence of one trailing space.

Interior Data Sets

Interactive input requires your active involvement during the execution of a program. Instead, you might prefer to have data supplied to a program from a data collection that is stored within the program itself. Each data statement is composed of one or more data items that is limited by the BASIC source line length of 255 characters.

This section describes how input can be supplied to a BASIC program from an interior data set.

DATA Statement

Purpose

Stores an interior data set within an external routine through one or more DATA statements.

Format

DATA datalist

datalist	List of constants that are separated by commas. Unquoted string constants can be included.

Remarks

- The DATA statements in an external routine form a single interior data set. However, these statements need not be grouped on consecutive lines.

- Each value in an interior data set must have a representation that is compatible with the data type of the variable that receives the value.

- If a value in a DATA statement begins with a quotation mark, the value is assumed to be a quoted string constant since an unquoted string constant cannot begin with a quotation mark.

- A value in the interior data set is accessed through the READ statement.

- A DATA statement can provide null values to a READ statement. (A null value assigns a 0 value to an integer or real variable or the null string to a string variable.)

  A DATA statement provides a null value when it specifies no data or a separator (,) without data. For example, the following DATA statements each supply one null value:

  ```
  DATA
  DATA ,10
  DATA 10,,20
  DATA 10,20,
  ```

Examples

- The following are examples of the DATA statement.

```
DATA -3.2,8,STOP,1.23E5,-32,"START:FINISH"

DATA "I DON'T KNOW",I DO NOT KNOW,"X,Y, OR Z"

DATA JULIUS "DR. J" ERVING,"""MAGIC"" JOHNSON"
```

- The following example demonstrates the use of null values from DATA statements. The first DATA statement provides a null value for variable A and the second DATA statement provides null values for B, C, and D.

```
DATA
DATA,,
READ A, B, C, D
```

READ Statement

Purpose — Assigns a value to a variable from the interior data set of an external routine through the READ statement.

Format — READ varlist

varlist	List of variables that are separated by commas. This input variable list contains the variables to be assigned values from the interior data set.

Remarks

- The values to be assigned to variables in the variable list appear in DATA statements. Values are assigned in sequential order, starting with the first value in the first DATA statement. A pointer keeps track of the next available value.

- Each time a value is read, the pointer advances one item in the interior data set. When all the values in a given DATA statement have been exhausted, the pointer moves to the next DATA statement in the external routine.

- A runtime error results if the number of values remaining in the interior data set is too few to satisfy an input variable list.

- A numeric variable can only be assigned a numeric value. Mixing of integer and real data types is handled exactly as it is handled in an assignment statement. Thus, an integer input for a real variable is converted to type real. A real input for an integer variable is rounded to the nearest integer.

Examples — Assume that the pointer is set to the beginning of the DATA statement when the READ statement is executed. The variables A%, B, C$, and D$ are assigned the values 4, 4.0, "4", and "4", respectively.

```
READ A%,B,C$,D$
DATA 4,4,4,"4"
```

The pointer of an interior data set can be reset by the RESTORE statement.

RESTORE Statement

Purpose — Moves the pointer for the interior data set of an external routine to a new DATA statement.

Format — RESTORE label

label	Optional line label identifying the DATA statement to which the pointer is moved.

Remarks — If a label is specified, the pointer moves to the beginning of the first DATA statement associated with the label. If the label is omitted, the pointer moves to the beginning of the first DATA statement in the external routine.

Examples — The following program shows examples of the RESTORE statement.

```
   DEFINT X - Z
   READ XA, XB, XC, XD
   RESTORE
   READ YA, YB, YC, YD, YE, YF
   RESTORE 10
   READ ZA, ZB
   PRINT XA; XB; XC; XD; YA; YB; YC; YD; YE; YF; ZA; ZB
   '-------------- The Interior Data Set --------------
   DATA 1, 2
10 DATA 3, 4, 5
   DATA 6, 7, 8, 9
   END
```

The output from the above program appears below.

```
1  2  3  4  1  2  3  4  5  6  3  4
```

WIDTH Statement

Purpose Sets the page width for output that is sent to the terminal.

Format WIDTH pgwidth

pgwidth	Numeric expression whose value, when rounded to the nearest integer, specifies the page width to be used for output to the terminal.

Remarks

- The page width is the maximum number of characters that can be printed before a carriage return is generated.

  If the page width:

  - Exceeds the NOS/VE maximum page width, the maximum is used.
  - Is less than 14 (the length of a print zone), a runtime error results.

  If the length of a value to be printed:

  - Exceeds the space available on the current line, but is less than the page width, the value is printed at the beginning of the next line.
  - Exceeds the page width, as much of it as can fit on the current line is printed, and the value is continued on as many subsequent lines as needed.

- The page width specified is used until the program ends, or until the page width is changed by another WIDTH statement.

Examples This statement sets the page width for output to the terminal at 65 characters.

```
WIDTH 65
```

PRINT Statement

This section discusses the NOS/VE BASIC statements and format functions used for sending output to the terminal using the PRINT statement.

PRINT Statement Format

Purpose	Print output at the terminal.
Format	PRINT printlist

printlist — Optional list of expressions and format function references that are separated by commas or semicolons. One or more spaces between items is equivalent to a semicolon specification. The last item can be followed by a comma or semicolon.

Remarks

- A carriage return is issued:
  - If the print list is omitted.
  - On completion of any PRINT statement whose print list does not end with a comma, a semicolon, or a format function reference.
- The expressions in the print list are evaluated, and their values are printed in sequence. The spacing of the output is controlled by the punctuation that follows each print list item and by the format functions included in the print list.
- A numeric value printed by the PRINT statement:
  - Is preceded by a space if the value is nonnegative. No space precedes the minus sign of a negative value.
  - Is followed by a space (unless the value ends in the last print position of a line, in which case the trailing space is omitted).

Remarks (cont)

- The value of a real expression printed by the PRINT statement is displayed:
  - Without trailing zeros.
  - With no zero digit to the left of the decimal point if its magnitude is less than one.
  - Without a decimal point (integer format) if its fractional part is zero.
  - In decimal (fixed point) format if it can be represented as accurately in decimal format, using seven or fewer digits, as it can in exponential format. Otherwise, the exponential (floating point) format, using one digit to the left of the decimal point, is displayed.
- If the length of a value to be printed:
  - Exceeds the space available on the current line, but is less than the page width, the value is printed at the beginning of the next line.
  - Exceeds the page width, as much of it as can fit on the current line is printed, and the value is continued on as many subsequent lines as needed.

Examples

The following table shows several PRINT statements and the resulting output:

PRINT Statement	Output
PRINT 10.5^3	1157.625
PRINT -7.5^4	-3.1640625E+3
PRINT 10^30	1.E+30
PRINT 1.E30	1.E+30
PRINT +123.E20	1.23E+22
PRINT -.3E22	-3.E+21
PRINT .777E+18	7.77E+17
PRINT +.04E+26	4.E+24
PRINT +10.5210E+3	10521
PRINT -7.6E1	-76

Print Zones and Comma Format

The leftmost print position of a NOS/VE BASIC print line is designated position one. The print line is divided into 14-character print zones.

A comma in the print list of a PRINT statement moves the print cursor to the beginning of the next zone. If there are no more print zones in the current line, the cursor moves to the beginning of the next line. This is the beginning of the next print zone.

A comma at the end of the print list of a PRINT statement works exactly like a comma elsewhere in the print list. Printing continues at the current cursor position when a subsequent PRINT, PRINT USING, or WRITE statement is executed.

Examples

The following is an example of a PRINT statement.

```
PRINT 4.25,-48,"GOOD ANSWER"
```

The value 4.25 is printed in zone one, print positions 2-5 (a leading space is provided). The value -48 is printed in zone two, print positions 15-17 (no leading space). The value "GOOD ANSWER" is printed in zone three, print positions 29-39. A carriage return is issued because the print list does not end with a comma, semicolon, or format function. The output is as follows:

```
Zones:     1             2             3

Output:     4.25         -48           GOOD ANSWER
```

Examples

In the following sample program, each line is labeled for reference. The result of the execution of each line is explained beneath the program.

```
10 WIDTH 56
20 PRINT 1,2,3,4,
30 PRINT "LONGER THAN ONE ZONE",5
40 PRINT 6,
50 PRINT
60 PRINT 7 : END
```

10: Terminal page width set at 56 characters (four print zones).

20: Values 1, 2, 3, 4 printed in positions 2, 16, 30, 44, respectively. Ending comma causes cursor to move to first print zone on the next line.

30: Value "LONGER THAN ONE ZONE" printed in positions 1-20. Value 5 printed in position 30. Carriage return issued (no ending punctuation). Cursor moves to start of print line three.

40: Value 6 printed in position 2. Cursor moves to position 15.

50: Carriage return issued. Cursor moves to start of print line four.

60: Value 7 printed in position 2. Carriage return issued. Cursor moves to start of print line five. Program ends.

The output is as follows:

```
Zones:     1             2             3             4


Output:     1             2             3             4
           LONGER THAN ONE ZONE 5
            6
            7
```

Semicolon Format

A semicolon in the print list of a PRINT statement holds the print cursor at its current position. The next value printed immediately follows the last one printed.

This format causes printed string values to run together if no spacing is provided. However, one trailing space is provided after numeric values. In addition, one leading space is provided for positive values, but not for negative values.

If a printed value ends in the final print position of a line, a carriage return is not issued until a subsequent value is printed.

A semicolon at the end of the print list of a PRINT statement works exactly like a semicolon elsewhere in the print list. Printing continues at the current cursor position when a subsequent PRINT, PRINT USING, or WRITE statement is executed.

Examples

The following PRINT statement holds the print cursor at its current position, except one trailing space is provided after numeric values

```
PRINT "HOME";"WORK";-1;"OR";2;"OR";3;"HOURS"
```

This statement produces the output below. Note that no space separates consecutive strings, and no space separates a string from a subsequent negative value.

```
HOMEWORK-1 OR 2 OR 3 HOURS
```

One or more spaces (with no other punctuation) between items in the print list of a PRINT statement function exactly as if a semicolon were provided. However, spaces at the end of a print list have no meaning.

Examples

The lines labeled 10 and 20 (combined) print the value "GO TOGETHER", holding the cursor on the first print line.

The line labeled 30 causes a carriage return. The cursor moves to the start of print line two.

The line labeled 40 prints the value "PLEASE", and issues another carriage return. The cursor is positioned at the start of print line three when the program terminates.

```
10 PRINT "GO ";"TO";
20 PRINT "GET"; "HER";
30 PRINT
40 PRINT "PLEASE" : END
```

The output is as follows:

```
GO TOGETHER
PLEASE
```

SPC Format Function

Purpose Inserts spaces into a line of output.

Format SPC(spaces)

spaces	Numeric expression whose value, when rounded to the nearest integer, specifies the number of spaces to be printed.

Remarks

- If the number of spaces specified:
  - Exceeds the available space in the current line, the cursor moves to the beginning of the next line. No spaces on this new line are provided.
  - Is zero, the cursor does not move.
  - Is negative, a runtime error results.
- If a print list ends with an SPC function reference, no carriage return is issued unless the reference itself causes one.
- A semicolon or comma following a SPC function works exactly like a semicolon or comma in a print list.

Examples This statement prints the value "NAME   ADDRESS". The three spaces are provided by the SPC function reference.

```
PRINT "NAME";SPC(3);"ADDRESS"
```

TAB Format Function

Purpose Move the print cursor to a specified print position.

Format TAB(column)

column Numeric expression whose value, when rounded to the nearest integer, specifies the print position to which the print cursor is to be moved.

Remarks

- If the specified print position p:
  - Exceeds the page width w, the integer n = (p MOD w) is computed. The print cursor moves to print position n of the next line.
  - Is less than the current position of the print cursor, the cursor moves to print position p of the next line. The cursor never moves backwards as a result of a TAB function reference.
  - Is less than 1, a value of 1 is used. A warning is issued, but no runtime error results.
- If a print list ends with a TAB function reference, no carriage return is issued unless the reference itself causes one.
- A semicolon or comma following a TAB function works exactly like a ; or , in a print list.

Examples

- This statement prints the value "DIVISION" in print positions 1-8, the value "DEPARTMENT" in print positions 20-29, and the value "UNIT" in print positions 40-43. The intervening positions are filled with blanks.

```
PRINT "DIVISION";TAB(20);"DEPARTMENT";TAB(40);"UNIT"
```

- If N receives the value 10 through the INPUT statement, this program fragment prints the following output:

```
DEFINT I,N
INPUT N
FOR I = 1 TO N
  PRINT TAB(1 + 4*(I - 1) );"&";
NEXT I
```

These ampersands appear in columns 1, 5, 9, 13, 17, 21, 25, 29, 33, and 37. The print cursor remains at print position 38.

```
&   &   &   &   &   &   &   &   &   &
```

PRINT USING Statement

This section discusses the format options for sending program output to the terminal using the PRINT USING statement.

PRINT USING Statement Format

Purpose	Allows you to specify in detail how output should be displayed.
Format	PRINT USING formstr ; printlist

formstr	Required string expression whose value (the format string) specifies the format of the output.
;	Required delimiter separating the format string from the print list.
printlist	Nonempty list of expressions that are separated by commas or semicolons. One or more spaces between items is equivalent to a semicolon specification. The last item can be optionally followed by a comma or semicolon.

Remarks

- The print list must contain at least one item. The expressions in this list are evaluated, and their values are printed in sequence, using the format specified in the format string.

- If the length of a value to be printed:

  - Exceeds the space available on the current line, but is less than the page width, the value is printed at the beginning of the next line.

  - Exceeds the page width, as much of it as can fit on the current line is printed, and the value is continued on as many subsequent lines as needed.

- Commas and semicolons are interchangeable when used to separate items in the print list of a PRINT USING statement. In this context, they act only as delimiters, unlike their use in the PRINT statement. However, these marks have distinct interpretations when placed at the end of the print list.

- A PRINT USING statement never supplies spaces unless they are specifically designated in the format string. For example, the trailing space that a PRINT statement automatically provides after the printing of a numeric value is not provided by the PRINT USING statement.

- A carriage return is issued on completion of any PRINT USING statement whose print list does not end with either a comma or a semicolon.

- A comma or semicolon at the end of the print list of a PRINT USING statement controls the movement of the print cursor in exactly the same way as with the PRINT statement. Thus:

  - An ending comma causes the cursor to move to the beginning of the next print zone.

  - An ending semicolon holds the cursor in its current position.

  Printing continues at the current cursor position when a subsequent PRINT, PRINT USING, or WRITE statement is executed.

The format string is analyzed as a sequence of format components possibly separated by literal components. A format string must have at least one format component or a runtime error results.

A format component is a string of format characters that specify how the next print list item is to be displayed.

A literal component is a string which is not used to specify format. When a literal component is reached, its value is printed, exactly as it appears.

A pointer keeps track of the current format component within a format string. After a value is printed using the current format component, the pointer moves forward to the next format component. This causes any intermediate literal component to be printed.

If the end of a format string is reached, but not all print list items have been printed, the pointer wraps around to the beginning of the format string, and printing continues.

If a format component is ill-formed, or is inappropriate for the data type of a corresponding print list item, a runtime error results.

There are two sets of format characters, one for string values and one for numeric values.

String Format Characters

This section discusses three format characters (! & \) used in the format string of a PRINT USING statement to specify how a string is to be printed.

Each format character controls the length of the field (the section of the print line) reserved for the output of a string.

Remember that punctuation between items in a print list only delimits consecutive items. However, the punctuation at the end of the print list controls the subsequent movement of the print cursor.

An exclamation point (!) specifies that only the first character of a string is to be printed. If the string is the null string, a space is printed.

Examples

In the following program fragment note that the wrap-around feature is used. A carriage return is issued on completion of the output because the print list does not end with a comma or a semicolon.

```
DEFSTR A,B
LET A1 = "RESEARCH" : LET A2 = "DEVELOPMENT"
LET B1 = "PUBLICATIONS" : LET B2 = "GRAPHICS"
PRINT USING "! AND ! : "; A1,A2;B1,B2
```

The output produced by this program fragment appears below:

```
R AND D : P AND G :
```

An ampersand (&) specifies that a string is to be printed in a field equal in length to that of the string.

Examples

In the following program fragment note that the wrap-around feature is used. The print cursor remains at its current position on completion of the output because the print list ends with a semicolon.

```
DEFSTR C,D,X
LET C1 = "ALL" : C2 = "NONE"
LET D1 = "COMPILE" : LET D2 = "EXECUTE"
LET X1 = "OR" : LET X2 = "AND"
PRINT USING "OPTIONS:  ! & ! / "; C1,X1,C2;D1,X2,D2;
```

The output produced by this program fragment appears below:

```
OPTIONS:  A OR N / OPTIONS:  C AND E /
```

A pair of reverse slants (\) with m spaces between them specifies that a string is to be printed in a field of length (m + 2), where m is a nonnegative integer.

The size of m is limited only by the requirement that the entire PRINT USING statement be contained in one line.

If the string value is too long to fit in the field, the first (m + 2) characters of the string are printed. Otherwise, the string is left-justified, and trailing spaces fill the field.

Examples

In the following example the string value is too long to fit in the specified field.

```
DEFSTR C,F,L
LET LN = "LASTNAME" : LET FN = "FIRSTNAME"
LET CN = "CITY"
PRINT USING "CODE NAME:  \    \!"; LN,FN
PRINT USING "RESIDENCE:  \    \@"; CN
```

The output produced by this program fragment appears below. The value of LN is truncated to fit the specified 6-position field. The value of CN is left-justified in a 6-position field.

```
CODE NAME:  LASTNAF
RESIDENCE:  CITY  @
```

Standard Numeric Format Characters

This section discusses four standard format characters (# . + -) used in the format string of a PRINT USING statement to specify how a number is to be printed.

Remember that punctuation between items in a print list serves only to delimit consecutive items. However, the punctuation at the end of the print list controls the subsequent movement of the print cursor.

A number sign (#) in a format string reserves one position in a field. This position can be filled with a digit, comma, or arithmetic sign.

A period (.) in a format string reserves one position in a field for a decimal point and specifies where the decimal point is to appear.

If the field specified for printing an integer reserves:

- More positions than are needed, the integer is right-justified, and leading spaces are used to fill the field.

- Fewer positions than are needed, the field is lengthened to accommodate the value. In addition, a percent sign (%) is displayed as the first character in the field to flag the format overflow.

If the field specified for printing a number in decimal format reserves:

- Fewer positions to the right of the decimal point than are needed, the number is rounded to fit within the field.

- Fewer positions to the left of the decimal point than are needed, the field is lengthened to accommodate the value. In addition, a percent sign (%) is displayed as the first character in the field to flag the format overflow.

- More positions to the right of the decimal point than are needed, trailing zeros are used to fill the field.

- More positions to the left of the decimal point than are needed, leading spaces are used to fill the field. However, at least one digit is displayed to the left of the decimal point unless the period is the leftmost character in the format component.

Examples

The format overflow in the third line occurs because only two positions are reserved to the left of the decimal point for a value that has three such digits.

```
PRINT USING "ANSWER ##:  ##.##"; 3,84.568
PRINT USING "ANSWER ##:  ##.##"; 7,.951
PRINT USING "ANSWER ##:  ##.##"; 10,372.2
```

The output produced by this program fragment appears below.

```
ANSWER  3:  84.57
ANSWER  7:   0.95
ANSWER 10:  %372.20
```

A plus symbol (+) or minus symbol (-) in a numeric format component reserves one position in a field. This symbol specifies how the sign of a nonzero value is to be displayed. No sign is displayed for the value zero.

A numeric format component whose:

- Leftmost character is the plus symbol specifies that the sign of a number, plus or minus, is to be displayed to the left of the number.

- Rightmost character is the plus symbol specifies that the sign of a number, plus or minus, is to be displayed to the right of the number.

- Rightmost character is the minus symbol specifies that the minus sign of a negative number is to be displayed to the right of the number. The plus sign of a positive number is not displayed under this format.

If no plus or minus symbol appears in a numeric format component, the sign of a negative number is printed to the left of the number. The sign of a positive number is not displayed under this default format.

Note that the default format does not automatically reserve a space in a field for the minus sign of a negative number. Thus, the space required to print the minus sign can cause overflow even if the absolute value of that number fits in the specified field.

NOTE

The plus symbol causes an arithmetic sign to be displayed regardless of what that sign is. In contrast, the minus symbol causes only minus signs to be displayed. However, both formats reserve one position in a field for the sign. The default format causes only minus signs to be displayed, but does not reserve a sign position in a field.

Examples

In the third line, format overflow occurs under the default format because only one position is reserved for a negative value that needs two positions, one for the digit and one for the sign.

```
PRINT USING "#+ | +# | #- | ##" ;  5,-9,-5,9
PRINT USING "#+ | +# | #- | ##" ;  -5,9,5,-9
PRINT USING "#- | #- | # | #"  ;  5,-9,5,-9
```

The output produced by this program fragment appears below.

```
5+ | -9 | 5- |  9
5- | +9 |  5 | -9
 5 | 9- |  5 | %-9
```

Special Numeric Format Characters

This section discusses four special format characters (^ , * $) that are used in the format string of a PRINT USING statement to print numbers in special ways.

Remember that punctuation between items in a print list serves only to delimit consecutive items. However, the punctuation at the end of the print list controls the subsequent movement of the print cursor.

Recall the exponential format:

```
    r*10^s = rEs.
```

A circumflex (^) placed after the digit position characters in a numeric format component reserves one position in a field for the exponent used in exponential format. A minimum of three circumflexes is required so that the form E+n or E-n, where n is a single digit, can be displayed.

If the part of the field specified for printing the exponent reserves:

- Fewer positions than are needed, the field is lengthened to accommodate the value. In addition, a percent sign (%) is displayed as the first character in the field to flag the format overflow.

- More positions than are needed, the exponent is right-justified, and leading zeros are used to fill this section of the field.

Examples

Consider the format in which commas are used to group those digits of a number that lie to the left of the decimal point.

```
PRINT USING "#.#^^^"; 36000
PRINT USING "#.###^^^^+"; 0.00235
PRINT USING "+##.###^^^^"; -0.00235
PRINT USING ".###^^^^"; 0.00235
```

The output produced by this program fragment appears below.

```
3.6E+4
2.350E-03+
-23.500E-04
.235E-02
```

Examples

The following is an example of Digit Grouping Format:

```
2,576,421.93
```

A comma in a numeric format component reserves a position in a field, and specifies that a number is to be printed using the digit grouping format.

A comma can appear anywhere in the component except as the first or last character. If a comma is the first or last character in a numeric format component, it is treated as part of a literal component instead of as a format character.

A comma used in conjunction with an exponential format reserves an extra field position, but does not affect the display.

Examples The print cursor moves to the next print zone on completion of the output because the print list ends with a comma.

```
PRINT USING "#,###,###.#"; 1234567.89,
```

The output produced by this statement appears below.

```
1,234,567.9
```

A pair of asterisks (*) at the beginning of a numeric format component reserves two positions in a numeric field, and specifies that any leading spaces in the field are to be filled with asterisks.

The asterisk format can be used in conjunction with the exponential format.

Examples In the following example note the format overflow for the second printed value. The print cursor remains at its current position on completion of the output because the print list ends with a semicolon.

```
PRINT USING "**##. "; 1.74,-1532.1,123.57;
```

The output produced by this statement appears below.

```
***1.7   %-1532.1   *123.6
```

A pair of dollar signs ($) at the beginning a numeric format component reserves two positions in a numeric field, and specifies that a dollar sign is to precede the leftmost digit of a printed value.

The dollar format cannot be used in conjunction with the exponential format. Also, only the trailing minus symbol or default sign format can be used with the dollar format.

A pair of asterisks followed by a dollar sign at the beginning of a numeric format component combines the asterisk and dollar formats. This hybrid format reserves three positions in a numeric field. It specifies that a dollar sign is to precede the leftmost digit of a printed value and leading spaces are to be filled with asterisks.

Examples

The format overflow in the third line occurs because only four positions are reserved to the left of the decimal point for a value that has five such digits.

```
PRINT USING "$$###.##-  "; 7642.259
PRINT USING "$$###.##-  "; -432.81
PRINT USING "$$###,.##"; 54321.0
PRINT USING "**$###.##"; 1.25
```

The output produced by this program fragment appears below.

```
$7642.26
   $432.81-
%$54,321.00
****$1.25
```

Format Characters as Literals

An underscore (_) in a format string causes the character that follows to be treated as part of a literal component. This literal character is printed exactly as it appears. The underscore is not printed.

This notation enables you to override the special significance of a format character. A preceding underscore suppresses the format function of this character.

To print an underscore, include two consecutive underscores. The first one removes the significance of the second one, allowing the second one to be printed.

Any character can follow an underscore. However, this format character is useful only when it is followed by one of the characters listed below.

```
\   !   &   #   .   +   -   ^   ,   *   $   _
```

Examples

In the PRINT USING statement, the underscore format character is used so that the second ampersand and the first number sign are treated as literal characters rather than as format characters.

```
DEFINT I,N : DEFSTR A,B,F
DATA 3,WILLIAM,MARY,5,LEWIS,CLARK,8,FRANKLIN,ELEANOR,2
READ N
FOR I = 1 TO N
  READ A,B,F
  PRINT USING "& _& & ON DETAIL _##"; A,B,F
NEXT I
```

The output from this program fragment appears below.

```
WILLIAM & MARY ON DETAIL #5
LEWIS & CLARK ON DETAIL #8
FRANKLIN & ELEANOR ON DETAIL #2
```

Scanning Format Strings

Remember that the format string of a PRINT USING statement is analyzed as a sequence of format components possibly separated by literal components.

Format components are found by scanning the format string from left to right. The beginning of a format component is identified by a format character. The format component that begins with this character is defined as the longest character sequence that can be interpreted as a format component, taking into account the characters encountered along the way.

This means that the appearance of some characters might prohibit other characters from consideration later on as members of the same format component. For example, once a circumflex (^) is encountered, the subsequent appearance of a number sign (#) cannot be considered part of the same format component. Instead, the number sign is interpreted as a member of a different format component.

Examples

The second period cannot be part of the first numeric format component because only one period is allowed in such a component. Since the second period is followed by a number sign, this period must belong to a second numeric format component.

```
PRINT USING "#,###.#,.#"; 5432.1,0.6
```

Although a comma is a numeric format character, the second comma cannot be part of either numeric format component because a comma cannot begin or end such a component. Hence, the second comma is interpreted as a literal character and is printed as is.

Therefore, the format string consists of three components. The first seven characters define a numeric format component, the subsequent comma defines a literal component, and the last two characters define a second numeric format component.

The output produced by this statement appears below.

```
5,432.1,.6
```

WRITE Statement

Purpose	Writes values at the terminal in a form that resembles a list of BASIC constants, complete with separating commas.
Format	WRITE writelist

writelist — Optional list of expressions that are separated by either commas or semicolons.

Remarks

- Commas and semicolons are interchangeable when used to separate items in the write list.

- If the write list is omitted, a carriage return is issued. Otherwise, the expressions in the write list are evaluated, and their values are printed in sequence in the form described below.

- When the WRITE statement writes data:
  - Commas are written between values making the output look like a delimited list.
  - Quotation marks are provided to delimit string constants.
  - Each quotation mark embedded in a string is written as a pair of successive quotation marks.
  - No spaces are provided between printed values. For example, the trailing space that a PRINT statement automatically provides after the printing of a numeric value is not provided by the WRITE statement.
  - A carriage return is always issued after the output is produced.

- If the length of a value to be printed:
  - Exceeds the space available on the current line, but is less than the page width, then the value is printed at the beginning of the next line.
  - Exceeds the page width, as much of it as will fit on the current line is printed, and the value is continued on as many subsequent lines as are needed.

Examples

The following program fragment shows an example using the WRITE statement.

```
READ A%,B%,C$,D$
DATA 472,-561,MR. "T","KIDS"
WRITE A%,B%,C$,D$
```

The output from this program fragment appears below.

```
472,-561,"MR. ""T""","KIDS"
```

The result is a delimited list of BASIC constants.

BEEP Statement

Purpose	Sends BEL, the ASCII bell character, to the terminal.
Format	BEEP
Remarks	The BEEP is equivalent to the following statement.

```
PRINT CHR$(7);
```

Some terminals ignore the BEL character.

Arrays 11

Arrays 11

An array is a data structure which allows logically related values with the same data type to be stored under a single name.

This chapter discusses NOS/VE BASIC arrays. It includes descriptions of related library functions and examples of array input and output.

Array Overview

An array is a collection of memory locations that are identified by a single name. This name is called the array name.

The memory locations in the array act as storage boxes for a group of related values with the same data type. Each memory location can store a single value from the group of related values.

A particular memory location in an array is referenced using the array name and a sequence of numbers called subscripts. The subscripts identify the memory location by its position within the array. This named memory location is called an array element or a subscripted variable.

An array element is similar to a scalar variable, but uses a more complex reference format. An external routine can contain an array with the same name as a scalar variable because they have different reference formats.

The data type associated with an array name establishes the data type of every element of the array. Limits on the values of array elements are the same as those on scalar variables of like data types.

Each array has one or more dimensions.

The dimensioning of arrays in a program can be either static or dynamic, determined at compile time by the BASIC command. Under static dimensioning, the size and shape of an array are fixed at compile time. This can provide more efficient programs for applications that do not require the flexibility of dynamic dimensioning. Arrays dynamically dimensioned at compile time can have their dimensions changed during program execution.

A dimension is a set of consecutive integers used to index the memory locations within an array. A dimension is defined by specifying its lower and upper bounds. Each integer in a dimension is called a subscript and can be negative.

The size of a dimension is computed from the lower and upper bounds by the formula:

    Size = Upper Bound - Lower Bound + 1.

The size and bounds of a static array dimension are fixed at compile-time.

The size and bounds of a dynamic array dimension can change during program execution.

The number of dimensions of an array determines the number of subscripts that are necessary to identify an array element.

If n dimensions are used, the array is called an n-dimensional array. In particular:

- A one-dimensional array is also called a list or vector.

- A two-dimensional array is also called a table or matrix.

The number of dimensions of an array is fixed at compile-time and is limited only by the NOS/VE BASIC maximum line length.

The value of an array element is accessed by specifying the array name followed by a list of subscripts, one from each dimension. The dimension corresponding to the nth listed subscript is called the nth dimension.

Array Element References

Purpose	References an element of an N-dimensional array.
Format	arrname(sub1 , sub2 , ... , subN)
	arrname — Name of the array containing the element.
	subJ — Numeric expression whose value specifies the Jth subscript, where (1 <= J <= N).
Remarks	The value of SUBJ is rounded to the nearest integer k. This specifies that the element being referenced has a subscript of k in the Jth dimension. If k is less than the lower bound or greater than the upper bound for the dimension, a runtime error results.

Examples

- In the following list, the value of each element of a one-dimensional array named DAY appears beneath the reference that accesses the value. Array DAY has only one dimension. This dimension has a lower bound of -1 and an upper bound of 3. This means the array has a size of 5.

DAY(-1)	DAY(0)	DAY(1)	DAY(2)	DAY(3)
15.4	-3.1	2.6	18.9	-34.2

The array element DAY(2) has the value 18.9. The subscripted variable DAY(0) has the value -3.1.

You might refer to the value 2.6 as the value of the third array element, and yet this element has a subscript of 1. This illustrates a conceptual difficulty that arises when a dimension does not have a lower bound of one.

- Consider a two-dimensional array named GAME% whose values appear in the following matrix.

$$\begin{array}{cc|cccc|} & & \multicolumn{4}{c}{\text{column}} \\ & & 1 & 2 & 3 & 4 \\ \cline{3-6} \text{Row} & 1 & 0 & -2 & 4 & 6 \\ & 2 & -1 & 3 & -5 & 7 \\ & 3 & 6 & -4 & 2 & 0 \\ & 4 & -7 & 5 & -3 & 1 \\ \cline{3-6} \end{array}$$

Assume that the first dimension identifies a row and the second dimension identifies a column. Both dimensions have a lower bound of one and an upper bound of four.

Array Element	Value
GAME%(2,3)	-5
GAME%(3,2)	-4
GAME%(1,1)	0
GAME%(3,4)	0

Dimension Bound Specification

This section discusses the ways in which dimension bounds for arrays are established and changed. It also describes the library functions which pertain to arrays.

OPTION BASE Statement

Purpose	Sets the default lower bound (or base) for all arrays in an external routine to either 0 or 1. These two choices are provided because they are the ones that are most frequently used in applications.
Format	OPTION BASE choice
	choice Option base, specifying the default lower bound for all arrays in an external routine. Possible values: 0 or 1.

Remarks

- The OPTION BASE statement in an external routine:
  - Defines the default lower bound for every implicitly dimensioned array in the routine.
  - Defines the default lower bound to be used for every dimension statement in the routine.
  - Must precede all the dimension statements in the routine.
- An external routine can have at most one OPTION BASE statement. If omitted, the option base is set to 0.

Default Specification

Default array bound specification is provided so that arrays of limited size can be dimensioned implicitly.

By default:

- The lower bound of each dimension of an array is the option base (0 or 1).
- The upper bound of the first and second dimensions is 10. The upper bound of all other dimensions is the option base.

Hence, if b is the option base, an n-dimensional array is dimensioned

```
    (b:10 , b:10 , b:b , ... , b:b)
```

by default. Each n-dimensional array has (11-b)^MIN(n,2) elements.

Arrays that are shared through the COMMON statement have default dimension bounds as described previously. For example, if the array is only dimensioned in the calling routine, then the dimensions are the same for the array in the called routine. However, if the array is dimensioned differently in the calling and the called routine, then its actual size can differ.

Examples

If this statement is the first statement of a program, it declares by default that A is a one-dimensional real array whose 11 elements are subscripted 0 through 10. The array element A(5) takes the value 8.0. The remaining elements take the value 0.0.

```
LET A(5) = 8.0
```

DIM Statement

Purpose	Explicitly establishes or changes the dimensions of each array listed. This is done by specifying lower and upper bounds for each dimension of each array.

Format

```
DIM  adlist
```

adlist	List of array declarations that are separated by commas.

Remarks

- Note that the number of dimensions of an array is specified at compile-time by the first array reference and cannot be changed at runtime.

- At runtime in a statically dimensioned routine, a DIM statement in the runtime execution path is passed over with no effect.

- An array declaration for an N-dimensional array has the format:

  ```
  arrname(lower1:upper1 , lower2:upper2 , ... ,
  lowerN:upperN)
  ```

arrname	Identifier, naming the array. The data type associated with this identifier establishes the data type of all elements in the array.
lowerJ	Optional numeric expression whose value, when rounded to the nearest integer, specifies the lower bound of the Jth dimension, where (1 <= J <= N). If omitted, the subsequent colon is also omitted.
upperJ	Numeric expression whose value, when rounded to the nearest integer, specifies the upper bound of the Jth dimension, where (1 <= J <= N).

Remarks (cont)

- The DIM statement sets the dimension bounds for listed arrays. If an array was previously dimensioned, the DIM statement in a dynamically dimensioned routine redimensions the array using new dimension bounds. If the size of a dimension is decreased, or the subscript range of a dimension is shifted, only those array elements whose subscripts are preserved in the new indexing set remain accessible. New elements that were not previously defined are set to zero or the null string, as dictated by data type.

- In a statically dimensioned routine, the DIM statement is a compile-time declaration rather than an executable statement; a static array can be declared only once in an external routine.

- If the lower bound for a dimension is not specified:

  - The lower bound is the option base (0 or 1) in effect for the external routine containing the DIM statement.

  - The single expression appearing for that dimension in the array declaration specifies the upper bound.

- The magnitude of a dimension bound (lower or upper) cannot exceed (2^31 - 1).

- If a dynamically dimensioned formal array is redimensioned by a called routine, the dimension bounds of the corresponding actual array are simultaneously changed in the calling routine. If a dynamically dimensioned called routine attempts to redimension a statically dimensioned formal array, a runtime error results.

- Any array not explicitly dimensioned in a DIM statement is implicitly dimensioned using the default array bounds. That is, the lower bound of each dimension is the option base. The upper bound is either 10 or the option base, depending on the number of the dimension.

- The maximum number of elements for an array depends on the maximum contiguous storage allowed for your installation and account. See your site administrator for specific information.

Array Library Functions

Purpose

Returns the lower and upper bounds, respectively, of a specified dimension of an array.

Format

```
LBOUND(arrname , dimnum)
UBOUND(arrname , dimnum)
```

arrname	Name of the array being analyzed.
dimnum	Optional numeric expression, specifying the dimension whose lower or upper bound, respectively, is to be returned. If omitted, the preceding comma is also omitted, and the value 1 is assumed.

Remarks

The value of DIMNUM is rounded to the nearest integer. If this integer is less than one or greater than the number of dimensions of ARRNAME, a runtime error results.

Examples

- These statements declare a one-dimensional integer array named MONTH%. The single dimension has a size of 12, with a default lower bound of 1, and an upper bound of 12.

```
OPTION BASE 1
DIM MONTH%(12)
```

(The following examples assume dynamically dimensioned arrays.)

- These statements declare a two-dimensional real array named MARK. The first dimension has a size of 8, with a default lower bound of 0, and an upper bound of 7. The second dimension has a size of 6, with a lower bound of 4 and an upper bound of 9. The LBOUND function reference in the third line returns the lower bound of the second dimension of MARK. Hence, N% receives the value 4.

```
OPTION BASE 0
DIM MARK(7,4:9)
LET N% = LBOUND(MARK,2)
```

- The line labeled 10 declares a one-dimensional string array named A$. The single dimension has a size of 7, with a lower bound of 0, the default option base, and an upper bound of 6. The line labeled 80 shrinks the size to 5, and shifts the subscript range. This yields new lower and upper bounds of 4 and 8, respectively.

```
10 DIM A$(6)
       .
       .
80 DIM A$(4:8)
90 LET M% = UBOUND(A$)
```

As a result, the values of A$(0), A$(1), A$(2), and A$(3) are lost; the values of A$(4), A$(5), and A$(6) are preserved. The new elements A$(7) and A$(8) are set to the null string. The UBOUND function reference in the line labeled 90 returns the upper bound of the first dimension of A$. Hence, M% receives the value 8.

Array Input/Output

The values of array elements can be assigned and printed using FOR-NEXT loops.

Examples

- This program fragment stores the NxN multiplication table in the array A. The size of the table is specified using the INPUT statement (line labeled 10). The DIM statement (line labeled 20) establishes the required dimension bounds.

```
    OPTION BASE 1
    DEFINT A,I,J,N
10  INPUT N
20  DIM A(N,N)
    FOR I = 1 TO N
       FOR J = 1 TO N
          LET A(I,J) = I*J
       NEXT J
    NEXT I
```

Examples (cont)

- This program fragment stores a sequence of integral powers of 2 in array B. These values are then printed as a list of numbers that are separated by commas. The lower and upper bounds on the exponent are specified with the INPUT statement (line labeled 30).

```
    DEFINT I,M,N
30  INPUT M,N
    DIM B(M:N)
    FOR I = M TO N
        LET B(I) = 2^I
        PRINT B(I);",";
    NEXT I
```

  If M and N are assigned the values -2 and 6, respectively, the following output is generated.

```
.25 , .5 , 1 , 2 , 4 , 8 , 16 , 32 , 64 ,
```

- This program fragment illustrates the use of zone printing for a two-dimensional string array. Effective table display using zone printing requires a limited number of columns (5 or less for a 79-character line). If zone printing is used with two-dimensional numeric arrays, columns will not be neatly aligned unless all elements have the same number of digits.

```
OPTION BASE 1
DEFINT I,J,M,N
READ M,N
DIM S$(M,N) ' Small N required for zone printing.
FOR I = 1 TO M
    FOR J = 1 TO N
        READ S$(I,J)
        PRINT S$(I,J), ' Zone printing for columns.
    NEXT J
    PRINT ' Carriage return when row completed.
NEXT I
```

Examples (cont)

This program fragment uses the PRINT USING statement to produce a table of quarterly profits. The first dimension of QTR.PROFIT represents the year. The second dimension represents the quarter of the year. The figures for each year are printed on a separate line using zone printing. The print field for each quarterly figure allows for profits or losses of up to $99,999.99. Commas are used to group digits, and the sign trails each figure. A blank line separates the output for consecutive years.

```
FOR I = 1 TO M
    FOR J = 1 TO 4 ' One column for each quarter of year.
        PRINT USING "$$#,###.##+    "; QTR.PROFIT(I,J);
    NEXT J
    PRINT ' Carriage return when year (row) completed.
    PRINT ' Blank line between years (rows).
NEXT I
```

ERASE Statement

Purpose Frees storage occupied by dynamically dimensioned arrays.

Format ERASE anlist

anlist List of array names that are separated by commas. This list specifies the arrays that are to be erased.

Remarks

- When an ERASE statement is executed, both the lower and upper bounds of each dimension of a listed array are set to the option base (0 or 1). The value of the single remaining element is set to either zero or the null string, as dictated by data type.

- If a listed array is:

  - Shared with other external routines through the COMMON statement, the array is erased in every external routine.

  - A formal array, the corresponding actual array is also erased in the calling routine.

- The second item above points out a major difference between the ERASE and CLEAR statements. For more information about the CLEAR statement, see chapter 4.

- A compile-time-error results if an ERASE statement is used in a routine compiled with statically dimensioned arrays.

String Processing 12

String Processing 12

This chapter describes the NOS/VE BASIC statements, operations, and library functions used in string processing. The string library functions have been categorized according to usage.

Many of the items discussed in this chapter refer to character positions within a string. The leftmost character position is labeled position one.

String Expression Review

A string expression is one or more quoted string constants, string variables, and string function references that are separated by plus signs. Subexpressions occurring within a string expression can be enclosed by parentheses.

In this context, the plus sign is called the concatenation operator. Concatenation, the only string operation, joins two string operands. The length of the string produced is the sum of the lengths of the operands.

Examples

The following string expression (A$ + B$) is read "A is concatenated with B".

If A$ = "LOOK" and B$ = "OUT", then:

(A$ + B$)	Has the value "LOOKOUT".
(B$ + A$)	Has the value "OUTLOOK".

Colon-Substring Notation

Purpose	Allows you to address a substring by its location within a host string variable. A colon-substring reference can be used both to assign values to substrings and to access values for other processing.
Format	svar(pos1 : pos2)

svar	String identifier denoting the host variable that contains the substring being referenced. SVAR cannot be a substring expressed by colon substring notation or a MID$ reference.
pos1, pos2	Numeric expressions whose values specify the positions within the host string of the first and last characters, respectively, of the substring being referenced.

Remarks

- To help explain the evaluation rules, suppose S$ is a string variable of length n. The reference S$(POS1:POS2) denotes the substring of S$ made up of the characters in positions POS1 through POS2, where POS1 and POS2 are defined as follows:

  If the values of either POS1 or POS2 are:

  - Less than one, they are increased to one.

  - Greater than n, they are decreased to n.

  - Nonintegers, they are rounded to the nearest integers.

    Hence,

    ```
    POS1  =  CINT( MIN( MAX(1,POS1) , n) ),
    and
    POS2  =  CINT( MAX( MIN(POS2,n) , 1) ).
    ```

  In addition, if POS1 is greater than POS2 , then the substring referenced is a null substring that precedes the jth character of S$ and follows the (j-1)st character (if the latter exists).

- A substring referenced by colon-substring notation is dynamic. This means that the substring and its host can change length as a result of an assignment.

  During an assignment, a substring of length n can be filled with:

  - More than n characters. The substring expands accordingly, increasing its length and the length of its host.

  - Less than n characters. The substring contracts accordingly, decreasing its length and the length of its host.

  - Exactly n characters. No length adjustment is needed, and none occurs.

Examples

- Prints the value "TEST".

```
PRINT S$(3:6)
```

If S$ = "A/TEST CASE", then:

- Passes control to the line labeled 10 because the second character of S$ is a slant.

```
IF S$(2:2) = "/" THEN 10
```

- Inserts the value "STRANGE*" between the second and third characters of S$, giving S$ the value "A/STRANGE*TEST CASE".

```
LET S$(3:1) = "STRANGE*"
```

- Replaces the substring value "CASE" with the value "EXAMPLE", giving S$ the value "A/TEST EXAMPLE".

```
LET S$(8:11) = "EXAMPLE"
```

MID$ Statement

Purpose	Replaces characters in a substring of a host string variable with some characters from another string.

Format MID$(svar , pos , length) = string

svar	String identifier denoting the host variable that contains the substring being referenced. SVAR cannot be a substring expressed by colon substring notation or a MID$ reference.
pos	Numeric expression whose value specifies the first position in the host string to receive a replacement character.
length	Optional numeric expression whose value specifies the substring length.
string	String expression whose value contains the characters used in the replacement.

Remarks

- The values of POS and LENGTH are rounded to the nearest integers. Denote these integers by j and k, respectively.

- The k-character substring of SVAR beginning with position j is replaced by the first k characters of the value of STRING. That is, positions j through (j + k - 1) of SVAR are replaced by the first k characters of the value of STRING.

- If j is less than one or k is less than zero, a runtime error results.

- If j is greater than the length of the host string, or k is equal to zero, then the value of the host string is not altered.

Examples

- Replaces the substring value "B/CD" with the value "X*YZ" giving S$ the value "AX*YZ EF".

  ```
  If S$ = "AB/CD EF" and T$ = "X*YZ" then:
  MID$(S$,2,4) = T$
  ```

- Replaces the substring value "D " with the value "X*" giving S$ the value "AB/CX*EF".

  ```
  If S$ = "AB/CD EF" and T$ = "X*YZ" then:
  MID$(S$,5,2) = T$
  ```

  The substring referenced in a MID$ statement is static. This means that the substring and its host never change length as a result of the execution of a MID$ statement.

  As a result, there are some cases that do not fit the general description just provided. These involve instances where the relative lengths of the strings involved do not mesh.

  Suppose the length of the receiving substring exceeds the length of the string of replacement characters. The string of replacement characters replaces the left portion of the substring, leaving the rest of the substring unaltered.

- The MID$ reference attempts to replace the substring value "CD" with the value "*". The asterisk replaces the letter C, leaving the letter D as is. S$ now has the value "AB*D".

  ```
  LET S$ = "ABCD"
  LET MID$(S$,3,2) = "*"
  ```

Examples (cont)

- The first three characters of the substring value "REM" are replaced by the value "OUR". As a result, T$ has the value "POURED".

  ```
  LET T$ = "PREMED"
  LET MID$(T$,2,4) = "OUR"
  ```

  If the specified substring extends beyond the end of the host string, a runtime error does not result. Instead, the MID$ statement performs the requested replacement until the end of the host string, ignoring references to nonexistent positions.

- Replaces the substring value "CD" with the first two characters of the value "/YZ". S$ now has the value "AB/Y".

  ```
  If S$ = "ABCD" , then:
  LET MID$(S$,3,6) = "/YZ"
  ```

- Treats the substring length as if it were two instead of six. The first character of the substring value "CD" is replaced by the value " ". As a result, S$ has the value "AB D".

  ```
  If S$ = "ABCD" , then:
  LET MID$(S$,3,6) = " "
  ```

- Both colon-substring notation and the MID$ statement can be used to make the same assignment.

  Suppose the variables S$ and T$ have lengths m and n, respectively. Let j and k be integers, with (1 <= j <= m). Let x denote the minimum of (k, m - j + 1, n).

  The following two statements are equivalent.

  ```
  LET MID$(S$,j,k) = T$

  LET S$(j:j + x - 1) = T$(1:x)
  ```

  A MID$ reference appearing in an expression denotes a string library function call. This function is described later in this chapter.

Substring Manipulation Functions

The substring manipulation functions provide a way to reference substrings that have specified characteristics.

LEN Function

Purpose	Returns the length of the value of a string argument.
Format	LEN(string)
	string String expression. The length of the value of this expression is returned.
Remarks	The value returned is a nonnegative integer.

Examples

- The following are examples of the LEN function.

LEN("CASE#1")	Returns the value 6.
LEN("")	Returns the value 0.
LEN("HOME RUN")	Returns the value 8.
LEN("""")	Returns the value 1.

INSTR Function

Purpose	Returns the position within a host string at which the first occurrence of a specified substring is found.
Format	INSTR(pos , string , substring)

pos	Optional numeric expression whose value specifies the position within the host string at which the search begins.
string	String expression whose value is the host string being searched.
substring	String expression whose value is the substring to be located.

Remarks

- The value returned is always an integer.

- If POS is omitted, the search for the value of SUBSTRING begins by default at the first position of the value of STRING. Otherwise, the value of POS is rounded to the nearest integer j, and the search begins with position j.

- Only an occurrence of the substring that begins at or after position j is located. The function returns the position of the first character of this occurrence. Occurrences that follow the one located, or begin before position j, are not found.

- If j is less than one or greater than the maximum string length, a runtime error results.

- If the host string is the null string, j exceeds the length of the host string, or the substring is not found, then a value of zero is returned.

- If the substring to be located is the null string, then the value j is returned. That is, the null string is found at the first position searched.

NOTE

Repeated occurrences of a specified substring can be located by modifying the value of the search position argument after each success and continuing the search.

Examples

- Since the first argument is omitted, the search begins with the first position of "BANANA". The first occurrence of "ANA" begins at position two, so the function returns the value 2. The second occurrence of "ANA" is not located.

  ```
  INSTR("BANANA","ANA")
  ```

- Returns the value 0 because the substring "ZIPCODE" is not contained within the value of S$.

  ```
  If S$ = "NAME/ADDRESS/CITY/STATE/ZIP", then:
  INSTR(S$,"ZIPCODE")
  ```

  Returns the value 10, which is the position of the first "E" that occurs at or after position five.

  ```
  If S$ = "NAME/ADDRESS/CITY/STATE/ZIP", then:
  INSTR(5,S$,"E")
  ```

  Addresses the substring "NAME/" using colon-substring notation. The INSTR function reference locates the position of the first slant. This example is equivalent to the colon-substring reference S$(1:5).

  ```
  If S$ = "NAME/ADDRESS/CITY/STATE/ZIP", then:
  S$(1: INSTR(S$,"/") )
  ```

LEFT$ Function

Purpose Returns a left portion of the value of a host string.

Format LEFT$(string , length)

string	String expression whose value is the host string.
length	Numeric expression whose value specifies the length of the leading substring to be returned.

Remarks

- The value of LENGTH is rounded to the nearest integer j. The value returned is a string containing the first j characters of the host string.

- IF j is negative, a runtime error results. If j is zero, the null string is returned. If j exceeds the length of the host string, then the entire host string is returned.

Examples

- References the first four characters of the value of S$. The value "NEWS" is returned.

```
If S$ = "NEWS|WEATHER-SPORTS.", then:
LEFT$(S$,4)
```

- Addresses the left portion of the value of S$, ending with the first occurrence of a hyphen. The value "NEWS|WEATHER-" is returned.

```
If S$ = "NEWS|WEATHER-SPORTS.", then:
LEFT$(S$, INSTR(S$,"-") )
```

- References the left half of the value of S$. The value "NEWS|WEATH" is returned.

```
If S$ = "NEWS|WEATHER-SPORTS.", then:
LEFT$(S$, LEN(S$)\2 )
```

MID$ Function

Purpose Returns a specified portion of a host string.

Format MID$(string , pos , length)

string	String expression whose value is the host string.
pos	Numeric expression whose value specifies the position within the host string of the first character of the string to be returned.
length	Optional numeric expression whose value is the length of the string to be returned.

Remarks

- The values of POS and LENGTH are rounded to the nearest integers. Denote these integers by j and k, respectively. The k-character substring that begins in position j of the host string is returned.

- If j is less than one or k is less than zero, a runtime error results. If j is greater than the length of the host string, or k is equal to zero, then the null string is returned.

- If the length argument is omitted, or if the substring referenced extends beyond the end of the host string, then the substring running from position j through the end of the host string is returned.

- If a MID$ reference appears to the left of the equal sign of an assignment statement, then it represents the MID$ statement.

Examples

- References the 8-character substring of S$ beginning in position five. The function returns the value "US AT WO".

  ```
  If S$ = "GENIUS AT WORK", then:
  MID$(S$,5,8)
  ```

- References the 6-character substring of S$ that begins in position one. The function returns the value "GENIUS".

  ```
  If S$ = "GENIUS AT WORK", then:
  MID$(S$,1,6)
  ```

- Attaches the right portion of S$ beginning with position five to the string "HE SAW ". The value "HE SAW US AT WORK" is assigned to the variable V$.

  ```
  If S$ = "GENIUS AT WORK", then:
  LET V$ = "HE SAW " + MID$(S$,5)
  ```

RIGHT$ Function

Purpose

Returns a substring that is a specified right portion of a host string.

Format

RIGHT$(string , length)

string	String expression whose value is the host string.
length	Numeric expression whose value specifies the length of the trailing substring to be returned.

Remarks

- The value of LENGTH is rounded to the nearest integer k. The value returned is the string containing the last k characters of the host string.
- IF k is negative, a runtime error results. If k is zero, the null string is returned. If k exceeds the length of the host string, then the entire host string is returned.

Examples

- References the last seven characters of S$. The value "SPORTS." is returned.

  ```
  If S$ = "NEWS-WEATHER|SPORTS.", then:
  RIGHT$(S$,7)
  ```

- References the right portion of S$ with the same number of characters as the position number of the first occurrence of a hyphen (position five). The value returned is "ORTS." not "-WEATHER|SPORTS.".

  ```
  If S$ = "NEWS-WEATHER|SPORTS.", then:
  RIGHT$(S$, INSTR(S$,"-") )
  ```

- References the right half of S$. The value "ER|SPORTS." is returned.

  ```
  If S$ = "NEWS-WEATHER|SPORTS.", then:
  RIGHT$(S$, LEN(S$)/2 )
  ```

Notational Comparisons

Suppose the variable S$ has length n. Let j denote an integer, with (1 <= j <= n). Let k denote the length of a substring of S$, with (j + k - 1 <= n).

The LEFT$, MID$, and RIGHT$ functions can be expressed in colon-substring notation by the following formulas:

```
LEFT$(S$,j)   = S$(1:j)

MID$(S$,j,k) = S$(j:j + k - 1)

RIGHT$(S$,k) = S$(n - k + 1:n)
```

In addition, the LEFT$ and RIGHT$ functions can be expressed in terms of the MID$ function by the following formulas:

```
LEFT$(S$,j)   = MID$(S$,1,j)

RIGHT$(S$,k) = MID$(S$,n - k + 1)
```

Conversion Functions

The conversion functions change the representation of data. The first two functions listed accept string arguments and return related numeric values. The rest accept numeric arguments and return related string values.

ASC Function

Purpose	Returns the numeric ASCII code of the first character of the value of a string argument. This function complements the CHR$ function.
Format	ASC(string) string String expression. The first character of the value of this expression is converted to its corresponding ASCII code.
Remarks	The value returned is an integer in the range 0 through 255. If the value of STRING is the null string, a runtime error results.
Examples	Returns the value 83, which is the ASCII code for the uppercase S. ASC("SAMPLE")

VAL Function

Purpose	Returns the value of the leading numeric characters in the value of a string argument. This function complements the STR$ function.
Format	VAL(string) string: String expression whose value contains the leading numeric substring to be converted to a numeric constant.
Remarks	The value returned is a numeric representation of the leading numeric characters of the value x of STRING. Any leading spaces are ignored. If x does not contain a leading substring that can be interpreted as a numeric constant, or if x is the null string, then the integer 0 is returned.

Examples

- The following are examples of the VAL function.

```
VAL("   32RTZ")     Returns the value 32.

VAL("-47.5, 4.2")   Returns the value -47.5.
```

CHR$ Function

Purpose

Returns the literal character whose ASCII code is the value of a numeric argument. This function complements the ASC function.

Format

```
CHR$(code)
```

code	Numeric expression whose value specifies the ASCII code to be converted to a literal character.

Remarks

The value of CODE is rounded to the nearest integer j. The value returned is a string containing the single character with ASCII code j. If j is not in the range 0 through 255, a runtime error results.

Examples

This statement prints an asterisk, which has an ASCII code of 42.

```
PRINT CHR$(42.2)
```

HEX$ Function

Purpose Returns a character string containing a hexadecimal representation of the value of a numeric argument.

Format HEX$(number)

number	Numeric expression whose value is to be represented in printable hexadecimal form.

Remarks The value of NUMBER is rounded to the nearest integer j. The value returned is a string representation of the hexadecimal value of j.

Examples

- The following are examples of the HEX function.

  HEX$(63) Returns the character string "3F".

  HEX$(-1) Returns the character string "FFFFFFFFFFFFFFFF".

OCT$ Function

Purpose

Returns a character string containing an octal representation of the value of a numeric argument.

Format

OCT$(number)

number Numeric expression whose value is to be represented in printable octal form.

Remarks

The value of NUMBER is rounded to the nearest integer j. The value returned is a string representation of the octal value of j.

Examples

- The following are examples of the OCT$ function.

  OCT$(47)   Returns the character string "57".

  OCT$(-1)   Returns the character string "1777777777777777777777".

STR$ Function

Purpose

Returns a string representation of the value of a numeric argument. This function complements the VAL function.

Format

```
STR$(number)
```

number	Numeric expression whose value is to be converted to a string constant.

Remarks

The string returned has the same representation as would result from specifying NUMBER in a PRINT statement.

Examples

- This statement assigns the string constant "-44.7" to the variable A$.

  ```
  LET A$ = STR$(-44.7)
  ```

- This print statement returns the value * 47.3 *.

  ```
  PRINT "*";47.3;"*"
  ```

Miscellaneous String Functions

This section describes the remaining string library functions.

LCASE$ Function

Purpose	Converts the uppercase letters in the value of a string argument to their lowercase counterparts. This function complements the UCASE$ function.
Format	LCASE$(string) string String expression. The letters in the value of this expression undergo letter case conversion.
Remarks	The function returns the value of STRING with each uppercase letter replaced by its lowercase counterpart. The length of the returned value equals the length of the value of STRING.
Examples	This reference returns the value "act i of the new play". LCASE$("Act I of the New Play")

UCASE$ Function

Purpose	Converts the lowercase letters of a string argument to their uppercase counterparts. This function complements the LCASE$ function.
Format	UCASE$(string) string String expression. The letters in the value of this expression undergo letter case conversion.
Remarks	The function returns the value of STRING with each lowercase letter replaced by its uppercase counterpart. The length of the returned value equals the length of the value of STRING.
Examples	This reference returns the value "THE BASIC LANGUAGE". UCASE$("The BASIC Language")

SPACE$ Function

Purpose	Returns a string of spaces.
Format	SPACE$(length) length Numeric expression whose value specifies the length of the string of spaces to be produced.
Remarks	The value of LENGTH is rounded to the nearest integer j. The value returned is a string consisting of j spaces. If j is negative or greater than 65,535 (the maximum string length), a runtime error results.
Examples	This statement assigns the value "NAME ADDRESS" to S$. The function reference provides the three spaces.

```
LET S$ = "NAME" + SPACE$(3) +  "ADDRESS"
```

STRING$ Function

Purpose Returns a uniform string of characters.

Format STRING$(length , code)

STRING$(length , string)

length	Numeric expression whose value is the length of the uniform string to be produced.
code	Numeric expression whose value specifies the ASCII code of the character to be used in the uniform string.
string	String expression the first character of whose value is used in the uniform string.

Remarks

- In the first format, the values of LENGTH and CODE are rounded to the nearest integers. Denote these integers by j and k, respectively. The value returned is a uniform string containing j repetitions of the character with ASCII code k.
- In the second format, the value of LENGTH is rounded to the nearest integer j. The value returned is a uniform string containing j repetitions of the first character of the STRING.
- If the integer j, denoting the string length, is negative or greater than the maximum string length, a runtime error results.
- If the integer k, denoting the ASCII code, is not in the range 0 through 255, a runtime error results.
- A runtime error results if string is the null string.

Examples

- Returns the string value "!!!!!" since the exclamation point has an ASCII code of 33.

```
STRING$(5,33)
```

- Returns the string "####".

```
STRING$(4,"#1")
```

PARAMS$ Function

Purpose returns the parameter string that was specified in the SCL command used to execute the program. If no parameter string was specified, the null string is returned.

Format

```
PARAMS$
```

PARAMS$ has no arguments.

Remarks For more information about how a parameter string is specified when executing a NOS/VE program, see the SCL Object Code Management manual. To read about how a NOS/VE BASIC program is compiled and executed, see chapter 14.

Examples

- This command executes the program whose binary object code is stored in the file BIN in the working catalog. A PARAMS$ function reference within the program returns the value "BREAK TIME". Note that apostrophes delimit NOS/VE SCL strings, whereas quotation marks delimit NOS/VE BASIC strings.

```
EXECUTE_TASK  FILE=BIN  PARAMETER='BREAK TIME'
```

- This command executes the program whose binary object code is stored in the default binary file $LOCAL.LGO. A PARAMS$ function reference within the program returns the value "CODENAME". Note that no string delimiters are used with this format.

```
LGO  CODENAME
```

Files 13

Files 13

This chapter explains file usage in NOS/VE BASIC and describes the statements and library functions related to input and output.

A file is a sequence of records. A record is a sequence of values or a sequence of characters.

By accessing files, a BASIC program can:

- Retrieve data that has been previously stored in an exterior data set.
- Send data to an exterior data storage area.

The Input and Output chapter describes how a BASIC program can receive data from the terminal, access data from an interior data set, and send program output to the terminal.

Overview of the NOS/VE File System

This section gives a quick overview of the NOS/VE file system. The section contains the following topics:

Specifying Files
Using Temporary Files
Using Permanent Files
Using the Working Catalog

For a detailed discussion of the NOS/VE file system, see the NOS/VE System Usage manual.

NOS/VE organizes files hierarchically into catalogs. A catalog is a collection of file entries and catalog entries. Each NOS/VE file belongs to a catalog. Three catalogs are important to remember: the temporary catalog, $LOCAL; the master catalog; and the working catalog. The temporary catalog contains temporary files, and the master catalog contains permanent files and subcatalogs. The working catalog is the catalog you are currently using when logged in to NOS/VE.

Specifying Files

To specify a NOS/VE file, you specify a file reference or a portion of a file reference. The full file reference consists of several parts:

```
    :family.user_name.catalog.file_name.cycle.file_position
```

When you specify a file, you do not need to provide the entire file reference as long as you specify enough of the file reference to uniquely identify the file. For examples of specifying files, see the following sections.

Although the terms file, file name, file path, and file reference are often used interchangeably, they have different meanings. For descriptions of these terms and the other parts of a file reference, see appendix A, Glossary.

Using Temporary Files

The temporary catalog $LOCAL contains temporary files only. These files are preserved only throughout your NOS/VE session; they are discarded when you log out of NOS/VE.

You specify a temporary file by using $LOCAL in place of a user name and catalog name in a file reference as follows:

```
    $LOCAL.file_name
```

The following file paths specify several temporary files:

```
    $local.a

    $local.lgo

    $local.output

    $local.$output
```

$LOCAL is the default working catalog; that is, $LOCAL is the working catalog when you log in to NOS/VE. You can change the working catalog by using the SET_WORKING_CATALOG command, which is discussed later in the section titled Using the Working Catalog.

NOS/VE creates the following standard files in your $LOCAL catalog for each job:

```
    $ECHO
    $ERRORS
    $INPUT
    $LIST
    $OUTPUT
    $RESPONSE
```

The standard files provide a default file for use by job files and other files.

Using Permanent Files

The master catalog associated with a user name contains permanent files and subcatalogs. NOS/VE preserves permanent files across job executions and system deadstarts.

You specify a permanent file in your master catalog by using your user name or the keyword $USER in a file reference as follows:

```
.user_name.catalog.file_name

$USER.catalog.file_name
```

The following file paths specify several permanent files:

```
.joe_user.prolog

$user.epilog

.joe_user.time_cards.progl

$user.time_cards.lgo
```

You specify a permanent file belonging to another user by using the corresponding user name in the file reference. The following file paths specify permanent files belonging to different user names:

```
.joe_user.prolog

.smith.epilog

.bproject.time_cards.progl
```

The following file paths specify a permanent file named PROG1. The file belongs to the user name JOE_USER in the V01 family:

```
:v01.joe_user.progl

.joe_user.progl

$user.progl
```

You can only use $USER to specify the preceding file if you are logged in to the user name JOE_USER.

Using the Working Catalog

The working catalog is the catalog NOS/VE assumes if you do not specify a catalog on a file reference. When you log in, $LOCAL is, by default, the working catalog.

To change the working catalog, use the SET_WORKING_CATALOG (SETWC) command. The following commands show several examples of setting the working catalog:

```
set_working_catalog catalog=$user

set_working_catalog catalog=$local

set_working_catalog catalog=$user.time_cards
```

Use the working catalog to reduce typing and to increase simplicity in referencing your NOS/VE files. For example, you might have a catalog, $USER.TIME_CARDS containing the following files:

```
DATA_1
DATA_2
PROG1
PROG2
```

To compile PROG1 when your working catalog is $LOCAL, you could use the following command:

```
basic input=$user.time_cards.prog1
```

However, by setting the working catalog to $USER.TIME_CARDS, the compilation command is much simpler:

```
set_working_catalog catalog=$user.time_cards
basic input=prog1
```

Within a BASIC program, the working catalog affects the files opened with the OPEN statement. You can use the BASIC SCL statement to set the working catalog within a BASIC program, as shown in the following examples:

```
SCL "SET_WORKING_CATALOG $USER"

SCL "SET_WORKING_CATALOG $LOCAL"

SCL "SET_WORKING_CATALOG $USER.TIME_CARDS"
```

The following table shows several BASIC OPEN statements and the corresponding file references of the files opened if the working catalog is set to $USER and you are logged in to the user name JOE_USER in the V01 family:

BASIC OPEN Statement	Corresponding Opened File
OPEN "A" FOR INPUT AS #1	:v01.joe_user.a
OPEN "SMITH.A" FOR INPUT AS #1	:v01.joe_user.smith.a
OPEN ".SMITH.A" FOR INPUT AS #1	:v01.smith.a
OPEN ":V02.SMITH.A" FOR INPUT AS #1	:v02.smith.a
OPEN "A.3" FOR INPUT AS #1	:v01.joe_user.a.3
OPEN "$LOCAL.A" FOR INPUT AS #1	$local.a

The following table shows several BASIC OPEN statements and the corresponding file references of the files opened if the working catalog is set to $LOCAL and you are logged in to the user name JOE_USER in the V01 family:

BASIC OPEN Statement	Corresponding Opened File
OPEN "A" FOR INPUT AS #1	$local.a
OPEN ".SMITH.A" FOR INPUT AS #1	:v01.smith.a
OPEN "$USER.A" FOR INPUT AS #1	:v01.joe_user.a
OPEN ".JOE_USER.A" FOR INPUT AS #1	:v01.joe_user.a
OPEN "$LOCAL.A" FOR INPUT AS #1	$local.a
OPEN "SMITH" FOR INPUT AS #1	$local.smith
OPEN ":V02.SMITH.A" FOR INPUT AS #1	:v02.smith.a
OPEN "$USER.A.3" FOR INPUT AS #1	:v01.joe_user.a.3

The following table shows several BASIC OPEN statements and the corresponding file references of the files opened if the working catalog is set to $USER.TIME_CARDS and you are logged in to the user name JOE_USER in the V01 family:

BASIC OPEN Statement	Corresponding Opened File
OPEN "A" FOR INPUT AS #1	:v01.joe_user.time_cards.a
OPEN ".SMITH.A" FOR INPUT AS #1	:v01.smith.a
OPEN "$USER.A" FOR INPUT AS #1	:v01.joe_user.a
OPEN ".JOE_USER.A" FOR INPUT AS #1	:v01.joe_user.a
OPEN "$LOCAL.A" FOR INPUT AS #1	$local.a
OPEN "SMITH.A" FOR INPUT AS #1	:v01.joe_user.time_cards.smith.a
OPEN ":V02.SMITH.A" FOR INPUT AS #1	:v02.smith.a
OPEN "A.3" FOR INPUT AS #1	:v01.joe_user.time_cards.a.3

For a description of the BASIC SCL statement, see chapter 5, Decision and Branching.

To determine your current working catalog, use the $CATALOG function with the DISPLAY_VALUE (DISV) command:

```
    display_value value=$catalog
```

The following dialog shows several DISPLAY_VALUE commands and the corresponding NOS/VE responses:

```
    /set_working_catalog catalog=$local
    /display_value value=$catalog
    :$LOCAL

    /set_working_catalog catlog=$user
    /display_value value=$catalog
    :V01.JOE_USER

    /set_working_catalog catlog=$user.basic
    /display_value value=$catalog
    :V01.JOE_USER.BASIC
```

Overview of BASIC File Usage

A NOS/VE BASIC program can read data from and write data to NOS/VE files. The file can be connected to a terminal or located on a mass storage device.

The NOS/VE standard files $INPUT and $OUTPUT are available to a BASIC program during execution.

$INPUT specifies the file from which programs read input when no other file is specified. The default connection for the standard input file is the NOS/VE temporary file INPUT. For interactive mode, input is received from the terminal.

$OUTPUT specifies the file to which program output is written when no other file is specified. The default connection for the standard output file is the NOS/VE listing file OUTPUT. For interactive mode, program output appears at the terminal.

A BASIC program can read and write both coded and binary data.

Coded data is stored as a sequence of ASCII codes, one for each character in the data value. Binary data is stored using the computer´s internal binary representation.

Binary data can be processed more efficiently than coded data because no translation is required. However, only coded data can be printed in readable form.

In general, binary data is written to a file only if the data is to be read later by a BASIC program, and a printed copy of the file is not needed.

BASIC provides both sequential and random access methods for reading and writing files.

Sequential access is used for I/O involving files that contain coded data. A sequential file can be accessed with an I/O mode of INPUT, OUTPUT, or APPEND. Since a terminal is a sequential file, I/O through a terminal is a special case of general I/O using files.

Random access is used for I/O involving files that contain binary data. A random file must be accessed with an I/O mode of RANDOM.

It is often useful to think of a data file as a named collection of related records, where a record is a set of related data items called fields.

For example, a payroll file is a collection of employee records. A given employee record might contain fields corresponding to the name, sex, social security number, grade level, and salary.

A record in a sequential file is a single line of data. An individual value in the line can be thought of as a field.

A record in a random file is a sequence of bytes. A field is a specified sub-sequence of a record.

When a sequential file is accessed:

- With an I/O mode of INPUT, records are read sequentially from the beginning of the file.
- With an I/O mode of OUTPUT, records are written sequentially from the beginning of the file. Any data stored prior to the current file access is lost.
- With an I/O mode of APPEND, records are written sequentially following the last preexisting record. Any data stored prior to the current file access is preserved.

A sequential file rewind is possible only by closing and reopening the file.

When a random file is accessed:

- Records can be read and written in any order.
- An explicit format for individual fields within a record is specified.

Random I/O:

- Requires more specifications in a program than does sequential I/O.
- Requires more knowledge of the nature of the program data than does sequential I/O.
- Is usually inappropriate for data intended for interchange with processors other than NOS/VE BASIC.
- Cannot be used with terminal files.

Channel Numbers

Purpose	Associates a NOS/VE file with a BASIC program.

Format

channel

#	Sometimes optional number sign.
channel	Numeric expression whose value, when rounded to the nearest integer, specifies a channel number denoting a file.

Remarks

- Usually, a program can access a file only by explicitly opening a channel. A channel number is associated with a file using the OPEN statement. This number is used to denote both the channel and the file that is accessed through the channel. Thus, a file is usually accessed by a channel number reference rather than by its name.

- In general, a channel number must be between 0 and 99. However, some of the BASIC statements described in this chapter do not allow a channel number of 0. Appropriate details are included with the discussion of each statement.

- When a channel number of 0 is allowed, it denotes either the standard file $INPUT or the standard file $OUTPUT, depending on the context.

- The number sign in a channel number reference is:

  - Mandatory if the reference appears in an INPUT, LINE INPUT, PRINT, PRINT USING, or WRITE statement.

  - Optional if the reference appears in a CLOSE, FIELD, GET, OPEN, PUT, or WIDTH statement.

Remarks (cont)

- A number sign cannot precede a channel number that is used as an argument to a library function.

- Each channel has a record pointer that points to the last record in the corresponding file that was completely processed through the channel using a read or write.

- A file can be accessed through more than one channel concurrently. Each channel uses a separate NOS/VE instance of open and so, maintains its own file position. Each file operation updates the file position for its channel; it does not update the file positions of other channels. Thus, two or more channels can be reading a file with no effect on each other. However, if one of the concurrent opens writes to the file, the system updates file positions for all opens. This may lead to unexpected results and should be avoided.

- A file currently accessible through at least one open channel is called open. Closing all channels for a file is referred to as closing the file.

- A runtime error results if:

  - A channel number is outside of the range allowed.

  - A channel number specified in a library function call or in a BASIC statement other than an OPEN statement does not denote an open channel.

- Note that only the file PRINT (created in the working catalog) and the standard files $INPUT and $OUTPUT can be accessed from a BASIC program without using a channel number reference.

- Statements involving the file PRINT are provided for compatibility with popular microcomputer versions of BASIC.

OPEN Statement

Purpose Opens a channel through which a BASIC program can access either an existing NOS/VE file, or a new file that is created.

Format

```
(A) OPEN filname  FOR  keymode  AS  chanref  LEN = length

(B) OPEN strmode , chanref , filname , length
```

filname	String expression whose value is a NOS/VE file reference specifying the file for which a channel is being opened. This file name can be a path identifying any NOS/VE file. For example, you can specify a temporary file, a permanent file, or a file in the working catalog. If the file does not exist and the I/O mode is OUTPUT, APPEND, or RANDOM, a new file is created.
chanref	Channel number reference. The specified channel number must be between 1 and 99, inclusive. Use of the number sign (#) is optional. A runtime error results if the channel number is already in use.
keymode	Optional keyword defining the I/O mode to be used whenever the file is accessed through the specified channel. Possible values: INPUT, OUTPUT, or APPEND. If this parameter is omitted, the keyword FOR is also omitted, and an I/O mode of RANDOM is specified by default.
strmode	String expression. The first character of the value of this expression defines the I/O mode to be used whenever the file is accessed through the specified channel. Possible values: I, O, or R, denoting INPUT, OUTPUT, and RANDOM, respectively. No APPEND mode is possible using format (B).
length	Optional numeric expression whose value, when rounded to the nearest integer, establishes the record length (in bytes) for FILNAME.

Remarks

- The OPEN statement:
  - Assigns a channel number to denote the file.
  - Defines the I/O mode to be used whenever the file is accessed through the channel.
  - Allocates a buffer associated with the channel.
  - Establishes the record length (in bytes) for the file. The length of the buffer associated with the channel is initially equal to the record length.
- A runtime error results if:
  - The NOS/VE file organization, the NOS/VE file access mode, and the BASIC I/O mode are not compatible.
  - A file is opened with an I/O mode of INPUT, but is not attached.
  - The specified length is less than one byte, or exceeds the maximum record length allowed for NOS/VE files (2^42) -1 bytes.
  - FILNAME is a preexisting file and the specified length exceeds the file´s already established maximum record length attribute.

Remarks

- If the LENGTH parameter is omitted:
  - The keyword LEN and the equal sign are also omitted from format (A).
  - The comma preceding LENGTH is also omitted from format (B).
  - And FILNAME is a preexisting file, the file's already established record length is used.
  - And FILNAME is being created, a maximum record length of 128 bytes is established.
- An existing file that is accessed with an I/O mode of:
  - INPUT, OUTPUT, or APPEND, must have a NOS/VE file organization of SEQUENTIAL.
  - RANDOM, must have a NOS/VE file organization of BYTE ADDRESSABLE.
- A file that is created with an I/O mode of:
  - INPUT, OUTPUT, or APPEND, is assigned a NOS/VE file organization of SEQUENTIAL.
  - RANDOM, is assigned a NOS/VE file organization of BYTE ADDRESSABLE.
- Note that the two formats for the OPEN statement are not equivalent since format (B) does not provide for an I/O mode of APPEND.

Examples

Each example below is displayed using both formats, if possible.

- The following examples open the file named EAM in the working catalog for sequential input as channel #4, using the file´s established record length or the default length of 128:

```
OPEN "EAM" FOR INPUT AS #4
OPEN "I", #4, "EAM"
```

- The following examples open the file named PAY in the working catalog for random I/O as channel #7, with a record length of 64 bytes:

```
OPEN "PAY" AS 7 LEN=64
OPEN "R", 7, "PAY", 64
```

- The following example opens the file named MY in the working catalog for sequential append as channel #2, using the files established record length or the default length of 128. This example cannot be expressed using format (B):

```
OPEN "MY" FOR APPEND AS #2
```

- The following examples open the file named PAY_DATA in the working catalog for sequential input as channel #5, using the file´s established maximum record length or the default length of 128. The file is in the catalog $USER:

```
OPEN "$USER.PAY_DATA" FOR INPUT AS #5
OPEN "I", #5, "$USER.PAY_DATA"
```

- The following example opens cycle #76 of the file named DATA in the working catalog for sequential input as channel #180. The file is in the master catalog of the user name NORRIS:

```
OPEN ".NORRIS.DATA.76" FOR INPUT AS #180
```

CLOSE Statement

Purpose

Closes channels that were opened by the OPEN statement.

Format

CLOSE chanlist

chanlist	Optional list of channel number references that are separated by commas. Specified channel numbers must be between 1 and 99, inclusive. Use of number sign (#) is optional.

Remarks

The channels with the specified channel numbers are closed. For a channel with an I/O mode of INPUT or RANDOM, the data in the associated buffer is purged. For a channel with an I/O mode of OUTPUT or APPEND, the data in the associated buffer is written to the corresponding file. If CHANLIST is omitted, this statement closes all channels that have been opened in the executing program by OPEN statements.

Examples

- This statement closes the channels having channel numbers 2, 44, and 5.

  ```
  CLOSE 2,#44,5
  ```

- This statement closes all channels that have been opened by OPEN statements.

  ```
  CLOSE
  ```

LOC Function

Purpose

Returns the number of the record to which the record pointer for the specified channel currently points.

Format

LOC(channel)

channel	Numeric expression whose value, when rounded to the nearest integer, specifies a channel number. This channel number must be between 1 and 99, inclusive. A number sign (#) cannot precede a channel number that appears as an argument to a library function.

Remarks

If the specified channel has an I/O mode of:

- INPUT, the function returns the number of records (lines) that have been read through the channel since the channel was established. The integer 1 (not 0) is returned if no record has been read since the channel was established.

- OUTPUT or APPEND, the function returns the number of records (lines) that have been written through the channel since the channel was established. The integer 1 (not 0) is returned if no record has been written since the channel was established.

- RANDOM, the function returns the number of the record (byte sequence) that has most recently been accessed through the channel using a GET or PUT statement. The integer 0 is returned if no record has been accessed since the channel was established.

Examples

The IF-THEN statement causes a branch to the line labeled 150 if more than 25 records are read through channel #10.

```
OPEN "ROSTER" FOR INPUT AS #10
IF LOC(10) > 25 THEN GOTO 150
```

Remember that the LOC function only returns the number of a completely processed record. A comma or semicolon at the end of a print list causes only part of a record to be written to a sequential file. In such a case, the LOC function is not incremented until writing is completed.

EOF Function

Purpose

Determines whether or not a specified channel record pointer has reached end-of-file status.

Format

EOF(channel)

channel	Numeric expression whose value, when rounded to the nearest integer, specifies a channel number denoting the open file whose status is to be checked. This channel number must be between 0 and 99, inclusive. A number sign (#) cannot precede a channel number that appears as an argument to a library function.

Remarks

- The value returned is the integer -1, for true, if end-of-file status has been reached. Otherwise, the integer 0, for false, is returned. If a channel number of 0 is specified, the status check applies to the standard file $INPUT.

- A channel with an I/O mode of OUTPUT or APPEND (including a terminal) has always reached end-of-file status. A channel with an I/O mode of RANDOM never reaches end of file status.

Examples

In this program fragment, the line labeled 10 avoids the runtime error that would result from an attempt to read more data than is currently stored in the file MIND.

```
OPEN "MIND" FOR INPUT AS #3
10  IF EOF(3) THEN END
LINE INPUT #3, S$
.
.
GOTO 10
```

Sequential I/O

This section discusses the BASIC statements that are used for Input/Output using sequential files.

Most of these statements are described for the special case of terminal I/O in the Input and Output chapter. This section concentrates on aspects that pertain specifically to files. You are then directed, as needed, to the appropriate description in the Input and Output chapter for further details.

INPUT Statement

Purpose Reads data from a sequential file.

Format INPUT chanref , varlist

chanref	Channel number reference specifying the open sequential file from which data is read. The specified channel number must be between 0 and 99, inclusive. Use of the number sign (#) is mandatory. A runtime error results if this channel does not permit access with an I/O mode of INPUT. If a channel number of 0 is specified, input is read from the standard file $INPUT.
varlist	List of variables that are separated by commas. This input variable list contains the variables that receive values from the specified file.

Remarks

- The INPUT statement reads the next record from the specified file. Commas are used to separate the values in a record of a sequential file. These values are assigned to the corresponding variables in the input variable list. There must be a one-to-one correspondence between values in the record and variables in the input variable list.

- A numeric variable can be assigned only a numeric value. Mixing of integer and real data types is handled exactly as it is handled in an assignment statement. Thus, an integer input for a real variable is converted to type real. A real input for an integer variable is rounded to the nearest integer.

- If a value in a record begins with a quote, the value is assumed to be a quoted string constant. Modified unquoted string constants (defined below) can also appear in a record.

- A modified unquoted string constant is an unquoted string constant that can contain an apostrophe or a colon. Since the compiler never sees a record, the restriction that these two characters be used only as delimiters (when outside of a quoted string) can be relaxed.

Examples

This program fragment reads values from the file named DATAFILE until the end of the file is reached. Note that each record must contain an ordered pair of values, one string and one real.

```
OPEN "DATAFILE" FOR INPUT AS #4
WHILE NOT EOF(4)
  INPUT #4, ITEM.NAME$,DOLLAR.VALUE
  .
  .
  .
WEND
```

Combining the format for input from the terminal with the format just described produces the following general format for the INPUT statement:

```
INPUT ; chanref , prompt  varlist
```

If CHANREF and the subsequent comma are omitted, input is read from the standard file $INPUT.

If the file specified by CHANREF is connected to a terminal, the INPUT statement works as described in the Input and Output chapter.

If the file specified in the INPUT statement is not connected to a terminal:

- And no record is available when requested, a runtime error results.
- The PROMPT parameter is ignored during execution.

LINE INPUT Statement

Purpose	Reads an entire line from a sequential file into a string variable.

Format

```
LINE INPUT  chanref , strvar
```

chanref	Channel number reference specifying the open sequential file from which data is read. The specified channel number must be between 0 and 99, inclusive. Use of the number sign (#) is mandatory. A runtime error results if this channel does not permit access with an I/O mode of INPUT. If a channel number of 0 is specified, input is read from the standard file $INPUT.
strvar	String variable that receives the input line.

Remarks

The LINE INPUT statement reads all the characters in the current line, including leading and trailing spaces, until the next carriage return is reached. A quotation mark is treated exactly like any other character, even if it is the first character.

Examples

This program fragment reads lines through channel #2 until the channel record pointer reaches the end of the file MESSAGES.

```
OPEN "MESSAGES" FOR INPUT AS #2
WHILE  NOT EOF(2)
    LINE INPUT #2, NOTE$
    .
    .
    .
WEND
```

Combining the format for line input from the terminal with the format just described produces the following general format for the LINE INPUT statement:

```
LINE INPUT ; chanref , prompt strvar
```

If CHANREF and the subsequent comma are omitted, line input is read from the standard file $INPUT.

If the file specified by CHANREF is connected to a terminal, the LINE INPUT statement works as described in the Input and Output chapter. For more information about the PROMPT parameter and line input from the terminal, see chapter 10.

If the file specified in the LINE INPUT statement is not connected to a terminal:

- And no line is available when requested, a runtime error results.
- The PROMPT parameter is ignored during execution.

WIDTH Statement

Purpose	Sets the page width for output that is written to a sequential file.

Format

```
(A)   WIDTH  chanref , pgwidth
(B)   WIDTH  spcfile , pgwidth
```

pgwidth	Numeric expression whose value, when rounded to the nearest integer, specifies the page width to be used whenever output is sent through the specified channel, or to the specified special file.
chanref	Channel number reference specifying an open sequential file. The specified channel number must be between 1 and 99, inclusive. Use of the number sign (#) is optional. A runtime error results if this channel does not permit access with an I/O mode of OUTPUT or APPEND.
spcfile	String expression whose value denotes one of two special files that are not normally accessed through channel number references. Possible values: "PRINT" or "OUTPUT". These values can use both lowercase and uppercase letters and can be followed by trailing spaces.

Remarks

- The page width is the maximum number of characters that can be written before a carriage return is generated.

- Format (A) defines the page width for output through the specified channel.

- Format (B) defines the page width for the standard file $OUTPUT if the value "OUTPUT" is specified and for the file PRINT in the working catalog if the value "PRINT" is specified. If necessary, the file PRINT is created.

- If CHANREF and the subsequent comma are omitted, the page width of the standard file $OUTPUT is set.

- If the page width:

  - Is different than the record length for the file being accessed, the buffer size is adjusted to accommodate the page width. This adjustment does not redefine the file's record length.

  - Exceeds the NOS/VE maximum page width, the maximum is used.

  - Is less than 14 (the length of a print zone), a runtime error results.

- If the length of a value to be written:

  - Exceeds the space available on the current line, but is less than the page width, then the value is printed at the beginning of the next line.

  - Exceeds the page width, as much of it as will fit on the current line is printed, and the value is continued on as many subsequent lines as are needed.

Examples

- Each of these statements sets the page width to 72 characters. This page width is to be used whenever output is sent through channel 8.

```
WIDTH #8, 72
WIDTH 8, 4*18
```

- This statement sets the page width for the file PRINT in the working catalog to 65 characters. If necessary, file PRINT is created.

```
WIDTH "PRINT", 65
```

PRINT Statement

Purpose Writes data to a sequential file.

Format PRINT chanref , printlist

chanref	Channel number reference specifying the open sequential file to which data is written. The specified channel number must be between 0 and 99, inclusive. Use of the number sign (#) is mandatory. A comma after the channel number reference is required. A runtime error results if this channel does not permit access with an I/O mode of OUTPUT or APPEND. If a channel number of 0 is specified, output is written to the standard file $OUTPUT.
printlist	Optional list of expressions and format function references (the print list).

Remarks

- The expressions in the print list are evaluated, and their values are written in sequence to the specified file. The format of the output is specified by the punctuation and format functions appearing in the print list.
- Output is not transmitted until an end of line is generated. This happens either when the specified channel is closed, or when control reaches the end of a printlist that does not end with a comma, semicolon, or format function.
- If CHANREF and the subsequent comma are omitted, output is sent to the standard file $OUTPUT.
- A comma after the channel number reference is required even if the print list is omitted.
- If the file specified by CHANREF is connected to a terminal, the PRINT statement works as described in the Input and Output chapter. For more information about the PRINTLIST parameter and the form of output produced using the PRINT statement, see chapter 10.

Examples This program fragment writes 40 records to the file named TOP40. Each record consists of an integer and a string that are separated by a comma.

```
DIM RANK(1:40),TITLE$(1:40) : DEFINT I,R
OPEN "TOP40" FOR OUTPUT AS #5
PRINT #5, "List of rank and title"
PRINT #5,
FOR I = 1 TO 40
    PRINT #5, RANK(I);", ";TITLE$(I)
NEXT I
```

LPRINT Statement

Purpose Writes data to the file PRINT in the working catalog. If necessary a file with this name is created.

Format LPRINT printlist

printlist	Optional list of expressions and format function references (the print list).

Remarks

- The expressions in the print list are evaluated, and their values are written in sequence to the file PRINT. The format of the output is specified by the punctuation and format functions appearing in the print list.

- Output is not transmitted until an end of line is generated. This happens either when the specified channel is closed, or when control reaches the end of a printlist that does not end with a comma, semicolon, or format function.

- The LPRINT statement works like the PRINT statement as described in the Input and Output chapter. For more information about the PRINTLIST parameter and the form of output produced using the PRINT statement, see chapter 10.

Examples If TITLE$ has the value "WONDERS, INC.", the sentence below is written to the file PRINT in the working catalog. If necessary, file PRINT is created.

```
LPRINT "THE BOOK <";TITLE$;"> IS NOT AVAILABLE."

THE BOOK <WONDERS, INC.> IS NOT AVAILABLE.
```

PRINT USING Statement

Purpose	Writes data to a sequential file using a specified display format.

PRINT chanref , USING formstr ; printlist

chanref	Channel number reference specifying the open sequential file to which data is written. The specified channel number must be between 0 and 99, inclusive. Use of the number sign (#) is mandatory. A runtime error results if this channel does not permit access with an I/O mode of OUTPUT or APPEND. If a channel number of 0 is specified, output is written to the standard file $OUTPUT.
formstr	Required string expression whose value specifies the format of the output. Also referred to as the format string.
printlist	Nonempty list of expressions. Also referred to as the print list.

Remarks

- The expressions in the print list are evaluated. Their values are written in sequence to the specified file, using the format defined in the format string.

- Output is not transmitted until an end of line is generated. This happens either when the specified channel is closed, or when control reaches the end of a printlist that does not end with a comma or semicolon.

- If CHANREF and the subsequent comma are omitted, output is sent to the standard file $OUTPUT.

- If the file specified by CHANREF is connected to a terminal, the PRINT USING statement works as described in the Input and Output chapter. For more information about the FORMSTR and PRINTLIST parameters, and the form of output produced using the PRINT USING statement, see chapter 10.

Examples

If D has the value 1289.431 , this program fragment writes the output to the file PAYMENT.

```
OPEN "PAYMENT" FOR OUTPUT AS #3
PRINT #3, USING "$$###.##"; D
```

The output from the program fragment appears below:

```
$1289.43
```

LPRINT USING Statement

Purpose	Writes data to the file PRINT in the working catalog, using a specified format string. If necessary a file with this name is created.

Format

```
LPRINT USING  formstr ; printlist
```

formstr	Required string expression whose value specifies the format of the output. Also referred to as the form list.
printlist	Nonempty list of expressions. Also referred to as the print list.

Remarks

- The expressions in the print list are evaluated, and their values are written in sequence to the file PRINT, using the format defined in the format string.
- Output is not transmitted until an end of line is generated. This happens either when the specified channel is closed, or when control reaches the end of a printlist that does not end with a comma or semicolon.
- The LPRINT USING statement works like the PRINT USING statement as described in the Input and Output chapter. For more information about the FORMSTR and PRINTLIST parameters, and the form of the output produced using the PRINT USING statement, see chapter 10.

Examples

If POPULATION% has the value 2576123, this statement writes the output to the file PRINT in the working catalog. If necessary, this file is created.

```
LPRINT USING "#####,###"; POPULATION%
```

The output appears below:

```
2,576,123
```

WRITE Statement

Purpose

Writes data to a sequential file in a form that resembles a list of BASIC constants, complete with separating commas.

Format

```
WRITE  chanref , writelist
```

chanref	Channel number reference specifying the open sequential file to which data is written. The specified channel number must be between 0 and 99, inclusive. Use of the number sign (#) is mandatory. A runtime error results if this channel does not permit access with an I/O mode of OUTPUT or APPEND. If a channel number of 0 is specified, output is written to the standard file $OUTPUT.
writelist	Optional list of expressions. Also referred to as the write list.

Remarks

- The expressions in the write list are evaluated, and their values are written in sequence to the specified file. The form of the output fits the requirements of data that is to be read through the INPUT statement.

- If CHANREF and the subsequent comma are omitted, output is sent to the standard file $OUTPUT.

- If the file specified by CHANREF is connected to a terminal, the WRITE statement works as described in the Input and Output chapter. The WRITELIST parameter and the form of the output produced using the WRITE statement are discussed in depth in the Input and Output chapter.

Examples

If N receives the value 4 in the INPUT statement, this program fragment writes the output to the file EVALUATION.

```
DEFINT N,P,X
OPEN "EVALUATION" FOR OUTPUT AS #8
DEF POLY(X) = 3*(X+5)*(X-4)
INPUT N
WRITE #8, POLY(N-1),POLY(N),POLY(N+1)
```

The output appears below:

```
-24,0,30
```

Random I/O

This section discusses the BASIC statements and library functions that are used for Input/Output using random files.

FIELD Statement

Purpose	Defines the fields of a buffer, and establishes string variables that coincide with these fields. The string variables are used as vehicles for moving data between the buffer and a BASIC program, and between the buffer and a NOS/VE random file.
Format	FIELD chanref , fieldlist

chanref	Channel number reference specifying the channel whose buffer is to be divided into fields. The specified channel number must be between 1 and 99, inclusive. Use of the number sign (#) is optional. A runtime error results if this channel does not permit access with an I/O mode of RANDOM.
fieldlist	Field list defines field sizes and corresponding string identifiers for the buffer associated with the specified channel.

A field list has the form:

sizel AS svarl , size2 AS svar2 , ... , sizeN AS svarN

sizeJ	Numeric expression whose value, when rounded to the nearest integer, establishes the length (in bytes) of the Jth field of the buffer, where (1 <= J <= N).
svarJ	String identifier corresponding to the Jth field, where (1 <= J <= N). SVARJ cannot be a substring expressed by colon substring notation or a MID$ reference. The length of SVARJ is the rounded value of SIZEJ.

Remarks

- Bytes are allocated in the order specified in the field list. A runtime error results if the number of bytes allocated exceeds the length of the buffer. However, the entire buffer need not be used.

- If a field's string identfier later receives a value through the LET, SWAP, INPUT, LINE INPUT, or READ statements, or through its use as an actual parameter, then the variable stops designating a field. A preexisting string variable that is redefined through the FIELD statement loses its former value.

- Note that the FIELD statement sets up the mechanism for data to be moved, but does not actually cause any movement. If more than one FIELD statement is executed for a single channel, all field specifications remain in effect concurrently.

- NOS/VE BASIC integers and real numbers require eight bytes of storage. The use of an 8-byte field preserves the numeric sense of a field.

- NOS/VE BASIC strings are stored, one ASCII character per byte, in as many contiguous bytes as are required to hold their current values.

- For random files, record length and buffer length are always equal. This is not the case with sequential files since the WIDTH statement adjusts buffer length without changing record length.

Examples

These statements open a channel to the random file EMPLOYEE and define the fields of the associated 50-byte buffer. NAME$ coincides with the first 34 bytes of the buffer, YEARS$ with the next 8 bytes, and SALARY$ with the last 8 bytes.

```
OPEN "EMPLOYEE" AS #2 LEN=50
FIELD #2, 34 AS NAME$, 8 AS YEARS$, 8 AS SALARY$
```

GET Statement

Purpose	Reads a record from a random file into the buffer associated with a specified channel.

Format

```
GET  chanref , numrec
```

chanref	Channel number reference specifying the random file from which a record is to be read. The specified channel number must be between 1 and 99, inclusive. Use of the number sign (#) is optional. A runtime error results if this channel does not permit access with an I/O mode of RANDOM.
numrec	Optional numeric expression whose value, when rounded to the nearest integer, specifies the number of the record to be read.

Remarks

If NUMREC is omitted, the preceding comma is also omitted. The record following the one pointed to by the channel record pointer is read into the buffer. Hence, the accessed record has a record number of (1 + LOC(x)), where x is the specified channel number.

Examples

The first two statements open a channel to the random file EMPLOYEE and define fields in the associated 50-byte buffer. The GET statement reads the first record (50 bytes) of binary data from EMPLOYEE into the buffer.

```
OPEN "EMPLOYEE" AS #2 LEN=50
FIELD #2, 34 AS NAME$, 8 AS YEARS$, 8 AS SALARY$
GET #2
```

Numeric Interpretation of Strings

Purpose

Reads a numeric value into a buffer from a random file that can only be referenced as a string since it is now the value of a string variable (defined in a FIELD statement). For a BASIC program to access this value as a number rather than a string, the interpretation of the value of the string variable must be changed.

Format

```
CVI(strvar)
CVS(strvar)
CVD(strvar)
```

strvar	String variable whose value is the string to be interpreted as numeric. A runtime error results if this value does not have a length of 8 bytes. No other argument checking is performed.

Remarks

- The CVI, CVS, and CVD functions interpret the value of a string variable as numeric. These functions complement the MKI$, MKS$, and MKD$ functions, respectively.
- The CVI function interprets a string as an integer. The CVS and CVD functions interpret a string as a real number.
- Note that these functions change the interpretation of a value without changing its representation. The bytes in which the value is stored are not changed.

Examples The following shows an example using the CVI function.

```
    DEFINT I,N : DATA 50 : READ N
    OPEN "SECRET" AS #7 LEN=28
30  FIELD #7, 20 AS COUNTY$, 8 AS POPULATION$
    FOR I = 1 TO N
50      GET #7
60      LET POP% = CVI(POPULATION$)
        PRINT USING "####,### PEOPLE LIVE IN & COUNTY_.": POP%,COUNTY$
    NEXT I
    END
```

Line labeled 30:	FIELD statement defines two fields for the 28-byte buffer associated with channel #7. COUNTY$ coincides with the first 20 bytes of the buffer; POPULATION$ coincides with the last 8 bytes.
Line labeled 50:	The GET statement reads the next record from the file SECRET into the buffer.
Line labeled 60:	The CVI function interprets the value of POPULATION$ as an integer.

String Interpretation of Numerics

Purpose	Converts numeric values to string values.

Format

```
MKI$(number)
MKS$(number)
MKD$(number)
```

number	Numeric expression whose value is the number to be interpreted as a string.

Remarks

- The MKI$, MKS$ and MKD$ functions interpret a numeric value as a string variable. These functions complement the CVI, CVS, and CVD functions, respectively.
- For the program to treat this value as a string rather than a number, the interpretation of the value must be changes without changing its representation.
- The value returned is an internal representation of the value of NUMBER in an 8-byte string.
- The MKI$ function interprets the value of an integer as a string. The MKS$ and MKD$ functions interpret the value of a real number as a string.
- Note that these functions change the interpretation of a value without changing its representation. The bytes in which the value is stored are not changed.

LSET and RSET Statements

Purpose

Assigns the value of a string expression to a string variable without changing the length of the string variable. These statements can be used to move data from a program into a buffer in preparation for random output. They can also be used for assignment outside the context of files.

Format

```
LSET  svar = string
RSET  svar = string
```

svar — String identifier naming the variable being assigned a value. SVAR cannot be a substring expressed by colon substring notation or a MID$ reference.

string — String expression whose value contains the characters used in the assignment.

Remarks

- If the length of the value of STRING is:
  - Greater than the length of SVAR, the value of STRING is truncated on the right before the assignment is made.
  - Less than the length of SVAR, the LSET function left-justifies the value of STRING, providing enough trailing spaces to preserve the length of the SVAR.
  - Less than the length of SVAR, the RSET function right-justifies the value of STRING, providing enough leading spaces to preserve the length of the SVAR.
- If SVAR is a string variable used for the first time and not a field name, it has a value of the null string with a length of zero. No assignment will occur.

Examples

- No assignment takes place if this is the first use of NAME$ and it is not a field name.

  ```
  LSET NAME$ = "MARY"
  ```

  Now NAME$ has a length equal to 4.

  ```
  LET NAME$ = "JOSE"
  ```

  ```
  LSET NAME$ = "JOE "
  ```

  would be the same as

  ```
  LET NAME$ = "JOE "
  ```

- If the variable STRVAR corresponds to a field of a buffer used for random I/O, the value assigned is moved into the buffer. A numeric value to be moved into the buffer must be interpreted as a string using the MKI$, MKS$, or MKD$ function. The field to receive such a numeric value must be 8 bytes in length.

  ```
  OPEN "WAGES" AS #9 LEN=28
  FIELD #9, 20 AS MONTH$, 8 AS GROSS$
  RSET MONTH$ = "APRIL"
  LSET GROSS$ = MKI$(2587)
  ```

  The first two statements create a channel to the random file WAGES and establish fields in the associated 28-byte buffer. The RSET statement right-justifies the value "APRIL" in the field corresponding to MONTH$. The LSET statement inserts a string interpretation of the integer value 2587 in the field corresponding to GROSS$.

PUT Statement

Purpose

Writes the data in a buffer to the associated random file.

```
PUT  chanref , numrec
```

chanref	Channel number reference specifying the random file to which a record is to be written. The specified channel number must be between 1 and 99, inclusive. Use of the number sign (#) is optional. A runtime error results if this channel does not permit access with an I/O mode of RANDOM.
numrec	Optional numeric expression whose value, when rounded to the nearest integer, specifies the number of the record to be written. A runtime error results if the specified number is less than one, or if the relative location in the random file implied by the product of the record number and the record length exceeds the maximum file size for NOS/VE.

Remarks

If NUMREC is omitted, the preceding comma is also omitted. The data is written to the record following the one pointed to by the channel record pointer. Hence, the record written has a record number of (1 + LOC(x)), where x is the specified channel number.

Examples

The first two statements open a channel to the random file WAGES and define fields for the associated 28-byte buffer. The RSET and LSET statements move data into the buffer. The PUT statement writes the data in the buffer to the fifth record of file WAGES. Any data previously stored in record number 5 is destroyed.

```
OPEN "WAGES" AS #9 LEN=28
FIELD #9, 20 AS MONTH$, 8 AS GROSS$
RSET MONTH$ = "APRIL"
LSET GROSS$ = MKI$(2587)
PUT #9, 5
```

Compilation and Execution 14

Compilation and Execution 14

A NOS/VE BASIC program is processed in two stages: program compilation and program execution. This chapter describes the two-stage process.

Some knowledge of the NOS/VE System Command Language (SCL) is required to understand the material in this chapter. See the NOS/VE System Usage manual for a discussion of relevant terms and commands.

Compilation Overview

A NOS/VE BASIC program must be compiled before it can be executed.

Compiling a BASIC program means translating it from BASIC into machine language. This translation is performed by a processor known as a compiler.

The original BASIC program is referred to as the source program.

The compiled version of the program is referred to as the binary object program, or just the object program.

The time during which a program is being compiled is referred to as compile-time.

The BASIC compiler is called by entering the SCL BASIC command in response to the usual NOS/VE prompt (/).

When called, the compiler:

- Creates a binary object program.
- Produces a listing of the source program.
- Accumulates diagnostics for any errors found during compile-time.

NOS/VE BASIC compile-time errors are discussed in the Compile-Time Diagnostics appendix. For more information, see appendix C.

Once an error-free binary object program is created, you are ready for the program execution stage. The system executes the binary object program, not the source program.

Note that:

- A BASIC program need only be compiled once.
- An error-free binary object program can be executed any number of times.
- A single BASIC external routine in a program can be compiled by itself, but only the main program can initiate execution.

BASIC Compiler Command

Purpose

Calls the NOS/VE BASIC compiler with the SCL BASIC command, which can be entered at any NOS/VE prompt.

Format

```
BASIC
    INPUT=file
    BINARY=file
    LIST=file
    LIST_OPTIONS=list of keywords
    ARRAY_DIMENSIONS=keyword
    STATUS=status variable
```

Any file whose reference does not specify a catalog is assumed to be in the working catalog. Remember that $LOCAL is the default working catalog.

Parameters

INPUT or I

Specifies the file containing the source program to be compiled. The default input file is $INPUT.

When the BASIC command is invoked using the default INPUT parameter, input is expected from the standard file $INPUT. To terminate input and compile what is entered, enter the END_OF_INFORMATION value after a prompt.

The END_OF_INFORMATION value is a connection attribute. The default value is *EOI. To display the connection attribute value, enter the following SCL command:

```
display_term_conn_default, end_of_information
```

For more information, see the NOS/VE System Usage manual.

BINARY or B

Specifies the file to receive the binary object program. The default binary object file is $LOCAL.LGO.

LIST or L

Specifies the file to receive the compiler output listing. The default list file is $LIST.

Parameters (cont)	LIST_OPTIONS or LO

Specifies the information that is to appear in the compiler output listing. The list options are:

S	Source program listing.
O	Object program listing.
R	Cross-reference listing.
N	None.

The default list option is S.

ARRAY_DIMENSIONS or AD

Establishes either static or dynamic dimensioning for all arrays in the compiled program. Options are:

STATIC (S)	Fixes array dimensioning at compile-time.
DYNAMIC (D)	Allows redimensioning of any array at runtime.

The default array dimensioning is STATIC.

STATUS

Specifies an SCL status variable to receive the compiler-generated error status code. The default is no status variable.

Listing Options

The LIST_OPTIONS parameter is specified as a list of letters separated by commas in parentheses. If only one letter is specified, the parentheses can be omitted. Letters can appear more than once and in any order.

The letters S, O, and R represent switches that control the contents of the list file. The switch S is initially set. This means that the source listing is provided by default. The other switches are initially clear. This means that the object code and cross-reference listings are not provided unless explicitly requested.

The compiler scans the LIST_OPTIONS list from left to right.

When it finds:

- A letter other than N, the corresponding switch is set.
- The letter N, all switches are cleared.

The state of the switches when the end of the list is reached determines the contents of the list file.

Note that compile-time diagnostics are always written to the list file, even if LO=N is specified.

STATUS Variable

The STATUS parameter is the means by which the compiler makes the outcome of the compilation available to the surrounding SCL environment. The status is either normal, or one of three status condition codes is returned:

- 1, indicating warning errors.
- 2, indicating fatal errors.
- 3, indicating catastrophic errors.

If the source program contains errors of more than one severity, the status of the most severe error is returned.

Remember that in SCL, a series of two or more periods at the end of a line indicates that a command is to be continued on the next line.

Sample Compiler Calls

Examples

- This command specifies all default parameters.

```
BASIC
```

- This command specifies the following parameters:

```
BASIC  I=BASPROG  B=BIN  L=COMPLIST  LO=(N,O)
```

I=BASPROG	Reads the source program from the file BASPROG in the working catalog.
B=BIN	Writes the binary object program to the file BIN in the working catalog.
L=COMPLIST	Writes the compiler output listing to the file COMPLIST in the working catalog.
LO=(N,O)	Provides the binary object program listing but suppresses the source listing.

No status variable is specified.

- This command specifies the following parameters:

```
BASIC  INPUT=$USER.YEAR  LIST=RESULTS ..
LIST_OPTIONS=R  STATUS=STATE
```

INPUT=$USER.YEAR	Reads the source program from the file YEAR in the $USER catalog.
LIST=RESULTS	Writes the compiler output listing to the file RESULTS in the working catalog.
LIST_OPTIONS=R	Provides the source and the cross-reference listings.
STATUS=STATE	Specifies the status variable STATE.

By default, the binary object file is $LOCAL.LGO.

Program Execution

The time during which a program is executing is referred to as runtime. The execution stage of program processing really involves both loading and running the program.

Loading is performed by a processor known as a loader. Among other actions, the loader:

- Establishes links between external routines in a program.
- Establishes links between the program and system routines.
- Fetches library routines called by the program.
- Brings the binary object code into main memory for execution.
- Sends a summary diagnostic message to the standard file $ERRORS, which is connected to OUTPUT. The errors and loadmap are also sent to a file if you specify the LOAD_MAP parameter on either the EXECUTE_TASK or SET_PROGRAM_ATTRIBUTES commands.

EXECUTE_TASK

Purpose Executes a BASIC program by entering this SCL command.

Format EXECUTE_TASK FILE=file reference DEBUG_MODE=boolean

FILE (F)	Specifies the binary object file for the program (established by the BINARY_OBJECT parameter in the the BASIC compiler command).
DEBUG_MODE (DM)	Invokes the Debug utility which lets you debug your BASIC program. Values you can specify are:

ON	Invokes the Debug utility. You then execute your BASIC program from within the Debug utility.
OFF	Does not invoke the Debug utility. If you specify OFF, you cannot debug the BASIC program during this execution.

If DEBUG_MODE is omitted, OFF is assumed.

Remarks

- If the binary object file is in the $LOCAL catalog, you can execute your program by simply typing the file name in response to the system prompt. For example, if the binary object file is named $LOCAL.LGO, you can execute the file by typing LGO ("Load and Go"). If the binary object file is named $LOCAL.MY_BINARY, you can execute the file by typing MY_BINARY. When the binary object file is in the $LOCAL catalog, typing the file name executes that file even if the working catalog is not $LOCAL.

- If the binary object file is in a catalog other than $LOCAL, you must use the EXECUTE_TASK command to execute your program. For example, if the binary object file is in the file $USER.LGO and the working catalog is $LOCAL, execute your program by typing EXECUTE_TASK $USER.LGO. If the binary object file is in the file $USER.LGO and the working catalog is $USER, execute your program by typing EXECUTE_TASK LGO.

- The Debug utility is described in appendix E, Introduction to Debug.

- For detailed information about the EXECUTE_TASK command, see the NOS/VE Object Code Management Usage manual.

Examples

- These two commands are equivalent. Each one executes the binary object program that is stored in the file BIN in the $USER catalog.

  ```
  EXECUTE_TASK  FILE=$USER.BIN
  EXET  F=$USER.BIN
  ```

- This specification executes the binary object program stored in the file COMPILED_PROGRAM in the $LOCAL catalog.

  ```
  COMPILED_PROGRAM
  ```

- The following command invokes the Debug utility with the BASIC program $USER.BIN:

  ```
  EXECUTE_TASK FILE=$USER.BIN DEBUG_MODE=ON
  ```

  When you enter the preceding command, the Debug utility is invoked. You then use Debug commands to execute the BASIC program $USER.BIN.

In the following terminal session, the EDIT_FILE (EDIF) and INSERT_LINES (INSL) commands are used to create a one-line program. The program is placed in the file named SAMPLE in the $LOCAL catalog. This program is then compiled and executed. The program output appears on the terminal.

```
/edif  f=sample
Begin editing deck SAMPLE
ef/insl
Enter Text
?PRINT "THIS IS A SHORT PROGRAM."
?**
ef/quit
/basic  i=sample  l=list
/lgo
THIS IS A SHORT PROGRAM.
/
```

If any runtime errors occur under default handling, diagnostics are written to the standard error file $ERRORS. The default connection for the standard error file is the NOS/VE listing file OUTPUT. For interactive mode, this means that diagnostics appear at the terminal.

Runtime errors can be handled from within a program. For more information, see chapter 6.

To execute separately compiled external routines, they must be combined to form an object library.

RESEQUENCE Utility

Purpose Resequences a BASIC source program and automatically updates all line references.

Format

```
RESEQUENCE or RES
    INPUT=file reference
    OUTPUT=file reference
    NEW_BASE=integer
    NEW_INCREMENT=integer
    ERROR=file reference
    STATUS=status variable
```

Parameters INPUT or I

File containing the BASIC source text to be resequenced.

An error message is issued for any of the following:

- File is not found
- File is empty
- File is not available for read access
- File attributes do not allow sequential access

The INPUT parameter is required.

OUTPUT or O

File to which RESEQUENCE writes its results.

An error message is issued if the OUTPUT file is not available for write access or if the file attributes do not allow it to be written as a sequential file.

If OUTPUT is omitted, RESEQUENCE will rewrite its results to the same file from which it is read.

The record length of the output file must be less than the length specified on the MAXIMUM_RECORD_LENGTH attribute for that file. This condition must be true regardless of the type specified on the RECORD_TYPE attribute for the file.

Parameters (cont)

NEW_BASE or NB

Integer value that specifies the first label of the resequenced BASIC source text. Integer value must be within the range 1 to 999,999, inclusive.

An error message is issued if the integer value is outside the allowed range.

If NEW_BASE is omitted, the value 100 is used.

NEW_INCREMENT or NI

Integer value that specifies the increments to be used between successive labels in the resequenced source text. Integer value must be within the range 1 to 100,000.

An error message is issued if the integer value is outside the allowed range.

If NEW_INCREMENT is omitted, the value 10 is used.

ERROR or E

File to which RESEQUENCE writes diagnostic messages if it encounters problems. If no problems are encountered, this file is not opened.

If ERROR is omitted, the standard file $ERRORS is used.

STATUS

SCL status variable in which the termination status of RESEQUENCE is returned. If specified, the SCL command interpreter proceeds to the next command even if an abnormal condition is encountered.

If STATUS is omitted, any error that is detected by RESEQUENCE is reported to the user's environment and the SCL command interpreter skips succeeding commands in the current block.

Remarks

- A program compiled from resequenced source text can behave differently than previous compiled versions. Differences depend on how the ERL function is used.

  The ERL function returns the value of the label associated with the line having the most recent runtime error. The value of ERL may change after the program is resequenced. RESEQUENCE checks for the appearance of ERL in relational expressions. If ERL is compared to a simple integer constant that has the same value as a label in the routine, the constant is presumed to be a label reference and is changed.

- There are three types of problems that can be detected by RESEQUENCE. The first problem could be with the parameters or execution environment. The second type of problem is with the label definitions in the input file. The third problem is writing the newly labeled text to the output file.

- If the first problem is found, a diagnostic message is written to the error file and RESEQUENCE is terminated. Examples of such problems are parameter specification errors, specification of an empty input file, or specification of an output file that is not available with sufficient access to receive the resequenced text.

- After RESEQUENCE is satisfied with the parameters, it can still detect other errors. The types of problems that are detected during this phase will not terminate the program. In order to deliver as much information as possible, RESEQUENCE accumulates diagnostic messages in the error file and continues examining as much of the input text as possible.

- BASIC external routines are read from the input file one at a time. All the labels of an external routine are examined. For each label that is out of range or out of order, a diagnostic message is added to the errors file. If a problem is found with the label definitions in a routine, the references of that routine are not examined.

Remarks (cont)

- If no problems are found with the label definitions in a routine, RESEQUENCE checks all label references in the routine. For each reference to an undefined label, a diagnostic message is added to the errors file.

- If errors are found either with label definitions or label references, the results of RESEQUENCES´s effort are not written to the output file. After RESEQUENCE has examined all BASIC external routines in the input file, it terminates with a status indicating the severity of the worst problem it encountered.

Examples

- The BASIC source file RES_TEST contains the following:

```
42     REM  This is a RESEQUENCE example.
55     DEFINT i - n
56     INPUT "Specify index: ", i
60     ON i GOTO 73, 71, 72
65     END
71     PRINT "I =";  i   :  GOTO 65
72     PRINT "I =";  i   :  GOTO 65
73     PRINT "I =";  i   :  GOTO 65
```

  The following RESEQUENCE command is executed:

```
/resequence, res_test, res_result
```

  The RES_RESULT file contains the following:

```
100    REM  This is a RESEQUENCE example.
110    DEFINT i - n
120    INPUT "Specify index: ", i
130    ON i GOTO 170, 150, 160
140    END
150    PRINT "I =";  i   :  GOTO 140
160    PRINT "I =";  i   :  GOTO 140
170    PRINT "I =";  i   :  GOTO 140
```

- If the following RESEQUENCE command is executed:

```
/resequence, res_test, res_result, 1000, 100
```

  The RES_RESULT file contains the following:

```
1000   REM  This is a RESEQUENCE example.
1100   DEFINT i - n
1200   INPUT "Specify index: ", i
1300   ON i GOTO 1700, 1500, 1600
1400   END
1500   PRINT "I =";  i   :  GOTO 1400
1600   PRINT "I =";  i   :  GOTO 1400
1700   PRINT "I =";  i   :  GOTO 1400
```

Appendixes

Glossary A

The terms in this glossary appear in alphabetical order.

A

An array whose elements are passed to a NOS/VE BASIC routine when the routine is called. An actual array is denoted by an array name followed by parentheses that contain zero or more commas.

Actual Parameter

A variable or array whose value is passed to a procedure when the procedure is called.

Array

A collection of memory locations that are referenced by a single name and store logically related values of the same data type.

Array Element

A variable denoting one memory location in an array. An array element is referenced using the array name and a sequence of numbers called subscripts. The subscripts identify the memory location by its position within the array.

ASCII

An acronym for American Standard Code for Information Interchange. Under this coding system, each character in a prescribed set is given a 7-bit code. NOS/VE stores each 7-bit ASCII code right-justified in an 8-bit byte, with the first bit set to zero.

B

BASIC

An elementary programming language whose name is an acronym for Beginner´s All-purpose Symbolic Instruction Code.

Batch Mode

An execution mode where a job is submitted and processed as a unit with no intervention from the user. Contrast with interactive mode.

Beginning-of-information (BOI)

The point at which data begins in a file.

Binary Data

Data that is stored using the computer´s internal binary representation. Binary data cannot be printed in readable form. Contrast with Coded Data.

Binary Object Program

An executable machine language program that is produced from a source program by a compiler.

Bit

A binary digit, either 0 or 1. A bit is the smallest unit of storage in a computer. See also Byte.

Block

A group of logically or physically related statements or lines.

Block Function

A multi-statement user-defined function whose function body is a block. A block function is a NOS/VE BASIC routine. Contrast with Expression Function.

BOI

See Beginning-of-information.

Byte

A group of consecutive bits constituting a storage unit in the computer. A NOS/VE byte is 8 bits long and can hold the ASCII code for a single character.

C

Call-By-Address

A parameter passing style where the address of an actual parameter is passed to the corresponding formal parameter. Contrast with Call-By-Value.

Call-By-Reference

See Call-By-Address.

Call-By-Value

A parameter passing style where the value of an actual parameter is stored in a temporary memory location. The address of this temporary memory location, rather than the address of the actual parameter, is passed to the corresponding formal parameter. Contrast with Call-By-Address.

Catalog

1. A directory of files and catalogs maintained by the system for a user. The catalog $LOCAL contains all temporary file entries.

2. The part of a path that identifies a particular catalog in a catalog hierarchy. The format is as follows:

```
name.name. ... .name
```

   Each name is a catalog.

See also Catalog Name and File Path.

Catalog Name

The name of a catalog. The catalog name is used in a file path. By convention, the name of the user's master catalog is the same as the user's user name.

Channel

A path for transmitting data between a BASIC program and a NOS/VE file.

Character

A letter, digit, space, or other symbol that is represented by a code in one or more of the standard character sets. NOS/VE supports the American National Standards Institute (ANSI) standard ASCII character set (ANSI X3.4-1977).

Coded Data

Data that is stored as a sequence of ASCII codes. Coded data can be printed in readable form. Contrast with Binary Data.

Comment

A sequence of characters that is ignored by the compiler, and is used for program documentation.

Compile

To translate a program written in a high-level language into machine language program that can be loaded and executed.

Compiler

A processor that translates code from a high-level programming language into machine language. That is, a processor that translates a source program into a binary object program.

Compiler Sequence Number

See Line Number.

Compile-time

The time during which a program is being compiled.

Constant

A value that must remained fixed during program execution.

Cycle

A numbered version of a permanent file. All cycles of a file share the same file entry in a catalog. The file cycle is specified in a file reference by its cycle number or by a special indicator, such as $NEXT.

See also Cycle Number and Cycle Reference.

Cycle Number

An unsigned integer from 1 through 9999 that identifies a specific version of a permanent file.

See also Cycle and Cycle Reference.

Cycle Reference

The cycle of a permanent file to be accessed. A cycle reference can be either an unsigned integer from 1 through 9999 or one of the following designators:

$HIGH
$LOW
$NEXT

See also Cycle and Cycle Number.

D

Debug

The NOS/VE command utility for tracing and correcting program errors.

Default

The process by which a value, parameter, attribute, or option is assigned by the program or the system when the item is not specified by the user.

Diagnostic

An error message.

Dynamic Dimensioning

An array mode established at compile time that allows changing array dimensions during program execution.

E

End-of-information (EOI)

The point at which data in a file ends.

EOI

See End-of-information.

Exception

A runtime error.

Execution-time

See Runtime.

Expression

One or more constants, variables, or function references that are linked by operators. Subexpressions occurring within an expression can be enclosed within parentheses.

Expression Function

A single-statement user-defined function whose function body consists of a single expression. An expression is not a NOS/VE BASIC routine. Contrast with Block Function.

External Routine

A NOS/VE BASIC main program or subprogram. An external routine can be compiled by itself.

F

Family

A logical grouping of NOS/VE users that determines the location of their permanent files. A family can be subdivided into accounts and projects.

Family Path

Identifies a file via a family name and a user path using one of the following formats:

```
:family.user_path

$FAMILY.user_path
```

Field

1. A named subdivision of a record in a random file.

2. A section of the print line that is designated for specially formatted output.

File

A named collection of data, often organized into logically related groups called records. A file is identified by specifying a path and, optionally, a cycle reference (for permanent files) as follows:

```
path.cycle
```

File Access Method

The method by which records can be read from or written to a file. See also Random Access and Sequential Access.

File Access Mode

A NOS/VE file attribute that determines the I/O operations that can be performed on a file. Possible file access modes include read, write, and execute.

File Attribute

A characteristic of a file. Each file has a set of attributes that define the file structure and processing limitations.

File Name

The name of a NOS/VE file. The name is used in a file reference to identify the file.

See also File Reference and Name.

File Organization

The NOS/VE file attribute that determines which file access method can be used with a file. For example, SEQUENTIAL file organization permits sequential access, while BYTE ADDRESSABLE file organization permits random access.

File Path

Identifies a file. A path can include the family name, user name, subcatalog name or names, and file name.

File Position

The location in a file at which the next read or write operation begins. A file that can be positioned is identified by specifying a path, an optional cycle reference (for permanent files), and an optional file position as follows:

```
path.cycle.file_position
```

The file position designators are:

$ASIS	Leave the file in its current position.
$BOI	Position the file at the beginning-of-information.
$EOI	Position the file at the end-of-information.

File Reference

An SCL element that identifies a file and, optionally, the file position to establish prior to using the file. The format of a file reference is:

```
:family.user_name.catalog.file_name.cycle.file_position
```

See also Catalog, Cycle, Family, File, File Position, and User Name.

Formal Array

An array in a NOS/VE BASIC routine that acts as a placeholder for an actual array. A formal array is denoted by an array name followed by parentheses that contain zero or more commas.

Formal Parameter

A variable or array in a procedure that acts as a placeholder for an actual parameter.

Function

A procedure that returns a value to the place in an expression where the procedure was called. See also Block Function and Expression Function.

I

Identifier

A name that labels a program component or specifies some action or attribute within a program. A NOS/VE BASIC identifier can have at most 31 characters, and consists of a letter followed by a series of letters, digits, and periods, optionally followed by a type specification symbol.

Interactive Mode

A mode of execution where the user enters commands or data at the terminal during program execution.

Internal Routine

A NOS/VE BASIC block function or subroutine that is embedded within a main program or subprogram. An internal routine cannot be compiled by itself.

K

Keyword

An identifier that has a preassigned meaning when it is used in a specific context.

L

Label

A positive integer of at most six digits that is supplied by the programmer. A label can be used to reference a line during program execution. Contrast with Line Number.

Library Function

A system-supplied function.

Line Number

A number assigned to a program line by the compiler to denote the physical position of the line within a program. A line number cannot be used to reference a line during program execution. Contrast with Label.

Local File

A temporary NOS/VE file. The $LOCAL catalog contains temporary files. Temporary files are preserved only throughout your NOS/VE session; they are discarded when you log out of NOS/VE.

See also File, File Path, and Local Path.

Local Path

Identifies a local file as follows:

```
$LOCAL.file_name
```

M

Main Program

The only NOS/VE BASIC routine in a program that can be executed by itself. A main program is an external routine that is not a subprogram.

Master Catalog

The catalog NOS/VE maintains for each new user name. The master catalog contains entries for all permanent files and catalogs a user creates. By convention, the name of the master catalog is the same as the user name.

N

Name, NOS/VE

A combination of from 1 through 31 characters chosen from the following set:

Alphabetic characters (A through Z and a through z).
Digits (0 through 9).
Special characters (# @ $ _ [ ]   ^ ` { } | ~).

The first character of a NOS/VE name cannot be a digit.

Null String

A string constant or string variable that has a length of zero.

O

Object Program

See Binary Object Program.

P

Parameter Passing

The manner in which an actual parameter is passed to the corresponding formal parameter when a procedure is called. See also Call-By-Addresss and Call-By-Value.

Permanent Catalog

A catalog of permanent files.

Permanent File

A file preserved by NOS/VE across job executions and system deadstarts. A permanent file has an entry in a permanent catalog.

See also File and Permanent Catalog.

Plain Name

A NOS/VE BASIC identifier that does not contain a type specification symbol (%, !, #, or $) as its last character.

R

Random Access

A file access method in which records can be read or written in any order. Contrast with Sequential Access.

Random File

A file that can be accessed randomly.

Record

The smallest subdivision of a file that can be processed by a single I/O request.

Relative Path

Identifies a file via defaults established with the current working catalog or an absolute path. A relative path is used in a family path, user path, and local path. NOS/VE supplies any omitted values necessary to create an absolute path. For example, the following file reference identifies a NOS/VE file named PROG1_INPUT:

```
:v01.joe_user.basic.prog1_input
```

If you are logged in to the user name JOE_USER and the working catalog is set to $USER, you can identify the file named PROG1_INPUT with the following relative path:

```
basic.prog1_input
```

Reserved Word

A keyword that is reserved exclusively for program or system use and cannot be used by the programmer for his own purpose.

Routine

A NOS/VE BASIC main program, block function, or subroutine.

Runtime

The time during which a program is being executed.

S

Sequential Access

A file access method in which records must be read or written in the order of their physical location within a file. Contrast with Random Access.

Sequential File

A file that can be accessed sequentially.

Source Program

A program written in a high-level language such as BASIC or FORTRAN.

Standard File

A file that provides a default file for use by job files and other files. The standard files are identified by the following names:

$ECHO
$ERRORS
$INPUT
$LIST
$OUTPUT
$RESPONSE

When running a BASIC program in batch mode, messages generated by PRINT statements are written to the standard file $OUTPUT, which is connected to the real file OUTPUT; this OUTPUT file automatically prints when the job completes.

Static Dimensioning

An array mode that fixes array dimensions at compile time to provide greater program efficiency.

Status Condition Code

The 4-digit code that uniquely identifies a NOS/VE runtime diagnostic. The first two digits of a NOS/VE BASIC status condition code are 54.

Subprogram

A NOS/VE BASIC routine that can be compiled by itself, but cannot be executed by itself. A subprogram is either an external block function or an external subroutine. That is, a subprogram is an external routine that is not a main program.

Subroutine

A procedure that performs specific tasks for a calling procedure.

Subscripted Variable

See Array Element.

Substring

A string variable consisting of zero or more consecutive character positions within a given string variable. A NOS/VE BASIC substring is expressed using either colon-substring notation or a MID$ reference.

System Command Language (SCL)

The block-structured interpretive language that provides the interface to the features and capabilities of NOS/VE. All commands and statements are interpreted by SCL before being processed by the system.

T

Temporary File

A file in the NOS/VE $LOCAL catalog that disappears when the user logs off.

U

User Name

A name that identifies a NOS/VE user and the location of the user's permanent files in the user's family.

User Path

Identifies a file or catalog via a user name and, optionally, a relative path using one of the following formats:

```
.user_name.relative_path

$USER.relative_path
```

See also Relative Path and User Name.

V

Variable

A named memory location that is allowed to store different values at different times during program execution.

W

Working Catalog

The catalog used if no other catalog is specified on a file reference. The $LOCAL catalog is the default working catalog. You can change the working catalog by using the SET_WORKING_CATALOG command.

ASCII Character Set B

This appendix lists the ASCII character set (table B-1) used by the NOS/VE system.

NOS/VE supports the American National Standards Institute (ANSI) standard ASCII character set (ANSI X3.4-1977). NOS/VE stores the 7-bit ASCII code for each character right-justified in an 8-bit byte, with the first bit set to zero.

Table B-1. ASCII Character Set

ASCII Code				
Decimal Code	Hexadecimal Code	Octal Code	Graphic or Mnemonic	Name or meaning
000	00	000	NUL	Null
001	01	001	SOH	Start of Heading
002	02	002	STX	Start of Text
003	03	003	ETX	End of Text
004	04	004	EOT	End of Transmission
005	05	005	ENQ	Enquiry
006	06	006	ACK	Acknowledge
007	07	007	BEL	Bell
008	08	010	BS	Backspace
009	09	011	HT	Horizontal Tabulation
010	0A	012	LF	Line Feed
011	0B	013	VT	Vertical Tabulation
012	0C	014	FF	Form Feed
013	0D	015	CR	Carriage Return
014	0E	016	SO	Shift Out
015	0F	017	SI	Shift In
016	10	020	DLE	Data Link Escape
017	11	021	DC1	Device Control 1
018	12	022	DC2	Device Control 2
019	13	023	DC3	Device Control 3
020	14	024	DC4	Device Control 4
021	15	025	NAK	Negative Acknowledge
022	16	025	SYN	Synchronous Idle
023	17	027	ETB	End of Tran. Block
024	18	030	CAN	Cancel
025	19	031	EM	End of Medium
026	1A	032	SUB	Substitute
027	1B	033	ESC	Escape
028	1C	034	FS	File Separator
029	1D	035	GS	Group Separator
030	1E	036	RS	Record Separator

(Continued)

Table B-1. ASCII Character Set (Continued)

ASCII Code				
Decimal Code	Hexadecimal Code	Octal Code	Graphic or Mnemonic	Name or meaning
031	1F	037	US	Unit Separator
032	20	040	SP	Space
033	21	041	!	Exclamation Point
034	22	042	"	Quotation Marks
035	23	043	#	Number Sign
036	24	044	$	Dollar Sign
037	25	045	%	Percent Sign
038	26	046	&	Ampersand
039	27	047	'	Apostrophe
040	28	050	(	Opening Parenthesis
041	29	051	)	Closing Parenthesis
042	2A	052	*	Asterisk
043	2B	053	+	Plus
044	2C	054	,	Comma
045	2D	055	-	Hyphen
046	2E	056	.	Period
047	2F	057	/	Slant
048	30	060	0	Zero
049	31	061	1	One
050	32	062	2	Two
051	33	063	3	Three
052	34	064	4	Four
053	35	065	5	Five
054	36	066	6	Six
055	37	067	7	Seven
056	38	070	8	Eight
057	39	071	9	Nine
058	3A	072	:	Colon
059	3B	073	;	Semicolon
060	3C	074	<	Less Than
061	3D	075	=	Equal To
062	3E	076	>	Greater Than
063	3F	077	?	Question Mark
064	40	100	@	Commercial At
065	41	101	A	Uppercase A
066	42	102	B	Uppercase B
067	43	103	C	Uppercase C
068	44	104	D	Uppercase D
069	45	105	E	Uppercase E

(Continued)

Table B-1. ASCII Character Set (Continued)

ASCII Code				
Decimal Code	Hexadecimal Code	Octal Code	Graphic or Mnemonic	Name or meaning
070	46	106	F	Uppercase F
071	47	107	G	Uppercase G
072	48	110	H	Uppercase H
073	49	111	I	Uppercase I
074	4A	112	J	Uppercase J
075	4B	113	K	Uppercase K
076	4C	114	L	Uppercase L
077	4D	115	M	Uppercase M
078	4E	116	N	Uppercase N
079	4F	117	O	Uppercase O
080	50	120	P	Uppercase P
081	51	121	Q	Uppercase Q
082	52	122	R	Uppercase R
083	53	123	S	Uppercase S
084	54	124	T	Uppercase T
085	55	125	U	Uppercase U
086	56	126	V	Uppercase V
087	57	127	W	Uppercase W
088	58	130	X	Uppercase X
089	59	131	Y	Uppercase Y
090	5A	132	Z	Uppercase Z
091	5B	133	[	Opening Bracket
092	5C	134	\	Reverse Slant
093	5D	135	]	Closing Bracket
094	5E	136	^	Circumflex
095	5F	137	_	Underline
096	60	140	`	Grave Accent
097	61	141	a	Lowercase a
098	62	142	b	Lowercase b
099	63	143	c	Lowercase c
100	64	144	d	Lowercase d
101	65	145	e	Lowercase e
102	66	146	f	Lowercase f
103	67	147	g	Lowercase g
104	68	150	h	Lowercase h
105	69	151	i	Lowercase i
106	6A	152	j	Lowercase j
107	6B	153	k	Lowercase k
108	6C	154	l	Lowercase l
109	6D	155	m	Lowercase m
110	6E	156	n	Lowercase n

(Continued)

Table B-1. ASCII Character Set (Continued)

ASCII Code				
Decimal Code	Hexadecimal Code	Octal Code	Graphic or Mnemonic	Name or meaning
111	6F	157	o	Lowercase o
112	70	160	p	Lowercase p
113	71	161	q	Lowercase q
114	72	162	r	Lowercase r
115	73	163	s	Lowercase s
116	74	164	t	Lowercase t
117	75	165	u	Lowercase u
118	76	166	v	Lowercase v
119	77	167	w	Lowercase w
120	78	170	x	Lowercase x
121	79	171	y	Lowercase y
122	7A	172	z	Lowercase z
123	7B	173	{	Opening Brace
124	7C	174	|	Vertical Line
125	7D	175	}	Closing Brace
126	7E	176	~	Tilde
127	7F	177	DEL	Delete

Compile-time Diagnostics C

Compile-time is the time during which a program is being compiled. A compile-time error is a violation of the rules governing the structure and arrangement (the syntax) of BASIC statements. A diagnostic (error message) is issued by the compiler when such an error occurs. This diagnostic contains information to help you find the cause of the error.

This appendix discusses the format of NOS/VE BASIC compile-time diagnostics. It also lists and describes the BASIC compile-time diagnostics. The diagnostics are listed in numerical order.

When a program is compiled, the compiler generates a line number (compiler sequence number) for each program line. The first line is denoted line number 1, the second line is denoted line number 2, and so forth.

Line numbers are used in compile-time diagnostics to specify error location. Line numbers have nothing to do with BASIC labels, which you provide to reference lines from within your program.

NOS/VE BASIC runtime error diagnostics are listed in the Diagnostic Messages for NOS/VE Usage manual, publication number 60464613. The Diagnostic Messages manual is also available online. To read the online manual, log in to NOS/VE and type the following EXPLAIN command:

```
    explain manual=messages
```

The online manual named MESSAGES is then displayed.

You can also read the online manual when you receive a diagnostic message. When a diagnostic is displayed, type HELP. The Diagnostic Messages online manual is then displayed; the screen displayed describes the diagnostic message you received.

(BC 10) Unable to open LIST file {file}.

Severity Level: Fatal

Description: The list file specified by the LIST parameter of the BASIC command is not available for write access.

User Action: Check that the correct file was specified. This can be done with this command:
ATTACH_FILE,file,ACCESS_MODES=WRITE. Attach the list file requesting write access (modify, shorten, and append). For more on access modes, see the NOS/VE System Usage manual.

(BC 20) Unable to open INPUT file {file}.

Severity Level: Fatal

Description: The source text file specified by the INPUT parameter of the BASIC command is not available for read access.

User Action: Check that the correct file was specified. Attach the source text file requesting read access.

(BC 21) INPUT file specified for BASIC is empty.

Severity Level: Fatal

Description: The source file specified by the INPUT (I) parameter of the BASIC command contains no data.

User Action: Check that the correct file was specified and that the file position $EOI is not specified.

(BC 22) This source line exceeds {integer} characters; it has been truncated.

Severity Level: Fatal

Description: The number of characters in a source line exceeds the NOS/VE BASIC maximum line length of 255. As a result, the line has been truncated.

User Action: Split the source line into two or more valid lines.

(BC 23) Unprintable character, ASCII code {integer} (decimal), encountered.

Severity Level: Fatal

Description: Characters with ASCII decimal codes in the range 0-31 function as control characters and are unprintable. The message gives the decimal code of the unprintable character code.

User Action: Retype the source line.

(BC 26) System failure reading INPUT: {file}.

Severity Level: Catastrophic

Description: You have uncovered a problem with the NOS/VE BASIC compiler.

User Action: Follow the procedure established at your site for reporting software problems.

(BC 30) Unable to open the BINARY file {file}.

Severity Level: Fatal

Description: The binary object file {file} specified by the BINARY OBJECT parameter of the BASIC command is not available for modify access.

User Action: Check that the correct file was specified. This can be done with this command: ATTACH_FILE,file,ACCESS_MODES=WRITE. Attach the binary object file {file} requested by write access. For more information on access modes, see the NOS/VE System Usage manual.

(BC 31) BINARY OBJECT file {file} FILE_CONTENTS must be OBJECT or UNKNOWN.

Severity Level: Fatal

Description: The FILE_CONTENTS attribute of the file specified by the BINARY_OBJECT parameter is not OBJECT or UNKNOWN.

User Action: Check that the correct file was specified. To change the FILE_CONTENTS value, use a SET_FILE_ATTRIBUTES command specifying FILE_CONTENTS=DATA or FILE_CONTENTS=UNKNOWN.

Further Information: To display information about a file, use the DISPLAY_FILE_ATTRIBUTES command described in the NOS/VE System Usage and NOS/VE Commands and Functions manuals.

(BC 32) BINARY file {file}'s FILE_ORGANIZATION must be SEQUENTIAL or BYTE_ADDRESSABLE.

Severity Level: Fatal

Description: The FILE_ORGANIZATION attribute of the file specified by the BINARY_OBJECT parameter is not SEQUENTIAL or BYTE_ADDRESSABLE.

User Action: Check that the correct file was specified. To change the FILE_ORGANIZATION value, use a SET_FILE_ATTRIBUTES command specifying FILE_ORGANIZATION=SEQUENTIAL or FILE_ORGANIZATION=BYTE_ADDRESSABLE.

Further Information: To display information about a file, use the DISPLAY_FILE_ATTRIBUTES command described in the NOS/VE System Usage and NOS/VE Commands and Functions manuals.

(BC 33) BINARY file {file}'s FILE STRUCTURE must be DATA or UNKNOWN.

Severity Level: Fatal

Description: The FILE_STRUCTURE attribute of the file specified by the BINARY_OBJECT parameter is not DATA or UNKNOWN.

User Action: Check that the correct file was specified. To change the FILE_STRUCTURE value, use a SET_FILE_ATTRIBUTES command specifying FILE_STRUCTURE=DATA or FILE_STRUCTURE=UNKNOWN.

Further Information: To display information about a file, use the DISPLAY_FILE_ATTRIBUTES command described in the NOS/VE System Usage and NOS/VE Commands and Functions manuals.

(BC 34) The BINARY file {file} must have MODIFY access.

Severity Level: Fatal

Description: The job does not have modify access to the file specified by the BINARY_OBJECT parameter.

User Action: Check that the correct file was specified. To change the access mode, attach the file specifying modify access.

Further Information: To display information about a file, use the DISPLAY_FILE_ATTRIBUTES command described in the NOS/VE System Usage and NOS/VE Commands and Functions manuals.

(BC 50) Magnitude of integer constant is too large.

Severity Level: Warning

Description: The magnitude of an integer constant exceeds the maximum integer size of
(2^63 - 1), which is approximately (9.2 * 10^18).

User Action: Replace the invalid integer constant with a valid one.

(BC 51) Magnitude of real constant is too large.

Severity Level: Fatal

Description: The magnitude of a real constant exceeds the maximum real number size of (2^4095), which is approximately (5.2 * 10^1232).

User Action: Replace the invalid real constant with a valid one.

(BC 53) Malformed numeric constant.

Severity Level: Fatal

Description: A numeric constant is improperly formed. Most likely, a digit or decimal point in a numeric constant is not followed by a digit or separator.

User Action: Make the required correction.

(BC 60) Magnitude of hexadecimal constant is too large.

Severity Level: Fatal

Description: The magnitude of a hexadecimal integer constant exceeds the maximum integer size of (2^63 - 1). In BASIC hexadecimal form, this magnitude is written as &H7FFFFFFFFFFFFFFF.

User Action: Replace the invalid hexadecimal constant with a valid one.

(BC 61) Empty hexadecimal constant.

Severity Level: Fatal

Description: An inappropriate character or blank follows the symbols &H in a hexadecimal integer constant.

User Action: Delete any inappropriate characters and spaces between the symbols &H and the first hexadecimal digit.

(BC 62) Magnitude of octal constant is too large.

Severity Level: Fatal

Description: The magnitude of an octal integer constant exceeds the maximum integer size of (2^63 - 1). In BASIC octal form, this magnitude is written as &O777777777777777777777.

User Action: Replace the invalid octal constant with a valid one.

(BC 63) Empty octal constant or misuse of ampersand.

Severity Level: Fatal

Description: An inappropriate character or space follows the symbols &O in an octal integer constant, or follows the symbol & in a hexadecimal or octal integer constant.

User Action: If the ampersand is a typographical error, correct it. Delete any inappropriate characters and spaces.

(BC 70) Compiler failure during integer constant conversion.

Severity Level: Catastrophic

Description: You have uncovered a problem with the NOS/VE BASIC compiler.

User Action: Follow the procedure established at your site for reporting software problems.

(BC 71) Compiler failure during conversion of real constant.

Severity Level: Catastrophic

Description: You have uncovered a problem with the NOS/VE BASIC compiler.

User Action: Follow the procedure established at your site for reporting software problems.

(BC 75) Numeric overflow in evaluation of constant expression.

Severity Level: Fatal

Description: The evaluation of a numeric expression used for an array bound in a DIM statement yields an overflow.

User Action: Correct the array bound dimension that caused the overflow.

(BC 76) Numeric underflow in evaluation of constant expression.

Severity Level: Fatal

Description: The evaluation of a numeric expression used for an array bound in a DIM statement yields an underflow.

User Action: Correct the array bound dimension that caused the underflow.

(BC 77) Numeric indefinite in evaluation of constant expression.

Severity Level: Fatal

Description: The evaluation of a numeric expression used for an array bound in a DIM statement yields an indefinite value.

User Action: Correct the array bound dimension that caused the indefinite value.

(BC 78) Divide fault in evaluation of constant expression.

Severity Level: Fatal

Description: The evaluation of a numeric expression used for an array bound in a DIM statement yields a divide fault.

User Action: Correct the array bound dimension that caused the divide fault.

(BC 100) The second letter alphabetically precedes the first.

Severity Level: Fatal

Description: The letter in the second component of a type declaration statement alphabetically precedes the letter in the first component.

User Action: See if the letter in the second component is a typographical error. Make the required correction.

(BC 101) Only a single letter is legal in this position in a type declaration statement.

Severity Level: Fatal

Description: In the letter list of a type declaration statement, only a single letter can appear as an individual item or as a component of a letter range. A list of identifiers is not permitted.

User Action: Check for extraneous characters on either side of a letter or letter range component.

(BC 102) The default type of identifiers beginning with the letter {character} already has been established at line {integer}, column {integer}.

Severity Level: Fatal

Description: Once a letter has been used (or its use has been implied by a range of letters) in a DEFINT, DEFSNG, DEFDBL or DEFSTR statement, it may not be so used or implied again.

User Action: Replace {character} by an unreserved identifier.

(BC 110) Misuse of reserved word {identifier}.

Severity Level: Fatal

Description: A type declaration statement that lists the letter {identifier} is redundant or contradicts a previous declaration.

User Action: Make sure you have specified the correct letter. Delete a redundant specification. Delete an inconsistent specification from one of the two type declaration statements, and modify any identifiers that are affected by the change.

(BC 120) EXTERNAL must be followed by FUNCTION or SUB.

Severity Level: Fatal

Description: The keyword FUNCTION or SUB must follow the keyword EXTERNAL in the specification of an external routine.

User Action: Supply the appropriate keyword FUNCTION or SUB. Check for extraneous characters between the keyword EXTERNAL and the keyword FUNCTION or SUB.

(BC 121) SUB or FUNCTION required.

Severity Level: Fatal

Description: The keyword FUNCTION or SUB that must follow the keyword DECLARE or the optional keyword EXTERNAL in a DECLARE statement is missing or misplaced.

User Action: Supply the appropriate keyword FUNCTION or SUB. Check for extraneous characters between the keyword DECLARE and the keyword EXTERNAL. Check for extraneous characters between the keyword DECLARE and the keyword FUNCTION or SUB.

(BC 125) Subroutine name is missing or is not a plain name.

Severity Level: Fatal

Description: The subroutine name in a CALL or CALLX statement is missing, is not a valid identifier, or is invalid because it contains a type specification symbol.

User Action: Supply a valid plain name for the subroutine name, and make sure it matches the name used in the corresponding SUB statement. Remember that an identifier must begin with a letter.

(BC 127) Routine name must be an identifier.

Severity Level: Fatal

Description: A name specified for a routine in a FUNCTION or SUB statement is not a valid identifier. The invalid name probably does not begin with a letter or may contain disallowed characters.

User Action: Make the required correction.

(BC 128) Expression function name must be an identifier.

Severity Level: Fatal

Description: A name used for an expression function in a DEF statement is not a valid identifier. The invalid name probably does not begin with a letter or may contain disallowed characters.

User Action: Make the required correction.

(BC 129) Subroutine name cannot contain a type specification.

Severity Level: Fatal

Description: A subroutine name in a SUB statement must be a plain name, that is, it cannot contain a type specification symbol.

User Action: Delete the type specification symbol from the subroutine name. Modify the CALL statements that are affected by the change.

(BC 140) An OPTION BASE must be 0 or 1.

Severity Level: Fatal

Description: Only the integer 0 or 1 can be specified for an OPTION BASE.

User Action: Specify the appropriate OPTION BASE 0 or 1. Use a DIM statement to specify a dimension lower bound other than 0 or 1.

(BC 141) OPTION BASE already has been set at line {integer}, column {integer}.

Severity Level: Fatal

Description: At most one OPTION BASE statement can appear in each external routine. An OPTION BASE for arrays in this external routine has already been established at line {integer}, column {integer}.

User Action: Specify the appropriate OPTION BASE 0 or 1. Delete a redundant specification.

(BC 142) OPTION BASE must be followed by an integer.

Severity Level: Fatal

Description: The OPTION BASE value is missing or is not an integer.

User Action: Supply the appropriate OPTION BASE: 0 or 1. Check for extraneous characters on either side of 0 or 1.

(BC 143) OPTION must be followed by BASE.

Severity Level: Fatal

Description: The keyword BASE must follow the keyword OPTION.

User Action: Supply the required keyword BASE. Check for extraneous characters on either side of the keyword BASE.

(BC 147) OPTION BASE statement must precede the DIM statement at line {integer}, column {integer}.

Severity Level: Fatal

Description: An OPTION BASE statement must precede the first DIM statement that uses the default lower bound specification. The DIM statement at line {integer}, column {integer} is such a statement.

User Action: Move the OPTION BASE statement so that it precedes the DIM statement at line {integer}, column {integer}.

(BC 151) OPTION BASE statement must precede the array reference at line {integer}, column {integer}.

Severity Level: Fatal

Description: In an external routine for which arrays are statically dimensioned, the OPTION BASE statement must precede the first reference to an array that is not declared in a DIM statement.

User action: Move the OPTION BASE statement so that it precedes the array reference at line {integer}, column {integer}.

(BC 160) Identifier required in COMMON list.

Severity Level: Fatal

Description: A COMMON list is missing or contains a name that is not a valid identifier. A valid identifier begins with a letter.

User Action: Supply the COMMON list or make the required correction to the identifier.

(BC 161) This declaration of the COMMON array {array name} is redundant; see also line {integer}, column {integer}.

Severity Level: Warning

Description: This declaration of the array {array name} as a COMMON array is redundant. The array {array name} first appears in a COMMON statement at line {integer}, column {integer}.

User Action: Make sure you have declared the correct COMMON item. Delete a redundant declaration. Replace {array name} with the correct formal array or scalar variable.

(BC 162) This declaration of the COMMON scalar {variable name} is redundant; see also line {integer}, column {integer}.

Severity Level: Warning

Description: This declaration of the scalar variable {variable name} as a COMMON scalar is not needed. The scalar variable name {variable name} first appears in a COMMON statement at line {integer, column {integer}}.

User Action: Make sure you have declared the correct COMMON item. Delete a redundant declaration. Replace {variable name} with the correct scalar variable or formal array.

(BC 164) COMMON declaration of {variable name} must precede its first use at line {integer}, column {integer}.

Severity Level: Fatal

Description: A COMMON statement declaring the array or scalar variable {variable name} as a COMMON item must precede all references to {variable name}. The first reference to {variable name} appears at line {integer}, column {integer}.

User Action: Move the COMMON statement so that it precedes the first reference to {variable name}.

(BC 168) This declaration of the COMMON array {array name} is inconsistent with the one at line {integer}, column {integer}.

Severity Level: Fatal

Description: The number of dimensions of array {array name} specified in a COMMON statement is inconsistent with the previous specification at line {integer}, column {integer}.

User Action: Make sure you have specified the correct array name. Delete the array declaration that specifies the incorrect number of dimensions. Make sure that the COMMON declaration for {array name} precedes the first reference to {array name}.

(BC 175) ERASE statement is not allowed in a statically dimensioned routine.

Severity Level: Fatal

Description: An ERASE statement is used in a BASIC routine for which arrays are statically dimensioned.

User action: Delete the ERASE statement from the statically dimensioned routine.

(BC 180) An array name must be an identifier.

Severity Level: Fatal

Description: A name used for an array in a DIM or ERASE statement is not a valid identifier.

User Action: The invalid name probably does not begin with a letter. Make the required correction.

(BC 183) Array name {array name} in DIM statement must be followed by an open parenthesis.

Severity Level: Fatal

Description: A left parenthesis character [(] must follow an array name {array name} in a DIM statement.

User Action: Supply the required open parenthesis. Check for extraneous characters between the array name {array name} and the open parenthesis.

(BC 185) Expected a close parenthesis after the dimension bounds of {array name}

Severity Level: Fatal

Description: The right parenthesis character [)] must immediately follow the dimension bounds of the array {array name} in a DIM statement.

User Action: Supply the missing close parenthesis. Check for extraneous characters between the dimension bound specification and the close parenthesis.

(BC 190) Static array {text} has already been dimensioned at line {integer}, column {integer}.

Severity Level: Fatal

Description: In an external routine in which arrays are statically dimensioned, an array can only be declared once by a DIM statement.

User Action: Locate the array {text} dimensioning at line {integer}, column {integer}, determine the correct declaration and delete the extranious ones.

(BC 191) DIM of static array {text} must precede the reference at line {integer}, column {integer}.

Severity Level: Fatal

Description: In an external routine for which arrays are statically dimensioned, a DIM statement declares an array that has dimensions already inferred because of a reference in a previous statement.

User Action: Locate the reference at line {integer}, column {integer} and the DIM statement, determine the correct dimensioning and delete the incorrect one.

(BC 192) Array parameter {text} cannot be dimensioned in a statically dimensioned routine.

Severity Level: Fatal

Description: In a routine with arrays dimensioned statically, a DIM statement contains an array parameter of that routine.

User Action: Locate the DIM statement and delete array parameter {text}.

(BC 195) DIM bound expression cannot be evaluated.

Severity Level: Fatal

Description: In a routine with arrays statically dimensioned, a DIM statement contains a numeric expression used for a dimension bound that is composed of operands other than numeric constants or operators other than "+", "-", "/", " ", and "*".

User Action: Locate the DIM bound expression and look for invalid operands or operators.

(BC 196) Array subscript upper bound is less than the lower bound.

Severity Level: Fatal

Description: In a routine in which arrays are statically dimensioned, a DIM statement has a dimension's upper bound value that is less than the dimension's lower bound.

User Action: Locate the DIM statement and verify the upper and lower bound values.

(BC 197) Magnitude of array bound cannot exceed 2^31-1.

Severity Level: Fatal

Description: In a DIM statement in a routine in which arrays are dimensioned statically, the magnitude of a dimension bound is greater than 2^31-1.

User Action: Locate the DIM statement and correct the dimension bound.

(BC 200) This declaration of {identifier} is redundant; see also line {integer}, column {integer}.

Severity Level: Warning

Description: This declaration of the routine name {identifier} is not needed. The name {identifier} was first used at line {integer}, column {integer}.

User Action: If {identifier} has been used to name more than one internal routine within this external routine (fatal error), rename one of the internal routines. Modify all routine calls that are affected by the change. If the specification statement for routine {identifier} appears before a DECLARE statement and contains the name {identifier}, a warning error results.

(BC 205) This declaration of {identifier} is redundant; see also line {integer}, column {integer}.

Severity Level: Fatal

Description: This declaration of the routine name {identifier} is redundant. The name {identifier} was used at line {integer}, column {integer}.

User Action: If {identifier} has been used to name more than one internal routine within this external routine (fatal error), rename one of the internal routines. Modify all routine calls that are affected by the change. If the specification statement for routine {identifier} appears before a DECLARE statement and contains the name {identifier}, a fatal error results.

(BC 215) The expression function {function name} was previously defined on line {integer}, column {integer}.

Severity Level: Fatal

Description: The expression function name {function name} has already been used to name an expression function at line {integer}, column {integer}.

User Action: Delete a redundant definition. Rename one of the functions and modify all corresponding function references.

(BC 217) Illegal reference to expression function {function name} in it own definition.

Severity Level: Fatal

Description: Expression function {function name} cannot be referenced in its own defining expression. An expression function cannot be defined recursively.

User Action: If recursion is intended, rewrite the function as a block function. Otherwise, use another identifier in place of {function name} in the defining expression.

(BC 220) Equal sign expected before defining expression in DEF statement.

Severity Level: Fatal

Description: The equal sign that separates the expression function name and the optional formal parameter list and the defining expression is missing or misplaced.

User Action: Supply the required equal sign. Check for extraneous characters between the parameter list or function name and the equal sign.

(BC 223) Expression function parameter must be an identifier.

Severity Level: Fatal

Description: The name used for a formal parameter of an expression function is not a valid identifier. The invalid name probably does not begin with a letter.

User Action: Make the required correction.

(BC 226) Name of expression function appears in its own parameter list.

Severity Level: Fatal

Description: A single identifier has been used in a DEF statement to name both the expression function and a formal parameter.

User Action: Make sure the correct function name has been provided. Rename the formal parameter.

(BC 228) Close parenthesis required after formal parameter list of expression function.

Severity Level: Fatal

Description: A right parenthesis character [)] must follow the formal parameter list of an expression function.

User Action: Supply the required close parenthesis. Check for extraneous characters between the last formal parameter and the close parenthesis.

(BC 230) Formal parameter {variable name} is not referenced in the definition of {function name}.

Severity Level: Warning

Description: Formal parameter {variable name} appears in the parameter list for expression function {function name}, but does not appear in the defining function.

User Action: See if the formal parameter {variable name} has been misspelled in or inadvertently omitted from the defining expression. Delete {variable name} from the parameter list if it is not needed in the defining expression.

(BC 232) No defintion of {function name} appears in the body of the function.

Severity Level: Warning

Description: A returning value has not been assigned within the body of function {function name}. Hence, either zero or the null string will be returned, depending on the data type of {function name}.

User Action: Within the function body, assign a returning value to the function name {function name} using the format:

```
let function_name = xxx
```

where xxx is an expression whose value is compatible with the data type established by {function name}.

(BC 240) Formal parameter must be an identifier.

Severity Level: Fatal

Description: A name specified for a formal parameter in a SUB or FUNCTION statement is not a valid identifier.

User Action: The invalid name probably does not begin with a letter. Make the required correction.

(BC 242) Formal parameter list is inconsistent with the actual parameter list at line {integer}, column {integer}.

Severity Level: Fatal

Description: A formal parameter list in a routine is inconsistent with a previously specified actual parameter list at line {integer}, column {integer}.

User Action: Make sure there is a one-to-one correspondence between the formal and actual parameter lists. For routines, a real value cannot be passed to an integer formal parameter. An integer value cannot be passed to a real formal parameter. Make sure that corresponding formal and actual parameters have the same data type.

(BC 245) The subscript or parameter list of {identifier} is inconsistent with that at line {integer}, column {integer}.

Severity Level: Fatal

Description: The subscript list of array {identifier} or the parameter list of routine {identifier} is inconsistent with the previous one at line {integer}, column {integer}.

User Action: Check the number of subscripts in the references to array {identifier}. Change the incorrect reference. Make sure there is a one-to-one correspondence between the formal and actual parameter lists of routine {identifier}. Make sure corresponding parameters have the same data type. Remember that a real value cannot be passed to an integer formal parameter. An integer value cannot be passed to a real formal parameter.

(BC 250) This use of the internal function name {function name} is illegal except within {function name}.

Severity Level: Fatal

Description: A value can be assigned to the name of a function only from within that function.

User Action: If this error occurs within a function, perhaps the current function name was intended.

(BC 251) This use of external function name {function name} is illegal except with {function name}.

Severity Level: Fatal

Description: A value can be assigned to the name of function from only within that function.

User Action: If this error occurs within a function, perhaps the current function name was intended.

(BC 260) Reference to an undefined internal {FUNCTION or SUB} appears at line {integer}, column {integer}.

Severity Level: Fatal

Description: The internal {FUNCTION or SUB} is first referenced at line {integer}, column {integer}, but this routine has not been defined. The {FUNCTION or SUB} is replaced by FUNCTION or SUB.

User Action: Make sure the name {FUNCTION or SUB} is the correct routine name. See if the routine has been omitted from the program.

(BC 270) This use of {name} conflicts with the previous use at line {integer}, column {integer}.

Severity Level: Fatal

Description: This use of the identifier {name} is inconsistent with the prior use at line {integer}, column {integer}.

User Action: See if the present use of {name} is correct. Use distinct identifiers for distinct uses.

(BC 300) Internal routines may not contain other routines.

Severity Level: Fatal

Description: An internal routine cannot contain another routine.

User Action: Look for two routines that overlap. Rethink the structure of your program.

(BC 310) This is the {integer*} main program in this input file.

Severity Level: Warning

Description: The compiler has discovered more than one main program in the source file.

User Action: If an external routine is followed by a subprogram, the closing statement (END FUNCTION, END PROGRAM, or END SUB) of the external routine must be immediately followed by the subprogram specification statement. If any lines intervene, even comments, the compiler interprets them as a main program. Delete any such intervening lines.

(BC 320) END {identifier} is illegal at this point.

Severity Level: Fatal

Description: An END {identifier} statement appears in an inappropriate location. The {identifier} is replaced by the keyword FUNCTION, PROGRAM, or SUB.

User Action: An END {identifier} statement marks the physical end of a routine. Replace {identifier} with the appropriate keyword FUNCTION, PROGRAM, or SUB to identify the correct kind of routine (function, main program, or subroutine).

(BC 325) Nothing can follow an END PROGRAM statement on a line.

Severity Level: Fatal

Description: When an END PROGRAM statement is required, it must be the last statement in the main program's last line. No non-blank characters in the line that contains the END PROGRAM statement can follow the keyword PROGRAM.

User Action: Check for extraneous characters following the keyword PROGRAM. Make the required correction.

(BC 330) Nothing can follow the END {FUNCTION or SUB} statement on a line.

Severity Level: Fatal

Description: The END {FUNCTION or SUB} statement for an external routine must be the last statement in the routine's last line. No non-blank characters in the same line as this statement can follow the keyword {FUNCTION or SUB} The {FUNCTION or SUB} is replaced by the keyword FUNCTION or SUB.

User Action: Check for extraneous characters following the keyword {FUNCTION or SUB}. Make the required correction.

(BC 340) End of source file reached without finding an END {FUNCTION or SUB} for {name}.

Severity Level: Fatal

Description: An external or internal function must end with an END FUNCTION statement. An external or internal subroutine must end with an END SUB statement.

User Action: Supply the appropriate END FUNCTION or END SUB statement. Check for extraneous characters or missing spaces in the vicinity of these phrases.

(BC 400) Compiler bug encountered in declaration of {name}.

Severity Level: Catastrophic

Description: You have uncovered a problem with the NOS/VE BASIC compiler.

User Action: Follow the procedure established at your site for reporting software problems.

(BC 410) Compiler failure--DATA statement has an empty body.

Severity Level: Catastrophic

Description: You have uncovered a problem with the NOS/VE BASIC compiler.

User Action: Follow the procedure established at your site for reporting software problems.

(BC 500) Label must be an integer.

Severity Level: Fatal

Description: A label reference in a GOSUB, GOTO, ON-GOSUB, or ON-GOTO statement is missing or is not a positive integer. Note that a label must be a positive integer of at most six decimal digits.

User Action: Supply a valid label reference. Check for extraneous characters or missing spaces on either side of the label reference.

(BC 501) Zero is not a legal label.

Severity Level: Fatal

Description: A label must be a positive integer that does not exceed 999,999. Zero is not a valid NOS/VE BASIC label.

User Action: Replace 0 with a valid label. Remember that each label in an external routine must be greater than all preceding labels in that external routine.

(BC 502) Label cannot exceed 999,999.

Severity Level: Fatal

Description: A label used to address a line must be a positive integer of at most six decimal digits.

User Action: Supply a valid label and modify all label references that are affected by the change.

(BC 503) Label not strictly greater than its predecessor.

Severity Level: Fatal

Description: Each label in an external routine must be greater than all preceding labels in that external routine.

User Action: Relabel the line, and others as needed, so that the labels in the external routine form an increasing sequence.

(BC 506) Illegal reference to label zero.

Severity Level: Fatal

Description: A reference to 0 instead of a valid label reference appears in a GOSUB, GOTO, line IF, ON-GOSUB, ON-GOTO, or RESTORE statement. Note that a label must be a positive integer of at most six digits.

User Action: Replace the reference to 0 with an appropriate label reference.

(BC 508) Label reference required.

Severity Level: Fatal

Description: A label reference in a GOSUB, GOTO, ON-GOSUB, or ON-GOTO statement is missing, or is not a positive integer. Note that a label must be a positive integer of at most six digits.

User Action: Supply a valid label reference. Check for extraneous characters or missing spaces on either side of the label reference.

(BC 512) Label {integer} is undefined.

Severity Level: Fatal

Description: The label {integer} is being referenced, but has not been defined.

User Action: Make sure you referenced the correct label. Provide the appropriate line with the label {integer}.

(BC 515) This reference to label {integer} enters or exits an internal routine illegally.

Severity Level: Fatal

Description: A GOSUB, GOTO, ON ERROR, ON-GOSUB, ON-GOTO, or RESUME statement uses label {integer} to branch illegally into or out of an internal routine.

User Action: Make sure that {integer} is the correct destination of the branch. Reconsider the structure of the internal routine and the structure and placement of the destination code. Perhaps, after reorganization, a function or subroutine call can replace the statement that caused the illegal branch.

(BC 518) Label or zero required in ON ERROR GOTO statement.

Severity Level: Fatal

Description: A label reference or the value 0 must follow the keywords ON ERROR GOTO.

User Action: Supply the required label or value 0. Check for extraneous characters or missing spaces on either side of the label reference or value 0.

(BC 520) Expected GOTO or THEN to match the IF at line {integer}, column {integer}.

Severity Level: Fatal

Description: The keyword GOTO or THEN that must follow the test expression in an IF statement is missing or misplaced.

User Action: Supply the appropriate keyword GOTO or THEN and the subsequent statement or statements. Check for extraneous characters between the test expression and the keyword GOTO or THEN.

(BC 521) GOTO or GO TO required.

Severity Level: Fatal

Description: The keyword GOTO or the equivalent two-word phrase GO TO must follow the keywords ON ERROR.

User Action: Supply the required keyword GOTO or the two-word phrase GO TO. Check for extraneous characters or missing spaces on either side of the keyword GOTO, GO, or TO.

(BC 522) GOSUB or GOTO required.

Severity Level: Fatal

Description: The keyword GOSUB or GOTO must follow the numeric expression in an ON-GOSUB or ON-GOTO statement.

User Action: Supply the appropriate keyword GOSUB or GOTO. Check for extraneous characters or missing spaces on either side of the keyword GOSUB or GOTO.

(BC 523) Expected SUB or TO following GO.

Severity Level: Fatal

Description: The keyword GO must be followed by the keyword SUB or TO.

User Action: Supply the appropriate keyword SUB or TO. Check for extraneous characters on either side of the keyword SUB or TO. Note that the two-word phrases GO SUB and GO TO can be written as the single words GOSUB and GOTO.

BC 530) EXIT must be followed either by FUNCTION or by SUB.

Severity Level: Fatal

Description: The keyword FUNCTION or SUB must follow the keyword EXIT.

User Action: Supply the appropriate keyword FUNCTION or SUB. Check for extraneous characters on either side of the keyword FUNCTION or SUB.

(BC 531) EXIT is illegal in a main program.

Severity Level: Fatal

Description: The appearance of the keyword EXIT in a main program is illogical when it appears outside of an internal routine. In addition, EXIT must always be followed by the keyword FUNCTION or SUB, and must appear within an internal routine or a subprogram.

User Action: Use an END statement if you intended to terminate the program. If EXIT FUNCTION or EXIT SUB is what you intended, supply the appropriate keyword FUNCTION or SUB, and move this statement to its correct position within an internal routine.

(BC 534) EXIT {FUNCTION or SUB} is illegal when the immediately containing routine is {name}.

Severity Level: Fatal

Description: An EXIT {FUNCTION or SUB} statement makes no sense within the context of {name}.

User Action: Change {FUNCTION or SUB} so that it corresponds to the context of {name}. That is, replace {FUNCTION or SUB} keyword FUNCTION if {name} is the phrase AN EXTERNAL SUBROUTINE or AN INTERNAL SUBROUTINE. Replace {FUNCTION or SUB} with the keyword SUB if {name} is the phrase AN EXTERNAL FUNCTION or AN INTERNAL FUNCTION.

(BC 540) This IF and the block that begins at column {integer} are not properly nested.

Severity Level: Fatal

Description: The ELSE or THEN clause of a line IF statement must completely contain the block structure that begins at column {integer}.

User Action: See if the ELSE or THEN clause contains a block structure, such as a FOR-NEXT loop or IF construction, that carries over to a second line. Remember that a line IF construction must be completely contained in one line. If necessary, replace the line IF with a block IF construction.

(BC 541) This ELSE and the block which begins at column {integer} are not properly nested.

Severity Level: Fatal

Description: The ELSE clause of a line IF statement must completely contain the block structure that begins at column {integer}.

User Action: Make the required correction. If the block structure will not fit in the remaining space in the line, use a block IF construction.

(BC 542) This line IF and the block which ends at column {integer} are not properly nested.

Severity Level: Fatal

Description: The components of a line IF statement and the block structure that begins at column {integer} are not properly nested.

User Action: If you are using a line IF construction, see if a block structure, such as a FOR-NEXT loop, in the THEN clause carries over to a second line and is followed by an ELSE clause. Remember that a line IF construction must be completely contained in one line. If necessary, replace the line IF with a block IF construction.

(BC 546) Line IF requires an end-of-statement in this position.

Severity Level: Fatal

Description: A colon or end-of-line is required in this position in a line IF construction.

User Action: If the construction ends with a complete statement, a colon to separate two statements in the THEN clause has probably been omitted. Supply the required colon. Otherwise, check for extraneous characters after the last complete statement in the THEN clause.

(BC 550) Expected an ENDIF to close the ELSE at line {integer}, column {integer}.

Severity Level: Fatal

Description: If BLOCK or ELSE is left unclosed, the ENDIF is required.

User Action: Supply the required ENDIF statement. Check for extraneous characters on either side of the keyword ENDIF.

(BC 552) Expected an ELSEIF, ELSE, or ENDIF, to follow the {IF or ELSE} at line {integer}.

Severity Level: Fatal

Description: A block IF construction cannot end with a THEN block. If any ELSEIF components appear, the {IF or ELSE} is replaced by the keyword ELSEIF, and the last ELSEIF component is located. Otherwise, the {IF or ELSE} is replaced by the keyword IF, and the opening IF component is located.

User Action: Supply the missing ENDIF statement to complete the block IF construction. Determine whether any ELSEIF components and the ELSE block have been inadvertently omitted.

(BC 556) Expression following an {IF or ELSEIF} must be numeric.

Severity Level: Fatal

Description: The test condition for an {IF or ELSEIF} component of a block IF construction must be a numeric expression.

User Action: Make the required correction.

(BC 560) Expression following WHILE must be numeric.

Severity Level: Fatal

Description: The expression in a WHILE statement must be numeric.

User Action: Make the required correction.

(BC 563) WHILE statement at line {integer}, column {integer} has no corresponding WEND.

Severity Level: Fatal

Description: The loop, which begins with the WHILE statement at line {integer}, column {integer}, has not been closed with a WEND statement.

User Action: Supply the missing WEND statement.

(BC 570) NEXT required for FOR at line {integer}, column {integer}.

Severity Level: Fatal

Description: A loop that begins with the FOR statement at line {integer}, column {integer} has not been closed by a corresponding NEXT statement.

User Action: Supply the required NEXT statement. Check for extraneous characters on either side of the keyword NEXT.

(BC 571) Error in control variable; should have been {name}.

Severity Level: Fatal

Description: If a control variable is specified on a NEXT statement, it must correspond to a previous FOR statement at the same nesting level.

User Action: Supply the required control variable name.

(BC 573) TO required.

Severity Level: Fatal

Description: The keyword TO that must follow the initial value of the control variable in a FOR statement is missing or misplaced.

User Action: Supply the required keyword TO. Check for extraneous characters or missing spaces on either side of the keyword TO.

Example:

```
FOR I=1 TO N STEP 5
```

(BC 578) Control variable must be an identifier.

Severity Level: Fatal

Description: A name used for a control variable in a FOR statement is not a valid identifier.

User Action: The invalid name probably does not begin with a letter. Make the required correction.

(BC 579) Control variable must be numeric.

Severity Level: Fatal

Description: The control variable in a FOR-NEXT loop must be a numeric identifier.

User Action: Supply the required numeric variable. Check for the appearance of the $ type specification symbol instead of the intended symbol. If the invalid name is a plain name, replace it with a plain name that has not been typed with a DEFSTR statement. Be sure to change all references to the control variable.

(BC 1000) I/O mode must be APPEND, INPUT, or OUTPUT.

Severity Level: Fatal

Description: An I/O mode of APPEND, INPUT, or OUTPUT must be specified after the keyword FOR in this form of the OPEN statement. Omit the I/O mode and the keyword FOR to specify the default I/O mode of RANDOM.

User Action: Supply the appropriate keyword APPEND, INPUT, or OUTPUT. Check for extraneous characters between the keyword FOR and the I/O mode.

Example:

```
OPEN "FILENAME" FOR OUTPUT AS #1
OPEN "RANDOM_FILE" AS #8
```

(BC 1005) Expected "AS channel number".

Severity Level: Fatal

Description: The compiler expects the keyword AS to appear in this position in an OPEN statement.

User Action: In format (A) of the OPEN statement, the I/O mode must be followed by the keyword AS and a channel reference. Supply the keyword and a channel reference. Check for extraneous characters on either side of the keyword FOR. In format (B) of the OPEN statement, a comma must follow the I/O mode. Supply the required comma.

Examples:

```
OPEN "FILENAME" FOR OUTPUT AS #5

OPEN "OUTPUT",#5,"FILENAME"
```

(BC 1010) Expected = after LEN in record length specification.

Severity Level: Fatal

Description: The equal sign that must follow the keyword LEN when specifying file record length in an OPEN statement is missing or misplaced.

User Action: Supply the required equal sign. Check for extraneous characters between the keyword LEN and the equal sign.

Example:

```
OPEN "FILENAME" FOR OUTPUT AS #8 LEN=64
```

(BC 1015) Expected a comma between channel number and file name.

Severity Level: Fatal

Description: A comma must follow the channel reference in this form of the OPEN statement.

User Action: Supply the required comma. Check for extraneous characters between the channel reference and the comma.

Example:

```
OPEN "OUTPUT",#2,"FILENAME"
```

(BC 1050) A comma is required after the channel number in an INPUT statement.

Severity Level: Fatal

Description: A comma is required after the channel reference in an INPUT statement.

User Action: Supply the required comma. Check for extraneous characters between the channel reference and the comma.

(BC 1055) A comma or semicolon is required after the prompt in an INPUT statement.

Severity Level: Fatal

Description: The comma or semicolon after the prompt string in an INPUT statement is missing or misplaced.

User Action: Supply the appropriate comma or semicolon. Check for extraneous characters between the prompt string and the comma or semicolon.

(BC 1070) LINE can only be followed by INPUT.

Severity Level: Fatal

Description: The keyword INPUT that must follow the keyword LINE is missing or misplaced.

User Action: Supply the required keyword INPUT. Check for extraneous characters or missing spaces on either side of the keyword INPUT.

(BC 1075) Channel expression must be followed by a comma in a LINE INPUT statement.

Severity Level: Fatal

Description: The comma that must follow the channel reference in a LINE INPUT statement is missing or misplaced.

User Action: Supply the required comma. Check for extraneous characters between the channel reference and the comma.

(BC 1080) A comma or semicolon is required after the PROMPT in an LINE INPUT statement.

Severity Level: Fatal

Description: The comma or semicolon after the prompt string in a LINE INPUT statement is missing or misplaced.

User Action: Supply the appropriate comma or semicolon. Check for extraneous characters between the prompt string and the comma or semicolon.

(BC 1085) The input item in a LINE INPUT statement must be a string variable.

Severity Level: Fatal

Description: The input item in a LINE INPUT statement must be a string variable.

User Action: Replace your input item with a valid string variable. Check for a missing type specification symbol $ or a mistake in a DEFSTR statement.

(BC 1100) Comma required after channel number in PRINT statement.

Severity Level: Fatal

Description: The comma that must follow the channel reference in a PRINT statement is missing or misplaced.

User Action: Supply the required comma. Check for extraneous characters between the channel reference and the comma.

(BC 1110) Format string must be followed by a semicolon.

Severity Level: Fatal

Description: A semicolon must follow the format string in a PRINT USING statement.

User Action: Supply the required semicolon. Check for extraneous characters between the format string and the semicolon.

(BC 1120) Open parenthesis required after SPC or TAB.

Severity Level: Fatal

Description: The left parenthesis character [(] that must follow the SPC or TAB library function name is missing or misplaced.

User Action: Supply the required open parenthesis. Check for extraneous characters between the function name and the open parenthesis.

(BC 1121) Close parenthesis required after argument of SPC or TAB.

Severity Level: Fatal

Description: The right parenthesis character [)] for an SPC or TAB library function reference is missing or misplaced.

User Action: Supply the required close parenthesis. Check for extraneous characters between the argument and the close parenthesis.

(BC 1130) Comma required after channel number in WRITE statement.

Severity Level: Fatal

Description: A comma must follow the channel reference in a WRITE statement.

User Action: Supply the required comma, even if you omit the write list. Check for extraneous characters between the channel reference and the comma.

Examples:

```
WRITE #7,A

WRITE #5,
```

(BC 1150) Premature end-of-statement in WIDTH statement--no width has been specified.

Severity Level: Fatal

Description: The channel reference or special file reference in a WIDTH statement must be followed by a comma and a page width specification.

User Action: Supply the required comma and page width.

Examples:

```
WIDTH #3,65

WIDTH "PRINT",72
```

(BC 1155) Comma required after file name in WIDTH statement.

Severity Level: Fatal

Description: In this form of the WIDTH statement, a comma must follow the special file reference "PRINT" or "OUTPUT".

User Action: Supply the required comma. Check for extraneous characters between the file name and the comma.

(BC 1160) Comma or end-of-statement required after numeric expression in WIDTH statement.

Severity Level: Fatal

Description: A comma or end-of-statement is required after the numeric expression in a WIDTH statement.

User Action: If the numeric expression denotes a channel reference, supply the required comma. Check for extraneous characters between the channel reference and the comma. If the numeric expression denotes a page width for the NOS/VE standard file $OUTPUT, supply the required end-of-statement. Check for extraneous characters between the channel reference and the end-of-statement.

(BC 1200) A comma is required at this position in a FIELD statement.

Severity Level: Fatal

Description: A comma must follow the channel reference in a FIELD statement.

User Action: Supply the required comma. Check for extraneous characters between the channel reference and the comma.

(BC 1205) The reserved word AS is required at this position in a FIELD statement.

Severity Level: Fatal

Description: The keyword AS that must follow a field length specification in a FIELD statement is missing or misplaced.

User Action: Supply the required keyword AS. Check for extraneous characters or missing spaces on either side of the keyword AS.

Example:

```
FIELD #3, 30 AS S$, 20 AS T$
```

(BC 1210) A field name is required at this position in a FIELD statement.

Severity Level: Fatal

Description: A string identifier is required in this position in a FIELD statement to name a field.

User Action: Supply the required string identifier. See if the identifier has been inadvertently specified as type string.

(BC 1215) Field name {name} is not of type string.

Severity Level: Fatal

Description: A string identifier is required in this position in a FIELD statement to name a field.

User Action: Supply the required string identifier. See if the identifier has been inadvertently specified as type string.

(BC 1220) A substring cannot be used as a field name.

Severity Level: Fatal

Description: A substring expressed with colon substring notation or a MID$ reference cannot be used to name a field in a FIELD statement.

User Action: Assign the substring to a string identifier and use the string identifier to name the field.

(BC 1250) Lefthand side of {LSET or RSET} statement must be a string identifier.

Severity Level: Fatal

Description: The lefthand side of an LSET or RSET statement must be a string identifier. Note that a substring expressed with colon-substring notation or a MID$ reference is not permitted.

User Action: Supply the required string identifier. See if a dollar sign type specification symbol is missing.

(BC 1255) A substring cannot appear on the lefthand side of an {LSET or RSET} statement.

Severity Level: Fatal

Description: A substring expressed with colon substring notation or a MID$ reference cannot appear to the left of the equal sign in an {LSET or RSET} statement. The {LSET or RSET} is replaced by the keyword LSET or RSET.

User Action: Assign the substring to a string identifier and use the string identifier on the lefthand side of the {LSET or RSET} statement.

(BC 1260) An equal sign is required at this position in an {LSET or RSET} statement.

Severity Level: Fatal

Description: The equal sign that must follow the string identifier in the lefthand side of an {LSET or RSET} statement is missing or misplaced. The {LSET or RSET} is replaced by the keyword LSET or RSET.

User Action: Supply the required equal sign. Check for extraneous characters between the keyword LSET or RSET and the equal sign.

(BC 2005) Identifier is too long ({integer} characters); replaced with {identifier}.

Severity Level: Fatal

Description: An identifier that contains {integer} characters is too long. The invalid identifier has been replaced by the 31-character identifier {identifier} so that the compiler can continue to check for errors. The {identifier} is replaced by the 31-character name consisting of the first 30 characters of the invalid identifier followed by the last character of the invalid identifier.

User Action: Replace the invalid identifier with a valid identifier of at most 31 characters. Note that some other errors might have occurred as a result of the substitution of {name} for the invalid identifier.

(BC 2010) Expected an identifier.

Severity Level: Fatal

Description: A variable name that is required in this position is missing or is not a valid identifier.

User Action: Supply the required variable name. Check for extraneous characters before the variable name.

(BC 2020) String expression required.

Severity Level: Fatal

Description: A string expression is required in this position because of the context of the statement.

User Action: Analyze the context of the statement. Supply the required string expression or change the context. Make sure all identifiers have the appropriate data type. In particular, look for an incorrect or missing type specification symbol or DEFSTR statement.

(BC 2021) Numeric expression required.

Severity Level: Fatal

Description: A numeric expression is required in this position because of the context of the statement.

User Action: Analyze the context of the statement. Supply the required numeric expression or change the context. Make sure all identifiers have the appropriate data type. In particular, look for an incorrect or missing type specification symbol.

(BC 2025) Unclosed quoted string.

Severity Level: Fatal

Description: The closing quotation mark for a quoted string constant is missing.

User Action: Supply the required closing quotation mark.

(BC 2050) Catastrophic error in EMIT_STACKED_OPERATOR.

Severity Level: Catastrophic

Description: You have uncovered a problem with the NOS/VE BASIC compiler.

User Action: Follow the procedure established at your site for reporting software problems.

(BC 2060) This operator cannot be applied to a string operand.

Severity Level: Fatal

Description: A string operand is incompatible with the operator that is acting on it.

User Action: Make the data type of the operand compatible with the operator that is acting on it. Remember that logical and relational expressions have numeric values.

(BC 2070) Numeric and string operands cannot be mixed.

Severity Level: Fatal

Description: The data types of operands in an expression are incompatible.

User Action: Make sure all identifiers used in the expression have the appropriate data type. In particular, look for an incorrect or missing type specification symbol.

(BC 2075) Statement ends prematurely; an operand is expected.

Severity Level: Fatal

Description: An incomplete statement has been encountered. The compiler is expecting an operand or expression to complete the statement.

User Action: See if the ending portion of the statement has been inadvertently omitted. Most likely, an expression ends with an operator. Supply the subsequent operand.

(BC 2077) Premature end of statement.

Severity Level: Fatal

Description: The second variable in a SWAP statement is missing.

User Action: Supply the required variable.

(BC 2080) Operand cannot begin with the character {character}.

Severity Level: Fatal

Description: The character {character} is illegal as the first character of an operand.

User Action: See if an operand that should appear before the character {character} has been inadvertently omitted. Check to see if {character} is not an extraneous character.

(BC 2085) Character with ASCII decimal code {integer} cannot appear in an operand.

Severity Level: Fatal

Description: The character with ASCII decimal code {integer} cannot be used within an operand.

User Action: The illegal character is probably a typographical error. Make the required correction.

(BC 2100) Close parenthesis expected to match the open parenthesis at line {integer}, column {integer).

Severity Level: Fatal

Description: The right parenthesis character [)] that corresponds to the left parenthesis character [(] at line {integer}, column {integer} is missing or misplaced.

User Action: Supply the required close parenthesis. Check for extraneous characters before the close parenthesis.

(BC 2120) Expected close parenthesis to end the subscript which begins at line {integer}, column {integer}.

Severity Level: Fatal

Description: The close parenthesis for the subscript list that begins at line {integer}, column {integer} is missing or misplaced.

User Action: Supply the required close parenthesis. Check for extraneous characters between the last subscript and the close parenthesis.

(BC 2125) Close parenthesis or comma required in skeleton subscript.

Severity Level: Fatal

Description: A formal array in a COMMON, FUNCTION, or SUB statement contains a character other than a comma, or is missing a close parenthesis.

User Action: Make sure the correct number of commas have been provided. Check for extraneous characters within the formal array. Supply the required close parenthesis.

(BC 2127) Too many commas in skeleton subscript.

Severity Level: Fatal

Description: The number of dimensions of an array is limited only by the ability to fit an array element reference on a single line. The number of dimensions of a formal array, which is one more than the number of commas listed, makes such a reference impossible.

User Action: Redesign the algorithm using a group of smaller arrays.

(BC 2150) First parameter of LBOUND or UBOUND function reference must be an array name.

Severity Level: Fatal

Description: The first parameter of an LBOUND or UBOUND function reference must be an array name.

User Action: The invalid parameter probably does not begin with a letter. Make the required correction.

(BC 2155) Expected open parenthesis after LBOUND or UBOUND.

Severity Level: Fatal

Description: A left parenthesis character [(] must follow the library function name LBOUND or UBOUND.

User Action: Supply the required open parenthesis. Check for extraneous characters between the function name and the open parenthesis.

(BC 2160) Error in array name in ERASE statement.

Severity Level: Fatal

Description: A name used for an array in a DIM or ERASE statement is not a valid identifier.

User Action: The invalid name probably does not begin with a letter. Make the required correction.

(BC 2180) Use of {identifier} is illegal at this point.

Severity Level: Fatal

Description: The appearance of the item {identifier} in this position is illegal.

User Action: This is a very general error message. Make sure that {identifier} is not garbled. Make sure that the statement that contains {identifier} has the appropriate keywords spelled correctly. Check that this statement is properly located with respect to the statements around it and the context. This message often occurs as a result of some previous error that cause the compiler to get out of step.

(BC 2185) Equal sign expected.

Severity Level: Fatal

Description: An equal sign must follow the control variable in a FOR statement.

User Action: Supply the required equal sign. Check for extraneous characters between the control variable and the equal sign. Remember that a control variable cannot be an array element.

(BC 2190) Comma required between SWAP variables.

Severity Level: Fatal

Description: A comma must follow the first variable in a SWAP statement.

User Action: Supply the required comma. Check for extraneous characters between the first variable and the comma.

(BC 2195) String and numeric variables cannot be swapped.

Severity Level: Fatal

Description: The variables in a SWAP statement have incompatible data types. They must both be numeric or both be string.

User Action: Most likely, a type specification symbol has been incorrectly specified or is missing. Make the required correction.

(BC 2200) String array cannot be passed with the CALLX statement.

Severity Level: Fatal

Description: A string array cannot be passed to a FORTRAN (or other language subroutine) with a CALLX statement.

User Action: See if the array has been inadvertently specified as type string. A subroutine written in a language other than BASIC that has a string array as a formal parameter cannot be accessed from NOS/VE BASIC. Convert the subroutine to a BASIC subroutine.

(BC 2210) An array or string was used as a substring bound.

Severity Level: Fatal

Description: The beginning or ending position of a substring expressed with colon-substring notation cannot be a formal array or string expression.

User Action: See if a subscript has been omitted from an array element. Check for an incorrect type specification symbol at the end of an identifier.

(BC 2220) MID$ reference is missing an open parenthesis.

Severity Level: Fatal

Description: A left parenthesis character [(] must follow the keyword MID$.

User Action: Supply the required open parenthesis. Check for extraneous characters between the keyword MID$ and the open parenthesis.

(BC 2221) Expected close parenthesis to end the MID$ reference at line {integer}, column {integer}.

Severity Level: Fatal

Description: A right parenthesis character [)] must follow the last parameter in a MID$ statement or library function reference.

User Action: Supply the required close parenthesis. Check for extraneous characters between the last parameter and the close parenthesis.

(BC 2225) Comma expected between first and second parameter of MID$ reference at line {integer}, column {integer}.

Severity Level: Fatal

Description: A comma must follow the first parameter in a MID$ function reference or a MID$ statement.

User Action: Supply the required comma. Check for extraneous characters between the first parameter and the comma.

(BC 2226) Expected comma after string in MID$ reference.

Severity Level: Fatal

Description: A comma must follow the first parameter in a MID$ function reference or a MID$ statement.

User Action: Supply the required comma. Check for extraneous characters between the first parameter and the comma.

(BC 2230) Substring or MID$ reference not allowed as first argument of MID$ reference in this context.

Severity Level: Fatal

Description: The first parameter in a MID$ statement must be a string identifier. A MID$ reference or a substring are not permitted.

User Action: Assign the value of the MID$ reference or substring to a string identifier, and use the string identifier as the first parameter.

(BC 2235) Expected a colon in the substring bounds specification that begins at line {integer}, column {integer}.

Severity Level: Fatal

Description: A colon must separate the bounds of the substring that is being assigned a value. The colon in the substring bound specification that begins at line {integer}, column {integer} missing or misplaced.

User Action: Supply the required colon. Check for extraneous characters between the first bound specification and the colon.

(BC 2240) Error in specification of substring bounds.

Severity Level: Fatal

Description: The bound specification for a substring in a PRINT list contains an error.

User Action: Most likely, the colon that must separate the substring bounds is missing or misplaced. Supply the required colon. Check for extraneous characters between the first bound specification and the colon.

(BC 2250) Expected open parenthesis after LEN.

Severity Level: Fatal

Description: A left parenthesis character [(] must follow the word LEN for a LEN library function reference.

User Action: Supply the required open parenthesis. Check for extraneous characters between the keyword LEN and the open parenthesis.

(BC 4025) INPUT file (file reference) is not available.

Severity Level: Fatal

Description: The input file specified for the RESEQUENCE utility is not found.

User Action: Check that the correct file was specified.

(BC 4030) INPUT file (file reference) is not available for READ access.

Severity Level: Fatal

Description: The input file specified for the RESEQUENCE utility is not available for READ access.

User Action Check that the correct file was specified. Use the SET_FILE_ATTRIBUTES command to change the input file ACCESS_MODE to READ. To display information about a file use the DISPLAY_FILE_ATTRIBUTES command.

(BC 4035) INPUT file (file reference) specified for RESEQUENCE is empty.

Severity Level: Fatal

Description: The input file specified for the RESEQUENCE utility is empty.

User Action: Check that the correct file was specified.

(BC 4040) INPUT file (file reference) specified for RESEQUENCE cannot be read as a sequential file.

Severity Level: Fatal

Description: The input file specified for the RESEQUENCE utility is defined with attributes that prevent it from being read as a sequential file.

User Action: Check that the correct file was specified. Use the SET_FILE_ATTRIBUTES command to change the FILE_ORGANIZATION to SEQUENTIAL. To display information about a file use the DISPLAY_FILE_ATTRIBUTES command.

(BC 4055) Empty OUTPUT file (file reference) specified for RESEQUENCE must have APPEND access.

Severity Level: Fatal

Description: The output file specified for the RESEQUENCE utility is empty and is not available for append access.

User Action: Check that the correct file was specified. Use the SET_FILE_ATTRIBUTES command to change the output file ACCESS_MODE to APPEND. To display information about a file use the DISPLAY_FILE_ATTRIBUTES command.

(BC 4065) OUTPUT file (file reference) specified for RESEQUENCE cannot be written as a sequential file.

Severity Level: Fatal

Description: The output file specified for the RESEQUENCE utility is defined with attributes that prevent it from being written as a sequential file.

User Action: Check that the correct file was specified. Use the SET_FILE_ATTRIBUTES command to change the output FILE_ORGANIZATION to SEQUENTIAL. To display information about a file use the DISPLAY_FILE_ATTRIBUTES command.

(BC 4070) Resequenced line (integer) exceeds the maximum record length allowed on file (file reference).

Severity Level: Fatal

Description: The output file specified for the RESEQUENCE utility is a preexisting non-empty file with a record type of fixed length and a line of the resequenced text exceeds the length defined for the files records.

User Action: Check that the correct file was specified. Use the SET_FILE_ATTRIBUTES command to change the output file RECORD_TYPE to VARIABLE. The default record type is VARIABLE. To display information about a file use the DISPLAY_FILE_ATTRIBUTES command.

(BC 4120) A new label generated from RESEQUENCE parameters exceeds 999,999, the maximum label value.

Severity Level: Fatal

Description: The result of applying the new base and new increment to the input file lead to a new label with a numeric value greater than 999,999 (that is, if ((NLL - 1) * NI) + NB is greater than 999,999, where NLL is the number of labeled lines, NI is the new increment, and NB is the new base).

User Action: Supply a valid new label and modify all label references that are affected by the change.

(BC 4125) The old label defined at line (integer) exceeds 999,999, the maximum label value.

Severity Level: Fatal

Description: The input file contains a reference of a label that exceeds the value of the maximum legal label value.

User Action: Supply a valid label and modify all label references that are affected by the change.

(BC 4130) The label referenced at line (integer), column (integer) is not defined.

Severity Level: Fatal

Description: The input file contains a reference of a label that is not defined in the containing routine.

User Action: Make sure you referenced the correct label. Provide the appropriate line with the label (integer).

(BC 4135) The label definition at line (integer) of the RESEQUENCE input file is not greater than the preceding label definition.

Severity Level: Fatal

Description: The input file contains a label definition the value of which is not greater than that of the preceding label defined in the same routine.

User Action: Relabel the line, and others as needed, so that the labels in the external routine form an increasing sequence.

(BC 4150) No labels found in INPUT file (file reference).

Severity Level: Warning

Description: No labels are found in the input file specified for the RESEQUENCE utility.

User Action: Check that the correct input file was specified. If it was, this is a BASIC program that has no need for resequencing.

(BC 4160) Line (integer) exceeds the maximum length for a BASIC source program.

Severity Level: Warning

Description: The number of characters in the resequencing input file exceeds the NOS/VE BASIC maximum line length of 255 (integer).

User Action: Split the source line into two or more valid lines.

Library Functions Index D

This appendix provides an alphabetical list of all the NOS/VE BASIC library functions. The entry for each library function includes:

- The function name.
- A brief description of the function.
- A cross-reference to the major category and subgroup of major category to which the function belongs. These are provided to help you visualize each function in context.
- A page number for the function.

Library Functions Index

Name	Description	Category/Subgroup	Page
ABS	Absolute Value	Mathematical/Number Characteristic	8-16
ACOS	Arcosine	Mathematical/Trigonometric	8-8
ASC	Character to ASCII Code	String/Conversion	12-17
ASIN	Arcsine	Mathematical/Trigonometric	8-9
ATN	Arctangent	Mathematical/Trigonometric	8-10
CDBL	Numeric to Type Real	Mathematical/Number Characteristic	8-17
CEIL	Integer Ceiling	Mathematical/Number Characteristic	8-18
CHR$	ASCII Code to Character	String/Conversion	12-19
CINT	Round to Integer	Mathematical/Number Characteristic	8-19
COS	Cosine	Mathematical/Trigonometric	8-11
COSH	Hyperbolic cosine	Mathematical/Exponential	8-2
CSNG	Numeric to Type Real	Mathematical/Number Characteristic	8-20
CVD	Interpret as Real	Files/Numeric Interp. of Strings	13-33
CVI	Intepret as Integer	Files/Numeric Interp. of Strings	13-33
CVS	Interpret as Real	Files/Numeric Interp. of Strings	13-33
DEG	Radians to Degrees	Mathematical/Trigonometric	8-12
EOF	End-of-File	File/(none)	13-13
ERL	Label of Error Line	Error Processing/(none)	6-9
ERR	Status Condition Code	Error Processing/(none)	6-11
EXP	Exponential	Mathematical/Exponential	8-3
FIX	Truncate to Integer	Mathematical/Number Characteristic	8-21
FP	Fractional Part	Mathematical/Number Characteristic	8-22
HEX$	Decimal to Hexadecimal	String/Conversion	12-20
INSTR	Search for Substring	String/Substring Manipulation	12-9
INT	Integer Floor	Mathematical/Number Characteristic	8-23
LBOUND	Dimension Lower Bound	Array/(none)	11-8
LCASE$	Convert to Lowercase	String/Miscellaneous	12-23

Name	Description	Category/Subgroup	Page
LEFT$	Left Substring	String/Substring Manipulation	12-12
LEN	String Length	String/Substring Manipulation	12-8
LOC	Current Record Number	File/(none)	13-12
LOG	Natural Logarithm	Mathematical/Exponential	8-4
MAX	Maximum	Mathematical/Miscellaneous	8-25
MID$	Middle Substring	String/Conversion	12-13
MIN	Minimum	Mathematical/Miscellaneous	8-26
MKD$	Interpret as String	Files/Numeric Interp. of Strings	13-35
MKI$	Interpret as String	Files/Numeric Interp. of Strings	13-35
MKS$	Interpret as String	Files/Numeric Interp. of Strings	13-35
OCT$	Decimal to Octal	String/Conversion	12-21
PARAMS$	Parameter String	String/Miscellaneous	12-27
RAD	Degrees to Radians	Mathematical/Trigonometric	8-13
RIGHT$	Right Substring	String/Substring Manipulation	12-15
RND	Random Number	Mathematical/Miscellaneous	8-27
SGN	Sign	Mathematical/Number Characteristic	8-24
SIN	Sine	Mathematical/Trigonometric	8-14
SINH	Hyperbolic Sine	Mathematical/Exponential	8-6
SPACE$	String of Spaces	String/Miscellaneous	12-25
SQR	Square Root	Mathematical/Miscellaneous	8-28
STR$	String Value of Numeric	String/Conversion	12-22
STRING$	String/Uniform	String/Miscellaneous	12-26
TAN	Tangent	Mathematical/Trigonometric	8-15
TANH	Hyperbolic Tangent	Mathematical/Exponential	8-7
UBOUND	Dimension Upper Bound	Array/(none)	11-8
UCASE$	Convert to Uppercase	String/Miscellaneous	12-24
VAL	Numeric Value of String	String/Conversion	12-18

Introduction to Debug E

Debug is an SCL command utility that lets you debug a program during execution. Using Debug, you can stop execution at selected points, display the values of selected variables, and resume execution.

Debug is easy to use. It requires no modification of your source code and no knowledge of assembly language. You can reference variables by their symbolic names rather than their addresses in memory. Furthermore, you don't need to interpret memory dumps, insert PRINT statements into your program, or use a load map.

Debug can be used in line mode or screen mode. Also, you can use Debug to perform machine-level debugging as well as symbolic debugging. This discussion focuses on using screen mode Debug for symbolic debugging. For information about line mode Debug, machine-level debugging, and other Debug features, see the Debug Usage manual.

Screen mode Debug gives you all of the Debug features with the ease of use of a full screen interface. You can execute Debug commands by pressing function keys rather than typing commands. Online HELP enables you to learn screen mode Debug as you use it.

Using screen mode Debug, you can:

- View your source code as it executes (an arrow points to the next line to be executed).
- Change the values of program variables while execution is suspended.
- Change the location where execution of your program resumes.
- View the program units of your program.

Getting Started

Using Debug in screen mode requires that your terminal support full screen operation. If your terminal is not set up for full screen operation, see the NOS/VE System Usage manual for terminal definitions that support the full screen interface.

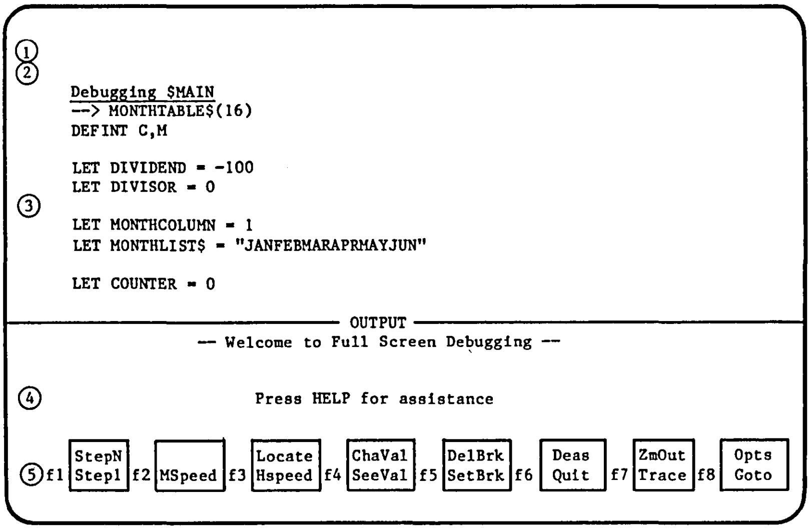
To use the symbolic capabilities of Debug, you must execute your compiled BASIC program with Debug mode turned on. Furthermore, to enter Debug in screen mode, you must enter the command:

```
CHANGE_INTERACTION_STYLE STYLE=SCREEN
```

For example, the following commands compile a BASIC program contained in permanent file $USER.EXAMPLE_BAS, specify screen mode, and execute the BASIC program with Debug mode turned on:

```
/basic input=$user.example_bas binary=lgo
/change_interaction_style style=screen
/execute_task file=lgo debug_mode=on
```

The source module of EXAMPLE_BAS is displayed in the following screen format on a Viking 721 terminal (on other terminals, the screen format may vary slightly).



```
①
②
   Debugging $MAIN
   --> MONTHTABLE$(16)
   DEFINT C,M

   LET DIVIDEND = -100
   LET DIVISOR = 0
③
   LET MONTHCOLUMN = 1
   LET MONTHLIST$ = "JANFEBMARAPRMAYJUN"

   LET COUNTER = 0
───────────────────────────── OUTPUT ─────────────────────────────
                 -- Welcome to Full Screen Debugging --

④                         Press HELP for assistance

⑤f1 StepN  f2         f3 Locate f4 ChaVal f5 DelBrk f6 Deas f7 ZmOut f8 Opts
    Stepl     MSpeed     Hspeed    SeeVal    SetBrk    Quit    Trace    Goto
```

①	Home line	The line on which you enter Debug commands and SCL commands.
②	Response line	The line on which short responses and advisory messages from Debug are displayed.
③	Source window	The area in which the program you are debugging is displayed.
④	Output window	The area in which the output generated by your program (or output delivered by Debug) is displayed.
⑤	Row of function key assignments	The Debug functions assigned to function keys. Also, you can enter Debug commands on the home line.

How to Get Help

There are two ways to get help information while using screen mode Debug:

1. The HELP key.

   Pressing the HELP key displays the Help window. The Help window overlays a portion of your screen and prompts you to enter the function for which you need help. If you press a function key, a short description of the function you select is displayed in the Help window. To exit HELP, press RETURN. Upon exiting HELP, your screen is restored to its original contents.

2. The HELP command.

   You can request help by entering the HELP command on the home line. This command is used to read an online manual while you are debugging your program. To leave the online manual, press QUIT. When you leave the online manual, the screen is restored to its contents before you entered HELP. For example, if you need information about BASIC substrings, press the HOME key and type the following HELP command on the home line:

   ```
   help s=substring m=basic
   ```

   This command takes you to the BASIC online manual for an explanation of BASIC substrings. To return to screen mode Debug, press QUIT.

   If you need information about the Debug utility, press the HOME key and type the HELP command without specifying any parameters:

   ```
   help
   ```

   Typing this command displays the beginning of the Debug online manual. To return to screen mode Debug, press QUIT.

   See the NOS/VE System Usage manual for more information about HELP.

Example

This example demonstrates some commonly used Debug functions. It is represented as a series of steps. To get the most benefit from this example, you should create the sample program, EXAMPLE_BAS, illustrated in figure E-1, then perform each step.

EXAMPLE_BAS is divided into the following test cases:

TEST1	A loop that increments a counter and then calls a subprogram to square and display the count. TEST1 demonstrates the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1, and STEPN functions.
TEST2	A loop that builds a 6-row table of 3-character strings. Input to the table is an 18-character list for the months JAN through JUN. TEST2 moves three characters at a time from the character list to the table and displays each entry. TEST2 shows how to step through loops, use line mode Debug commands in screen mode Debug, and how to scroll through Debug and program output data.
TEST3	A division test that results in a divide fault. TEST3 demonstrates how Debug handles execution errors.

In each test case, the application of some Debug functions is demonstrated. After you work this example, you can begin to debug your BASIC programs using screen mode Debug.

```
DIM MONTHTABLE$(16)
DEFINT C,M

LET DIVIDEND = -100
LET DIVISOR = 0

LET MONTHCOLUMN = 1
LET MONTHLIST$ = "JANFEBMARAPRMAYJUN"

LET COUNTER = 0

REM  TEST1:  Add to counter and call subroutine to square and display
REM          count.

LET COUNTER = 1
FOR COUNTER = 1 TO 10
    CALL SQUAREPROCEDURE (COUNTER)
NEXT COUNTER

REM  TEST2:  Create single column table for each month.

FOR MONTHROW = 0 TO 5
    MONTHTABLE$(MONTHROW) = MONTHLIST$(MONTHCOLUMN : MONTHCOLUMN + 2)
    PRINT "THE MONTH IS:  " MONTHTABLE$(MONTHROW)
    LET MONTHCOLUMN = MONTHCOLUMN + 3
NEXT MONTHROW

REM  TEST3:  Create divide fault.

LET QUOTIENT = DIVIDEND / DIVISOR
PRINT "ANSWER IS:  " ANSWER

END

REM  Subroutine SQUAREPROCEDURE

SUB SQUAREPROCEDURE (COUNTER)
    LET RESULT = 0
    LET RESULT = COUNTER * COUNTER
    PRINT COUNTER " TIMES" COUNTER " =" RESULT
END SUB
```

Figure E-1. Debug Example: Source File EXAMPLE_BAS

Preparing to Debug

After you create EXAMPLE_BAS, you must compile your program and prepare your Debug session for the screen mode environment. You can then execute EXAMPLE_BAS under Debug control. Do this as follows:

1. Assuming EXAMPLE_BAS is contained in permanent file $USER.EXAMPLE_BAS, prepare the screen mode environment and compile EXAMPLE_BAS by entering the following commands:

```
/change_interaction_style style=screen
/basic input=$user.example_bas binary=lgo
```

2. Execute EXAMPLE_BAS under control of Debug by entering the following command:

```
/execute_task file=lgo debug_mode=on
```

The source module of EXAMPLE_BAS is displayed in the Source window. The Debug functions are displayed at the bottom of the screen.

Display Screen Mode Commands

The functions below are used to display helpful information about the Debugging environment:

HELP -- Displays the Help window. Press a function key and a short explanation of the function's use appears in the Help window.

ZMIN -- Used to display the source listing in the Source window.

Now perform the following steps to become familiar with the Debug functions:

1. Press the HELP key. The Help window is displayed.

2. Press each function key corresponding to a function displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each function is displayed in the Help window.

3. Press RETURN. Exit HELP.

4. Press the ZMIN function key. The following message is displayed in the upper right hand corner of the screen:

```
Enter compiler input file for $MAIN
```

5. Enter the source file name:

```
example_bas
```

The source listing of EXAMPLE_BAS is displayed in the Source window. Also, some new functions are displayed at the bottom of the screen.

6. Press the HELP key. The Help window is displayed again.

7. Press each function key corresponding to the new function displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each new function is displayed in the Help window.

8. Press RETURN. Exit HELP.

Setting Breaks

It is often helpful to suspend program execution when debugging a program. The device for suspending execution of a program is called a break. In this sample session, the following functions are used to illustrate setting breaks:

FWD	-- Scrolls forward to the next screen of text.
FIRST	-- Displays the first screen of the source listing. Because FIRST is a lower priority function, it may not be assigned to a function key on terminals with only 16 function keys. Instead, FIRST is entered on the home line.
LOCATE	-- Prompts you to type in text, then searches the source listing for matching text. If a match is found, the cursor is moved to the line containing the matching text.
SETBRK	-- Sets an execution break on the line containing the cursor. The line is highlighted to show that it contains a break. Execution is suspended before the line containing the break is executed. Execution resumes with the statement on the line containing the break.

This section also uses the following item:

HOME	-- Moves the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.

Example

Perform the following steps to place three execution breaks in EXAMPLE_BAS:

1. Press the LOCATE function key. At the top right hand corner of the screen, you are prompted for the text to be located.

2. Enter the following text exactly as it appears in EXAMPLE_BAS:

   ```
   FOR MONTHROW
   ```

   The cursor is moved to the line:

   ```
   FOR MONTHROW = 0 TO 5
   ```

3. Press the SETBRK function key. A break is set and the line containing the cursor is highlighted to show that it contains an execution break.

4. Use the down-arrow key to move the cursor to the line containing:

   ```
   LET MONTHCOLUMN = MONTHCOLUMN + 3
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_BAS source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

5. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

6. Use the down-arrow key to move the cursor to the line:

   ```
   LET QUOTIENT = DIVIDEND / DIVISOR
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_BAS source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

7. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

8. Press the FIRST function key. The first screen of the EXAMPLE_BAS source listing is displayed in the Source window.

   If FIRST is not assigned to a function key, FIRST must be entered on the home line. To do this, press the HOME key. The cursor moves to the home line. Enter the following on the home line:

   ```
   first
   ```

   The first screen of the EXAMPLE_BAS source listing is displayed in the Source window.

Debugging TEST1

Using Debug, you can execute a program one line or several lines at a time. Also, you can examine a variable´s contents, change its contents, and execute code containing the variable several times. These capabilities are demonstrated in this sample session using the following functions:

CHAVAL --	Prompts you to enter a variable name and the value you want it to contain, then changes the variable´s contents to the new value.
GOTO --	Moves the execution pointer to the line that contains the cursor. Execution resumes with the statement on this line.
HSPEED --	Executes a program until a break is encountered or the program ends.
SEEVAL --	Prompts you to enter a variable name, then displays the value of the variable in the Output window.
STEP1 --	Executes a program one line at a time.
STEPN --	Executes N lines of a program, where N is an integer.

Perform the following steps to demonstrate the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1, STEPN:

1. Press the STEPN function key. In the upper right corner of the screen you are prompted for the number of lines to execute; enter:

   ```
   17
   ```

   STEPN executes 17 lines of EXAMPLE_BAS, moving the execution arrow to the statement:

   ```
   FOR COUNTER = 1 TO 10
   ```

2. Press the STEP1 function key. The FOR statement is executed; the execution arrow points to the statement:

   ```
   CALL SQUAREPROCEDURE (COUNTER)
   ```

3. Press the STEP1 function key seven times. An iteration of TEST1 is executed one line at a time. The output generated by the iteration is displayed in the Output window.

Example

4. Press the SEEVAL function key. A prompt to enter a variable name is printed in the upper right hand corner of the screen. Enter the name:

    ```
    counter
    ```

    The value of COUNTER is displayed in the Output window:

    ```
    counter = 2
    ```

    Thus, you can use SEEVAL to observe the contents of a variable.

5. Press the CHAVAL function key. A prompt for a variable name and its new value is displayed in the upper right hand corner of the screen; enter:

    ```
    counter=8
    ```

    The value of COUNTER is changed to 8.

6. Press the SEEVAL function key. When you are prompted for a variable name, enter:

    ```
    counter
    ```

    The following message is displayed in the Output window:

    ```
    counter = 8
    ```

    Thus, the change of COUNTER´s value is verified.

7. Press the STEPN function key. When you are prompted for the number of lines to execute; enter:

    ```
    7
    ```

    STEPN executes 7 lines of TEST1. The output generated by this loop iteration is displayed in the Output window.

8. Press the SEEVAL function key. When you are prompted for a variable name, enter:

    ```
    counter
    ```

    The value of COUNTER is displayed in the Output window:

    ```
    counter = 9
    ```

    Therefore, the value given to COUNTER in step 5 is used by the FOR statement.

9. Use the up-arrow key to move the cursor to the line:

   ```
   FOR COUNTER = 1 TO 10
   ```

10. Press the GOTO function key. The execution arrow moves to the line containing the cursor; execution resumes with this statement.

11. Press the HSPEED function key. Execution resumes from the FOR statement; COUNTER is initialized to 1. Execution of EXAMPLE_BAS continues until an execution break is encountered.

Debugging TEST2

After program execution is resumed in step 11 of TEST1, it stops at the break set on the PERFORM statement in TEST2. The following functions are used in TEST2 to illustrate more Debug capabilities:

BKW	-- Scrolls backward to the previous screen of text.
DELBRK	-- Deletes execution breaks.
HSPEED	-- Executes a program until a break is encountered or the program ends.

This section also uses the following items:

HOME	-- Press the HOME key to move the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.
DISPLAY_PROGRAM_VALUE	-- A line mode Debug command that displays the values of program variables.

Perform the following steps to learn how to execute loops one iteration at a time, execute line mode Debug commands, and scroll output data when using Debug:

1. Press the HSPEED function key. Execution stops at the break set on the last line of the FOR loop in TEST2; output generated by the loop is displayed in the Output window.

2. Press the HSPEED function key again. One iteration of the FOR loop is executed; execution stops at the break set at the statement, LET MONTHCOLUMN = MONTHCOLUMN + 3. Each time HSPEED is used, an iteration of the loop is performed. By using strategically placed execution breaks, as in this example, a loop can be executed one iteration at a time.

3. Press the HSPEED function key. One more loop iteration is performed.

4. Press the HOME key. The cursor moves to the home line.

5. Enter the line mode Debug command:

```
display_program_value name=$all
```

   The values of all variables in EXAMPLE_BAS are displayed in the Output window. Thus, line mode Debug commands can be used in screen mode Debug by entering them on the home line. For more information about using line mode Debug commands see the Debug Usage Manual.

6. Press the DELBRK key. The execution break is deleted. The highlight is removed from the line when the break is removed.

7. Press the down-arrow key until the cursor is inside of the Output window.

8. Press the BKW key. The data in the Output window scrolls backward. When the cursor is contained within the Output window, you can use the BKW and FWD keys to scroll backward and forward through the data in the window.

9. Press the HSPEED function key. The execution of EXAMPLE_BAS resumes, stopping when the line containing the third break is reached. The execution arrow points to the beginning of TEST3.

Debugging TEST3

After resuming execution of EXAMPLE_BAS in step 9 of section TEST2, execution stops at the begining of TEST3. In TEST3, Debug is presented with an execution error. The following functions are used in this sample session to demonstrate how Debug can be used when an exectuion error is encountered:

CHAVAL	-- Prompts you to enter a variable name and the value you want it to contain, then changes the variable´s contents to the new value.
GOTO	-- Moves the execution pointer to the line that contains the cursor. Execution resumes with the statement on this line.
SEEVAL	-- Prompts you to enter a variable name, then displays the value of the variable in the Output window.
STEP1	-- Executes a program one line at a time.
QUIT	-- Used to exit Debug.

Perform the following steps to finish the example:

1. Press the STEP1 function key. The DIVISION statement is executed, execution of EXAMPLE_BAS halts, and the following message flashes in the upper right hand corner of the screen:

   ```
   divide_fault
   ```

2. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   divisor
   ```

   The following message is displayed in the Output window:

   ```
   divisor = 0.
   ```

   A division by zero caused the execution error.

3. Press the CHAVAL function key. When you are prompted, enter:

   ```
   divisor=1.0
   ```

   The value of DIVISOR is changed to 1.

4. Press the SEEVAL function key. When you are prompted, enter:

   ```
   divisor
   ```

   The following text is displayed in the Output window:

   ```
   divisor = 1.0000E+0000
   ```

   The change to DIVISOR is verified.

5. Press the GOTO function key. The execution arrow points at the DIVISION statement and program execution resumes with this statement.

6. Press the STEP1 function key. The DIVISION statement is executed. Therefore, the GOTO and CHAVAL functions can be used in concert to recover from execution errors. However, to correct execution errors permanently, you must exit Debug, edit the program, and recompile it.

7. Press the STEP1 function key again. The result of the DIVISION statement is displayed in the Output window.

8. Press the STEP1 function key two times. EXAMPLE_BAS ends and the following message is displayed in the Output window:

   ```
   -- DEBUG: The status at termination was: NORMAL.
   ```

9. Press the QUIT function key. Exit Debug.

Now that you have concluded this example, you should be able to begin using screen mode Debug to debug your BASIC programs. For more information about screen mode Debug and line mode Debug commands, see the Debug Usage manual.

Index

Index

BASIC for NOS/VE Usage 60486313 E

e would like your comments on this manual. While writing it, we made some assumptions about who ould use it and how it would be used. Your comments will help us improve this manual. Please ake a few minutes to reply.

ho Are You?	How Do You Use This Manual?	Do You Also Have?
_ Manager	__ As an Overview	__ BASIC for NOS/VE Summary
_ Systems Analyst or Programmer	__ To Learn the Product/System	
_ Applications Programmer	__ For Comprehensive Reference	
_ Operator	__ For Quick Look-up	
_ Other ______		

hat programming languages do you use? ______

hich are helpful to you? __ Quick Index (inside back cover) __ Character Set (App. B)
__ Related Manuals figure
__ Other: ______

ow Do You Like This Manual? Check those that apply.

Yes	Somewhat	No	
__	__	__	Is the manual easy to read (print size, page layout, and so on)?
__	__	__	Is it easy to understand?
__	__	__	Is the order of topics logical?
__	__	__	Are there enough examples?
__	__	__	Are the examples helpful? (__ Too simple __ Too complex)
__	__	__	Is the technical information accurate?
__	__	__	Can you easily find what you want?
__	__	__	Do the illustrations help you?
__	__	__	Does the manual tell you what you need to know about the topic?

omments? If applicable, note page number and paragraph.

Continue on other side

'ould you like a reply? __ Yes __ No

'rom:

lame ______ Company ______

ddress ______ Date ______

______ Phone No. ______

______

'lease send program listing and output if applicable to your comment.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MN.

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA

Technology and Publications Division
Mail Stop: SVL104
P.O. Box 3492
Sunnyvale, California 94088-3492

FOLD FOLD

Comments (continued from other side)

Quick Index

The index beginning on the following page lists the BASIC statements and functions described in this manual and the page on which each is described.

CONTROL DATA