BASIC–PLUS–2

Reference Manual

Order Number: AA–JP30B–TK

May 1991

This manual provides reference information and examples on all BASIC–PLUS–2 commands, directives, statements, and functions.

Revision/Update Information:	This manual is a revision.
Operating System and Version:	RSX–11M Version 4.6 or higher RSX–11M–PLUS Version 4.3 or higher Micro/RSX Version 4.3 or higher RSTS/E Version 9.7 or higher
Software Version:	BASIC–PLUS–2 Version 2.7

Digital Equipment Corporation
Maynard, Massachusetts

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

This document was prepared using VAX DOCUMENT, Version 2.0.

Contents

2 Environment Commands

3 Compiler Directives

4 Statements and Functions

A BASIC–PLUS–2 Keywords

B Debugger Commands

C Editing Mode Commands

Figures

Tables

Preface

Intended Audience

This manual provides detailed reference information on all BASIC–PLUS–2 commands, directives, statements, and functions. Readers are presumed to have some previous knowledge of BASIC or another high-level programming language. This manual should be used with the *BASIC–PLUS–2 User's Guide*.

Operating Systems and Versions

BASIC–PLUS–2 runs on the following operating systems and versions:

RSX–11M Version 4.6 or higher
RSX–11M–PLUS Version 4.3 or higher
Micro/RSX Version 4.3 or higher
RSTS/E Version 9.7 or higher

Associated Documents

This manual is one of two manuals that form the BASIC–PLUS–2 documentation set. The other manual in the documentation set, the *BASIC–PLUS–2 User's Guide*, contains tutorial material on developing BASIC–PLUS–2 programs; it describes BASIC–PLUS–2 programming concepts and provides information on advanced programming techniques, including program optimization. If you are unfamiliar with a topic, you may want to read the information in the *BASIC–PLUS–2 User's Guide* before consulting this manual.

If you are not an experienced BASIC programmer, you should read the following manuals before using the BASIC–PLUS–2 document set:

- *Introduction to BASIC*
- *BASIC for Beginners*
- *More BASIC for Beginners*

Structure of This Document

This manual consists of four chapters and four appendixes.

Chapter 1	Summarizes BASIC–PLUS–2 program elements and structure
Chapter 2	Describes BASIC–PLUS–2 environment commands
Chapter 3	Describes BASIC–PLUS–2 compiler directives
Chapter 4	Describes BASIC–PLUS–2 statements and functions
Appendix A	Lists the BASIC–PLUS–2 keywords
Appendix B	Describes BASIC–PLUS–2 debugger commands
Appendix C	Describes BASIC–PLUS–2 editing mode commands
Appendix D	Lists and describes the Object Time System (OTS) routines

Chapters 2, 3, and 4, and Appendixes B and C provide reference material on each BASIC–PLUS–2 language element. The language elements are arranged in alphabetical order within each chapter or appendix, and each language element begins on a separate page. The language element descriptions contain the following information:

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

Please use the Reader's Comment form in the back of this book to report documentation errors, to comment on how information is presented, or to provide suggestions for future publications.

Conventions Used in This Document

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

Convention	Meaning
`Color`	Color in code examples denotes user input.
UPPERCASE letters	Uppercase letters in language syntax denote BASIC–PLUS–2 keywords and must be spelled exactly as shown; you can enter them in either upper or lower case in actual coding.
lowercase letters	Lowercase letters in language syntax denote mnemonics representing user-supplied names or characters.
BOLD type	Bold type is used to denote a term (pertaining to the software) that is being mentioned, defined, or explained for the first time in the manual.
Italic type	Italic type is used to refer to the generic or mnemonic terms that appear in syntax diagrams.
[]	Brackets enclose an optional portion of a format. Brackets around vertically stacked items indicate that you can select one of the enclosed items. You must include all punctuation as it appears in the brackets.
{}	Braces enclose a mandatory portion of a format. Braces around vertically stacked items indicate that you must choose one of the enclosed items. You must include all punctuation as it appears in the braces.
. . .	A vertical ellipsis indicates that code, which would normally be present, is not shown.
. . .	An ellipsis indicates that the immediately preceding item can be repeated. An ellipsis following a format unit enclosed in brackets or braces means that you can repeat the entire unit. If repeated items or format units must be separated by commas, the ellipsis is preceded by a comma (, . . .).

The following mnemonics are used in the syntax diagrams:

Mnemonic	Meaning
angle	An angle in radians
array	An array; syntax rules specify whether the bounds or dimensions can be specified
chnl-exp	An I/O channel associated with a file
com	Specific to a COMMON block
cond	Conditional expression; indicates that an expression can be either logical or relational

Mnemonic	Meaning
const	A constant value
data-type	A data type keyword
def	Specific to a DEF function
exp	An expression
file-spec	A file specification
func	Specific to a FUNCTION subprogram
int	An integer value
int-exp	An expression that represents an integer value
int-var	A variable that contains an integer value
label	An alphanumeric statement label
lex	Lexical; used to indicate a component of a compiler directive
line	A statement line; may or may not be numbered
line-num	A statement line number
lit	A literal value, in quotation marks
log-exp	Logical expression
map	Specific to a MAP statement
matrix	A two-dimensional array
name	A name or identifier; indicates the declaration of a name or the name of a BASIC–PLUS–2 structure, such as a SUB subprogram
num	A numeric value
param-list	A parameter list, such as for a SUB subprogram
pass-mech	A valid BASIC–PLUS–2 passing mechanism
real	A floating-point value
rel-exp	Relational expression
str	A character string
str-exp	An expression that represents a character string
str-var	A variable that contains a character string
sub	Specific to a SUB subprogram
target	The target point of a branch statement; either a line number or a label
unsubs-var	Unsubscripted variable, as opposed to an array element
var	A variable

Summary of Technical Changes

The following is a list of the major changes for Version 2.7 of BASIC–PLUS–2:

- The BASIC–PLUS–2 compiler can now run in Instruction and Data (I & D) memory space, thereby improving compilation performance. I & D support is selected when the compiler is built and installed; it does not impact the language syntax.
- The DCL compilation command BASIC has been expanded to allow you to compile multiple BASIC–PLUS–2 programs from DCL.
- A new qualifier, /[NO]BOUNDS, has been added to the environment COMPILE command and to the DCL command BASIC. If you specify /NOBOUNDS, the processing overhead of checking array boundaries on arrays of one or two dimensions is eliminated.
- Command line support has been expanded to allow longer command strings than was previously allowed.
- Several performance enhancements have been incorporated to allow faster compilation and run time execution. These enhancements are primarily internal and are not reflected in the language syntax; therefore, they do not require documentation in the user manuals.

1

Program Elements and Structure

The building blocks of a BASIC–PLUS–2 program are as follows:

- Program lines and their components
- The BASIC–PLUS–2 character set
- BASIC–PLUS–2 data types
- Variables and constants
- Expressions
- Program documentation

These building blocks are described in the following sections.

1.1 Components of Program Lines

A BASIC–PLUS–2 program is a series of program lines. Each program line can contain any or all of the following information:

- Line numbers or labels
- Statements and functions
- Compiler directives
- Comments
- A line terminator (carriage return)

These components are discussed in the following sections.

1.1.1 Line Numbers

Every BASIC–PLUS–2 statement must be associated with a line number. Thus, the first element in a BASIC–PLUS–2 program must be a line number. A line number must be an integer from 1 through 32767, followed by a space or a tab, and must be unique to that line. BASIC–PLUS–2 ignores leading spaces, tabs, and zeros in line numbers. Embedded spaces, tabs, and commas within line numbers cause BASIC–PLUS–2 to signal an error.

A line number followed by a carriage return does not constitute a BASIC–PLUS–2 program line. A numbered program line must contain a statement or a comment field. Comment fields are discussed in Section 1.1.5.1. A new line number terminates a BASIC–PLUS–2 program line.

A program line can contain any number of text lines; however, a text line cannot contain more than 255 characters on RSTS/E systems, or more than 132 characters on RSX systems.

BASIC–PLUS–2 uses line numbers to do the following:

- Indicate the order of statement execution
- Identify control points for branching
- Help in debugging and updating programs
- Find the location of run-time errors
- Resume processing after an error has been handled

1.1.2 Labels

A label is a 1- to 31-character name that identifies a statement or block of statements. It may immediately follow a line number. The label logically identifies a statement or block of statements. The label name must begin with a letter; the remaining characters, if any, can be any combination of letters, digits, dollar signs ($), underscores (_), or periods (.).

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

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

The colon is not part of the label name. It informs BASIC–PLUS–2 that the label is being defined rather than referenced. Consequently, the colon is not allowed when you use a label to reference a statement. For example:

```
200     GOTO Yes_routine
```

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

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

Example

```
100 IF A% = B%
    THEN 1000
    ELSE 1050

200 IF A$ = "YES"
    THEN GOTO Yes
    ELSE GOTO No
```

1.1.3 Statements

A BASIC–PLUS–2 statement consists of a statement keyword and optional operators and operands. For example, both of these statements are valid:

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

BASIC–PLUS–2 statements are either executable or nonexecutable:

- Executable statements perform operations (for example, PRINT, GOTO, and READ).
- Nonexecutable statements provide information to the compiler (such as characteristics and arrangement of data) and provide comments in the source program (for example, DATA, DECLARE, and REM).

BASIC–PLUS–2 can accept and process one statement on a line of text, several statements on a line of text, multiple statements on multiple lines of text, and single statements continued over several lines of text. Each line of program text is associated with the most recently specified line number.

1.1.3.1 Keywords

Every BASIC–PLUS–2 statement except LET and empty statements must begin with a keyword. A keyword is a reserved element of the BASIC–PLUS–2 language. Keywords are used to do the following:

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

Note

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

Keywords cannot be used in any context other than as BASIC–PLUS–2 keywords. The assignment STRING$ = "YES", for example, is invalid because STRING$ is a reserved BASIC–PLUS–2 keyword and therefore cannot be used as a variable. Appendix A in this manual contains a list of BASIC–PLUS–2 reserved keywords.

A BASIC–PLUS–2 keyword cannot be split across lines of text. There must be a space, tab, or special character such as a comma between the keyword and any other variable or operator.

In general, keywords cannot include embedded spaces, but there are exceptions in which spaces are optional or required. The exceptions are shown in Table 1–1.

Table 1–1 Keyword Space Requirements

Optional Space	Required Space	No Space
GO TO	BY DESC	FNEND
GO SUB	BY REF	FNEXIT
ON ERROR	BY VALUE	FUNCTIONEND
	END DEF	FUNCTIONEXIT
	END FUNCTION	NOECHO

(continued on next page)

Table 1–1 (Cont.) Keyword Space Requirements

Optional Space	Required Space	No Space
	END IF	SUBEND
	END SELECT	
	END SUB	
	EXIT DEF	
	EXIT FUNCTION	
	EXIT SUB	
	INPUT LINE	
	MAP DYNAMIC	
	MAT INPUT	
	MAT LINPUT	
	MAT PRINT	
	MAT READ	

1.1.3.2 Single-Statement Lines and Continued Statements

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

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

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

You can have a single statement span multiple text lines by entering an ampersand (&) and the Return key. BASIC–PLUS–2 ignores spaces or tabs between the ampersand and the carriage return. For example:

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

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

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

You can continue any BASIC–PLUS–2 statement, but you cannot continue a string literal or BASIC–PLUS–2 keyword. The following example generates the error message "Unterminated string literal".

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

This example is valid, although it prints two distinct strings:

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

You can join two string literals by using the string concatenation operator (+):

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

BASIC–PLUS–2 concatenates the two string literals at compilation and stores them as one string. When the PRINT statement executes, BASIC–PLUS–2 displays the one concatenated string literal rather than two separate string literals, thereby causing your program to execute faster and more efficiently.

Continued statements do not have program line numbers of their own, although the compiler counts and numbers them as sublines.

1.1.3.3 Multi-Statement Lines

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

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

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

Because all statements are on the same program line, any reference to line number 40 refers to all three statements. Execution begins with the first statement on the line. BASIC–PLUS–2 cannot execute the second statement without executing the first statement.

You can also write a multi-statement program line that associates all statements with a single line number by placing each statement on a separate line. This format requires only a space or tab at the beginning of each new line of text. BASIC–PLUS–2 assumes that such an unnumbered line of text is either a new statement or a clause in an IF-THEN-ELSE construct.

In the following example, each line of text contains a BASIC–PLUS–2 statement associated with program line number 400.

Example

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

BASIC–PLUS–2 also recognizes IF . . . THEN . . . ELSE constructs segmented over several lines of text and associates the THEN and ELSE keywords with the preceding IF statement. For example:

Example

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

The BASIC–PLUS–2 compiler assigns listing line numbers to text lines as they occur in the program. Blank text lines are assigned listing line numbers. Note the difference between program line numbers (which you create in your source program) and listing line numbers (which are assigned by the compiler).

Example

```
00001 100        REM    Determine if the user's response
00002                   was YES or NO.
00003 200        IF (A$ = "YES") OR (A$ = "Y")
00004            THEN PRINT "You typed YES"
00005            ELSE PRINT "You typed NO"
00006                 STOP
00007            END IF
```

You cannot use listing line numbers as targets of branch statements. The target of a branch statement such as GOTO must be a program line number or a label. See the *BASIC–PLUS–2 User's Guide* for more information on listing file formats.

You can use any BASIC–PLUS–2 statement in a multi-statement line; however, a REM statement in a multi-statement line must be the last statement on that line because the compiler ignores all text following a REM keyword until it reaches a new program line number.

A DATA statement included on a multi-statement line must be the last statement, because the compiler treats all text following a DATA statement as data until it reaches a new program line.

Because a leading space or tab not followed by a line number implies a new statement in a multi-statement line, compiler directives and immediate mode statements cannot be preceded by a space or tab.

Compiler directives and immediate-mode statements cannot appear between two text lines of a continued statement.

1.1.4 Compiler Directives

Compiler directives instruct the BASIC–PLUS–2 compiler to perform certain operations during program compilation.

By including compiler directives in a program, you can:

- Place program titles and subtitles in the header that appears on each page of the listing file
- Place a program version identification string in both the listing file and object module
- Start or stop the inclusion of listing information for selected parts of a program
- Start or stop the inclusion of cross reference information for selected parts of a program
- Include BASIC–PLUS–2 code from another source file
- Conditionally compile parts of a program
- Terminate compilation
- Display messages during the compilation

Follow these rules when using compiler directives:

- Compiler directives must begin with a percent sign.
- Compiler directives must be the only text on the line (except for %IF-%THEN-%ELSE-%END-%IF).
- Compiler directives cannot appear within a quoted string.
- Compiler directives must be preceded by a space, tab, or line number.

See the *BASIC–PLUS–2 User's Guide* and Chapter 3 in this manual for more information on compiler directives.

1.1.5 Comments

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

- Comment fields (including empty statements)

- REM statements

1.1.5.1 Comment Field

A comment field begins with an exclamation point (!) and ends with a carriage return. You supply text after the exclamation point to document your program. BASIC–PLUS–2 does not execute text in a comment field. For example:

Example

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

Here, BASIC–PLUS–2 executes only the FOR . . . NEXT loop. The comment fields, preceded by exclamation points, are not executed.

Comment fields help make your program more readable and allow you to format your program into readily visible logical blocks. They can also serve as target lines for GOTO and GOSUB statements:

Example

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

You can also use an exclamation point to terminate a comment field, but this practice is not recommended. You should make sure that there are no exclamation points in the intended comment field itself; if there is, BASIC–PLUS–2 treats the text remaining on the line as source code.

Note

Comment fields in DATA statements are invalid; the BASIC–PLUS–2 compiler treats the exclamation point and following text as data.

An empty statement consists of a line number and an exclamation point. Empty statements can make your program more legible by adding "white space" and visually separating logical program segments. Note that you can also include blank lines with neither a line number nor an exclamation point to make your program more legible.

In the following example, lines 100 and 300 are empty statements.

Example

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

1.1.5.2 REM Statements

A REM statement begins with the REM keyword and ends when BASIC–PLUS–2 encounters a new line number. The text you supply between the REM keyword and the next line number documents your program. Like comment fields, REM statements do not affect program execution. BASIC–PLUS–2 ignores all characters between the keyword REM and the next line number. Therefore, the REM statement can be continued without the ampersand (&) continuation character and should be the only statement on the line or the last of several statements in a multi-statement line:

Example

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

The REM statement is nonexecutable. When you transfer control to a REM statement, BASIC–PLUS–2 executes the next executable statement that lexically follows the referenced statement.

Note

The REM statement is supported primarily for compatibility with programs that were originally written for other BASIC language compilers. Because BASIC–PLUS–2 treats all text between the REM statement and the next line number as commentary, REM should be

avoided in programs that follow the implied continuation rules. It is recommended that you use comment fields instead.

In the following example, the conditional GOTO statement in line 20 transfers program control to line 10. BASIC–PLUS–2 ignores the REM comment on line 10 and continues program execution at line 20.

Example

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

1.1.6 Line Terminators

In the BASIC environment, a program line ends with a carriage return/line feed combination (the Return key) followed by an optional space or tab and a new line number. An ampersand (&) followed by a carriage return ends a line of text, but not the program line. All statements between the first line number and the next line number are associated with the first line number.

1.1.7 Lexical Order

Lexical order refers to the order in which the statements in a program are compiled. In general terms, BASIC–PLUS–2 compiles program lines in sequential order: multiple statements on a line of text are processed from left to right, and lines of text are processed from top to bottom.

Some BASIC–PLUS–2 statements, such as comments and MAP declarations, are nonexecutable. If program control passes to a nonexecutable statement, the BASIC–PLUS–2 compiler executes the first statement that lexically follows the nonexecutable statement.

1.2 BASIC–PLUS–2 Character Set

BASIC–PLUS–2 uses the full ASCII character set to define alphanumeric and special characters that are used in string variables. This includes the following:

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

See the *BASIC–PLUS–2 User's Guide* for the full ASCII character set and character values.

The BASIC–PLUS–2 compiler does not distinguish between upper- and lowercase letters except in string literals or within a DATA statement. The BASIC–PLUS–2 compiler does not process characters in REM statements or comment fields, nor does it process nonprinting characters unless they are part of a string literal.

In string literals, BASIC–PLUS–2 processes the following:

- Lowercase letters as lowercase
- Nonprinting characters

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

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

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

See Section 1.5.4 for more information on explicit literal notation.

1.3 BASIC–PLUS–2 Data Types

Each unit of data in a BASIC–PLUS–2 program has a specific data type that determines how that unit of data is to be interpreted and manipulated by the BASIC–PLUS–2 compiler. This data type also determines how many storage bits make up the unit of data.

BASIC–PLUS–2 recognizes four primary data types:

- Integer
- Floating-point
- Character string
- Record File Address (RFA)

Integer data is stored as binary values in a byte, word, or longword. These storage formats correspond to the BASIC–PLUS–2 data type keywords BYTE, WORD, and LONG; these are all subtypes of the type INTEGER.

Floating-point values are stored using a signed exponent and a binary fraction. BASIC–PLUS–2 allows only single and double floating-point formats. These storage formats correspond to the BASIC–PLUS–2 data-type keywords SINGLE and DOUBLE; these are both subtypes of the type REAL.

Character data consists of strings of bytes containing ASCII codes. The first character in the string is stored in the first byte, the second character is stored in the second byte, and so on. BASIC–PLUS–2 allows up to 32767 characters for a STRING data element.

In addition to this data type, BASIC–PLUS–2 also recognizes a special RFA data type to provide information about a record's file address. An RFA uniquely specifies a record in a file: you can access RMS files of any organization by the record's file address. By specifying the disk address of a record, RMS retrieves the record at that address. Accessing records by RFA is more efficient and faster than other forms of random record access. The RFA data type can only be used for the following:

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

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

The RFA data type requires six bytes of information. See the *BASIC–PLUS–2 User's Guide* for more information on Record File Addresses and the RFA data type.

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

Table 1–2 BASIC–PLUS–2 Data Types

Data Type Keyword	Size	Range	Precision (Decimal Digits)
		Integer	
BYTE	8 bits	-128 to +127	NA
WORD	16 bits	-32768 to +32767	NA
LONG	32 bits	-2147483648 to +2147483647	NA
		Real	
SINGLE	32 bits	$.29 * 10^{-38}$ to $1.7 * 10^{38}$	6
DOUBLE	64 bits	$.29 * 10^{-38}$ to $1.7 * 10^{38}$	16
		String	
STRING	One character per byte	Max = 32767	NA
		RFA	
RFA	6 bytes	NA	NA

In Table 1–2, REAL and INTEGER are generic data type keywords that specify floating-point and integer storage, respectively. If you use the REAL or INTEGER keywords to type data, the actual data type (SINGLE, DOUBLE, BYTE, WORD, or LONG) depends on the current default. If you do not explicitly type one of the appropriate subtypes, BASIC–PLUS–2 uses the current subtype defaults for REAL and INTEGER.

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

1.3.1 Implicit Data Typing

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

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

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

- BYTE, WORD, or LONG for INTEGER values
- SINGLE, or DOUBLE for REAL values

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

- The system default (REAL)
- The default data type set by the BASIC command at DCL level
- The data type set for the BASIC environment with the SET or COMPILE compiler command
- The data type set for the BASIC environment with the BASIC statement OPTION

The BASIC–PLUS–2 qualifiers for the SET and COMPILE commands are described in Chapter 2 of this manual.

Note that if you compile your program with the /TYPE_DEFAULT=EXPLICIT qualifier, you can still add the appropriate suffixes to your variable names or constant values. The suffixes are useful because they identify the data type of the variable or constant immediately; the reader does not have to refer to the declarations at the top of the program to see which data type applies to a particular program value. However, with the /TYPE_DEFAULT=EXPLICIT qualifier you must still explicitly assign data types to all program values, or BASIC–PLUS–2 signals an error.

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

1.3.2 Explicit Data Typing

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

In the following example, the first DECLARE statement associates the constant value 03060 and the STRING data type with a constant named *zip_code*. The second DECLARE statement associates the STRING data type with *emp_name*, the DOUBLE data type with *with_tax*, and the SINGLE data type with *int_rate*. No constant values are associated with identifiers in the second DECLARE statement because the keyword CONSTANT does not appear; the identifiers are variable names.

Example

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

With explicit data typing, each program variable within a program can have a different range and precision. You can explicitly assign data types to variables, constants, arrays, parameters, and functions; otherwise, compilation defaults apply.

Program values that you explicitly type as INTEGER will be compiled as BYTE, WORD, or LONG, depending on the default set by compilation qualifiers, the COMPILE or SET commands, or with the OPTIONS statement. Similarly, program values that you explicitly type as REAL are compiled as SINGLE or DOUBLE.

The /TYPE_DEFAULT=EXPLICIT qualifier or OPTION TYPE=EXPLICIT statement allows you to specify that all program data must be explicitly typed. Compiling a program with /TYPE_DEFAULT= EXPLICIT or specifying OPTION TYPE=EXPLICIT means that any program value not explicitly declared causes BASIC–PLUS–2 to signal an error.

It is recommended that you use the explicit data typing features of BASIC–PLUS–2 for new applications. See the *BASIC–PLUS–2 User's Guide* for more information.

1.4 Variables

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

Depending on the program operations specified, the value of a variable can change from statement to statement. BASIC–PLUS–2 uses the most recently assigned value when performing calculations. This value remains in effect until a new value is assigned to the variable.

BASIC–PLUS–2 accepts these general types of variables:

- Floating-point
- Integer
- String
- RFA

See the *BASIC–PLUS–2 User's Guide* for more information on RFA variables.

1.4.1 Variable Names

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

Variable names must conform to the following rules:

- The first character of all variable names must be an upper- or lowercase alphabetic character (A through Z).
- Variable names cannot contain embedded spaces.
- Reserved BASIC–PLUS–2 keywords are not allowed as variable names.
- Internal variable names must conform to the following rules:
  - Contain 1 to 31 characters
  - Each character following the first can be an upper- or lower-case letter (A through Z), a digit (0 through 9), a dollar sign ($), underscore (_), or period (.)
  - The last character can be a dollar sign ($) to denote a string variable or a percent sign to denote an integer variable
- External variable names must conform to the following rules:
  - Be explicitly declared
  - Contain 1 to 6 characters
  - Each character following the first can be an upper- or lower-case letter (A through Z), a digit (0 through 9), a dollar sign ($), or period (.)

1.4.2 Implicitly Declared Variables

BASIC–PLUS–2 accepts three types of implicitly declared variables:

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

The name of an implicitly declared variable defines its data type. Integer variables end with a percent sign (%), string variables end with a dollar sign ($), and variables of the default type (usually floating-point) end with any allowable character except a percent sign or dollar sign. All three types of variables must conform to the rules listed in Section 1.4.1 for naming variables. The current data type default (INTEGER, or REAL) determines the data type of implicitly declared variables that do not end in a percent sign or dollar sign.

A **floating-point variable** is a named location that stores a single-precision floating-point value. The current default size for floating-point numbers (SINGLE or DOUBLE) determines the data type of the floating-point variable. The following are valid floating-point variable names:

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

If a numeric value of a different data type is assigned to a floating-point variable, BASIC–PLUS–2 converts the value to a floating-point number.

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

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

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

If you assign a floating-point value to an integer variable, BASIC–PLUS–2 truncates the fractional portion of the value. It does not round to the nearest integer. For example:

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

BASIC–PLUS–2 assigns the value –5, not –6, to the integer variable.

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

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

Strings have both value and length. BASIC–PLUS–2 sets all string variables to a default length of zero before program execution begins, with the exception of those variables in a COMMON, MAP, or virtual array. See the COMMON statement and the MAP statement in Chapter 4 of this manual for information on string length in COMMON and MAP areas. See the *BASIC–PLUS–2 User's Guide* for information on default string length in virtual arrays.

During execution, the length of a character string associated with a string variable can vary from zero (signifying a null or empty string) to 32767 characters.

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

1.4.3 Explicitly Declared Variables

BASIC–PLUS–2 lets you explicitly assign a data type to a variable or an array. For example:

```
100  DECLARE DOUBLE Interest_rate
```

Data type keywords are described in Section 1.1.3.1. For more information on explicit declaration of variables, see the sections on the COMMON, DECLARE, DIMENSION, DEF, FUNCTION, EXTERNAL, MAP, and SUB statements in Chapter 4 of this manual. See also the *BASIC–PLUS–2 User's Guide*.

1.4.4 Subscripted Variables and Arrays

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

In the following example, the DECLARE statement sets the bounds of the array *emp_name* to 1000. Therefore, the maximum value for an *emp_name* subscript is 1000. The bounds of the array define the maximum value for a subscript of that array.

Example

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

Subscripts can be any positive WORD integer value from 0 through 32767.

Note

By default, BASIC–PLUS–2 signals an error if a subscript is larger than the allowable range. Note, however, that the amount of storage the system can allocate depends on available memory. Therefore, very large arrays may cause an internal allocation error even though the subscript is still within the specified range.

An **array** is a set of data ordered in any number of dimensions. A one-dimensional array, like *emp_name*(1000), is called a **list** or **vector**. A two-dimensional array, like *payroll_data*(5,5), is called a **matrix**. An array of more than two dimensions, like *big_array*(15,9,2), is called a **tensor**.

BASIC–PLUS–2 arrays are always zero based. The number of elements in any dimension always includes element number zero. For example, the array *emp_name* contains 1001 elements, since BASIC–PLUS–2 allocates element zero. *Payroll_data*(5,5) contains 36 elements because BASIC–PLUS–2 allocates row and column zero. For all arrays except virtual arrays, the total number of array elements cannot exceed 32767.

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

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

```
A(6%)

B (6%)

C$ (6%)
```

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

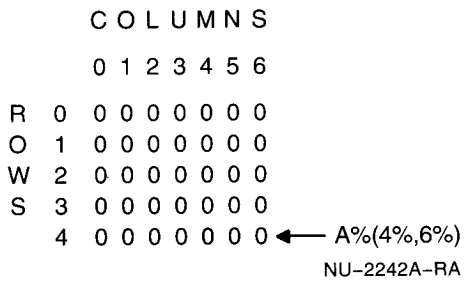
```
A(0%)   A(1%)   A(2%)   A(3%)   A(4%)   A(5%)   A(6%)
```

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

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

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

Figure 1–1 Subscripted Variables



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

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

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

BASIC–PLUS–2 arrays can be redimensioned at run time. See the *BASIC–PLUS–2 User's Guide* for more information on arrays.

1.4.5 Initialization of Variables

BASIC–PLUS–2 sets variables to zero or null values at the start of program execution. Variables initialized by BASIC–PLUS–2 include the following:

- Numeric variables and in-storage array elements (except those in MAP or COMMON statements).
- String variables (except those in MAP or COMMON statements).

- Variables in subprograms. Subprogram variables are initialized to zero or the null string each time the subprogram is called.

BASIC–PLUS–2 does not initialize the following:

- Virtual arrays
- Variables in MAP and COMMON areas

Note

Variables in a MAP statement referenced in an OPEN statement are initialized to zero or the null string when the file is opened. You can also use MACRO–11 routines to initialize MAP and COMMON areas. See the *BASIC–PLUS–2 User's Guide* for more information.

1.5 Constants

A **constant** is a numeric or character literal that does not change during program execution. A constant can also be named and associated with a data type. BASIC–PLUS–2 allows the following types of constants:

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

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

You can also use a special numeric literal notation to specify the value and data type of a numeric literal. Numeric literal notation is discussed in Section 1.5.4.

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

To simplify the representation of certain ASCII characters and mathematical values, BASIC–PLUS–2 also supplies some predefined constants.

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

1.5.1 Numeric Constants

A **numeric constant** is a literal or named constant whose value never changes. In BASIC–PLUS–2, a numeric constant can be either a floating-point number or an integer. The type and size of a numeric constant is determined by the following:

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

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

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

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

1.5.1.1 Floating-Point Constants

A **floating-point constant** is a literal or named constant with one or more decimal digits, either positive or negative, with an optional decimal point and an optional exponent (E notation). If the default data or constant type is INTEGER, BASIC–PLUS–2 will treat the literal as an INTEGER, unless it contains a decimal point or is in E notation.

The following are examples of floating-point literals if the default data type is REAL:

```
-8.738     239.21E-6     .79     299
```

The following are examples of floating-point literals if the default data type is INTEGER:

```
-8.738     239.21E-6     .79     299.
```

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

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

Table 1–3 compares numbers in standard and E notation.

Table 1–3 Numbers in E Notation

Standard Notation	E Notation
.0000001	.1E-06
1,000,000	.1E+07
-10,000,000	-.1E+08
100,000,000	.1E+09
1,000,000,000,000	.1E+13

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

1.5.1.2 Integer Constants

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

The following are examples of integer constants if the default data type is REAL:

```
81257%      -3477%      79%
```

The following are examples of integer constants if the default data type is INTEGER:

```
81257       -3477       79
```

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

If you want BASIC–PLUS–2 to treat numeric literals as integer numbers, you must do one of the following:

- Set the default data type to INTEGER
- Specify OPTION CONSTANT TYPE = INTEGER
- Make sure the literal has a percent sign suffix
- Use explicit literal notation

The BASIC–PLUS–2 compiler must convert numeric literals when assigning them to integer variables if they have different data types. This means that your program runs somewhat slower than it would if integer values were explicitly declared. You can prevent this conversion step if you do the following:

- Set the default data type to INTEGER
- Specify OPTION CONSTANT TYPE = INTEGER
- Use percent signs for integer constants
- Use numeric literal notation
- Use named integer constants

Note

You cannot use percent signs in integer constants that appear in DATA statements. An attempt to do so causes BASIC–PLUS–2 to signal the error message "Data format error" (ERR=50).

1.5.2 String Constants

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

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

You can embed double quotation marks within single quotation marks ('this is a "text" string') and vice versa ("this is a 'text' string"). Note, however, that BASIC–PLUS–2 does not accept incorrectly paired quotation marks and that only the outer quotation marks must be paired. The following character strings, for example, are valid:

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

The following strings are not valid:

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

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

The BASIC–PLUS–2 compiler determines the value of the string constant by scanning all its characters. For example, because of the number of spaces between the delimiters and the characters, these two string constants are not the same:

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

BASIC–PLUS–2 stores every character between delimiters exactly as you type it into the source program, including the following:

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

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

Example

```
300  PRINT "END-OF-FILE REACHED"
   .
   .
   .
     END
```

Output

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

Note, however, that BASIC–PLUS–2 prints double or single quotation marks when they are enclosed in a second paired set:

Example

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

Output

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

1.5.3 Named Constants

BASIC–PLUS–2 allows you to name constants. You can assign a name to a constant that is either internal or external to your program and refer to the constant by name throughout the program. This naming feature is useful for the following reasons:

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

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

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

1.5.3.1 Naming Constants Within a Program Unit

You name constants within a program unit with the DECLARE statement.

Example

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

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

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

The value associated with a named constant can be a compile-time expression as well as a literal value, as shown in the following example:

Example

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

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

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

You can specify the value of a constant with an expression for STRING and INTEGER data types, but not for floating-point constants. Only STRING and INTEGER constants can be named as expressions in DECLARE CONSTANT statements. The following example illustrates the concept of naming constants as expressions:

```
50  DECLARE DOUBLE CONSTANT   &
          min_value = 0
          max_value = PI
```

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

1.5.3.2 Naming Constants External to a Program Unit

To declare constants outside the program unit, use the EXTERNAL statement.

Example

```
50  EXTERNAL WORD CONSTANT IE.SUC
```

This line declares IE.SUC, a success code, to be an external WORD constant. BASIC–PLUS–2 allows only WORD constants. The task builder supplies the values for the constants specified in EXTERNAL statements.

External constant names cannot exceed six characters and must conform to the rules for naming external variables listed in Section 1.4.1. No external constant name can have embedded spaces. In BASIC–PLUS–2, the named constant might be a a global constant declared in a MACRO–11 program or an RMS–11 constant.

1.5.4 Explicit Literal Notation

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

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

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

A ASCII

R RAD-50

D Decimal (base 10)

B Binary (base 2)

O Octal (base 8)

X Hexadecimal (base 16)

The BASIC–PLUS–2 default radix is decimal. Binary, octal, and hexadecimal notation allow you to set or clear individual bits in the representation of an integer. This feature is useful in forming conditional expressions and in using logical operations. The ASCII radix causes BASIC–PLUS–2 to translate a single ASCII character to its decimal equivalent. This decimal equivalent is an INTEGER value; you specify whether the INTEGER subtype should be BYTE, WORD, or LONG.

Sign is required only when specifying a negative numeric string value. It must be outside the quoted string.

Num-str-lit is a numeric string literal. It can be the digits 0 and 1 when the radix is binary, the digits 0 through 7 when the radix is octal, the digits 0 through F when the radix is hexadecimal, and the digits 0 through 9 when the radix is decimal. When the radix is ASCII, *num-str-lit* can be any valid ASCII character. When the radix is RAD-50, *num-str-lit* must be exactly three valid

RAD-50 characters; if less than three characters are required, pad with spaces on the left or right.

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

B	BYTE
W	WORD
L	LONG
F	SINGLE
D	DOUBLE
C	CHARACTER

For example:

R" $"B	Specifies a BYTE decimal constant with a value of 27
R"ABC"W	Specifies a WORD decimal constant with a value of 1683
D"255"L	Specifies a LONG decimal constant with a value of 255
"4000"F	Specifies a SINGLE decimal constant with a value of 4000
-"125"B	Specifies a BYTE decimal constant with a value of -125
A"M"L	Specifies a LONG integer constant with a value of 77
A"m"B	Specifies a BYTE integer constant with a value of 109

If you specify a binary, octal, or hexadecimal radix, *data-type* must be an integer. If you do not specify a data type, BASIC–PLUS–2 uses the default integer data type. For example:

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

When you specify a radix other than decimal, overflow checking is performed as if the numeric string were an unsigned integer. However, when this value is assigned to a variable or used in an expression, the BASIC–PLUS–2 compiler treats it as a signed integer.

In the following example, BASIC–PLUS–2 sets all eight bits in storage location A. Because A is a BYTE integer, it has only eight bits of storage. Because the 8-bit two's complement of 1 is 11111111, its value is –1. If the data type were W (WORD), BASIC–PLUS–2 would set the bits to 0000000011111111, and its value would be 255.

Example

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

Output

```
-1
```

Note

In BASIC–PLUS–2, the letter D can appear in both the radix position and the data type position. A letter D in the radix position specifies that the numeric string is to be treated as a decimal number (base 10). A letter D in the data type position specifies that the value is to be treated as a double-precision, floating-point constant.

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

[radix] num-str-lit C

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

The following example declares a string constant named *control_g* (ASCII decimal value 7). When BASIC–PLUS–2 executes the PRINT statement, the terminal bell sounds.

Example

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

1.5.5 Predefined Constants

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

Predefined constants help you do the following:

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

Table 1–4 lists the predefined constants supplied by BASIC–PLUS–2, their ASCII values, and their functions.

Table 1–4 Predefined Constants

Constant	Decimal ASCII Value	Function
BEL (Bell)	7	Sounds the terminal bell
BS (Backspace)	8	Moves the cursor one position to the left
HT (Horizontal Tab)	9	Moves the cursor to the next horizontal tab stop
LF (Line Feed)	10	Moves the cursor to the next line
VT (Vertical Tab)	11	Moves the cursor to the next vertical tab stop
FF (Form Feed)	12	Moves the cursor to the start of the next page
CR (Carriage Return)	13	Moves the cursor to the beginning of the current line
SO (Shift Out)	14	Shifts out for communications networking, screen formatting, and alternate graphics
SI (Shift In)	15	Shifts in for communications networking, screen formatting, and alternate graphics

(continued on next page)

Table 1–4 (Cont.) Predefined Constants

Constant	Decimal ASCII Value	Function
ESC (Escape)	27	Marks the beginning of an escape sequence
SP (Space)	32	Inserts one blank space in program output
DEL (Delete)	127	Deletes the last character entered
PI	None	Represents the number PI with the precision of the default floating-point data type

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

Example

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

Output

```
NAME:
```

The following example shows how to print and underline a word on a VT100 terminal screen:

Example

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

Output

```
NAME:
```

Note that the *m* in the above example must be lowercase.

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

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

Example

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

1.6 Expressions

BASIC–PLUS–2 expressions consist of operands (numbers, strings, constants, variables, functions, and array elements) separated by arithmetic, string, relational, and logical operators.

Almost all BASIC–PLUS–2 expressions yield numeric values. The only exceptions are string concatenation expressions and invocations of string-valued functions. By using different combinations of numeric operators and operands, and by using the resulting values, you can produce the following:

- Numeric expressions
- String expressions
- Conditional expressions

BASIC–PLUS–2 evaluates expressions according to operator precedence and uses the results in program execution. Parentheses can be used to group operands and operators, thus controlling the order of evaluation.

BASIC–PLUS–2 does not detect integer or floating-point overflow when evaluating expressions. You must make sure that your calculations do not overflow; otherwise, the results will be unpredictable.

The following sections explain the types of expressions you can create and the way BASIC–PLUS–2 evaluates expressions.

1.6.1 Numeric Expressions

Numeric expressions consist of floating-point or integer operands separated by arithmetic operators and optionally grouped by parentheses. Table 1–5 shows how numeric operators work in numeric expressions.

Table 1–5 Arithmetic Operators

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

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

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

The following expression is not valid:

```
A - * B
```

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

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

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

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

When an expression contains operands with different data types, the data type of the result is determined by BASIC–PLUS–2's data type promotion rules:

- With one exception, BASIC–PLUS–2 promotes operands with different data types to the lowest common data type that can hold the largest or most precise possible value of either operand's data type. BASIC–PLUS–2 then performs the operation using that data type, and yields a result of that data type.

- The exception is that when an operation involves SINGLE and LONG data types, BASIC–PLUS–2 promotes the LONG data type to SINGLE rather than DOUBLE, performs the operation, and yields a result of the SINGLE data type.

Note that BASIC–PLUS–2 does sign extension when converting BYTE and WORD integers to a higher INTEGER data type (WORD or LONG). The high-order bit (the sign bit) determines how the additional bits are set when the BYTE or WORD is converted to WORD or LONG. If the high-order bit is zero (positive), all higher-order bits in the converted BYTE or WORD are set to zero. If the high-order bit is 1 (negative), all higher-order bits in the converted BYTE or WORD are set to 1.

Table 1–6 lists the data type results possible in numeric expressions that combine BYTE, WORD, LONG, SINGLE, and DOUBLE data.

Table 1–6 Result Data Types in BASIC–PLUS–2 Expressions

	BYTE	WORD	LONG	SINGLE	DOUBLE
BYTE	BYTE	WORD	LONG	SINGLE	DOUBLE
WORD	WORD	WORD	LONG	SINGLE	DOUBLE
LONG	LONG	LONG	LONG	SINGLE	DOUBLE
SINGLE	SINGLE	SINGLE	SINGLE	SINGLE	DOUBLE
DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE

For example, if one operand is SINGLE and one operand is DOUBLE, BASIC–PLUS–2 promotes the SINGLE value to DOUBLE, performs the specified operation, and returns the result as a DOUBLE value. This promotion is necessary because the SINGLE data type has less precision than the DOUBLE value, whereas the DOUBLE data type can represent all possible SINGLE values. If BASIC–PLUS–2 did not promote the SINGLE value and the operation yielded a result outside of the SINGLE range, loss of precision and significance would occur.

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

1.6.2 String Expressions

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

Example

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

Output

```
Type two words to be combined? rattle
? brained

rattlebrained
BASIC2
```

1.6.3 Conditional Expressions

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

Logical expressions contain integer operands and logical operators. BASIC–PLUS–2 determines whether the specified logical expression is true or false by testing the numeric result of the expression. Note that in conditional expressions, as in any numeric expression, when BYTE and WORD operands are converted to WORD and LONG, the specified operation is performed in the higher data type, and the result returned is also of the higher data type. When one of the operands is a negative value, this conversion will produce accurate but perhaps confusing results, because BASIC–PLUS–2 performs a sign extension when converting BYTE and WORD integers to a higher integer data type. See Section 1.6.1 for information on integer conversion rules.

1.6.3.1 Numeric Relational Expressions

Operators in numeric relational expressions compare the values of two operands and return either a –1 if the relation is true or a zero if the relation is false. The data type of the result is the default integer type.

Example 1

```
10  A = 10
    B = 15
    X% = (A <> B)

20  IF X% = -1%
       THEN PRINT 'Relationship is true'
       ELSE PRINT 'Relationship is false'
    END IF
```

Output 1

```
Relationship is true
```

Example 2

```
10  A = 10
    B = 15
    X% = A = B

20  IF X% = -1%
      THEN PRINT 'Relationship is true'
      ELSE PRINT 'Relationship is false'
    END IF
```

Output 2

```
Relationship is false
```

Table 1–7 shows how numeric operators work in numeric relational expressions.

Table 1–7 Numeric Relational Operators

Operator	Example	Meaning
=	A = B	A is equal to B.
<	A < B	A is less than B.
>	A > B	A is greater than B.
<= or =<	A <= B	A is less than or equal to B.
>= or =>	A >= B	A is greater than or equal to B.
<> or ><	A <> B	A is not equal to B.
==	A == B	A and B will print the same because they are equal to six significant digits.

1.6.3.2 String Relational Expressions

Operators in string relational expressions determine how BASIC–PLUS–2 compares strings. The BASIC–PLUS–2 compiler determines the value of each character in the string by converting it to its ASCII value. See the *BASIC–PLUS–2 User's Guide* for a list of ASCII values. BASIC–PLUS–2 compares the strings character by character, left to right, until it finds a difference in ASCII value.

In the following example, BASIC–PLUS–2 compares *A$* and *B$* character by character. The strings are identical up to the third character. Because the ASCII value of *Z* (90) is greater than the ASCII value of *C* (67), *A$* is less than *B$*. BASIC–PLUS–2 evaluates the expression *A$* < *B$* as true (–1) and prints "ABC comes before ABZ."

Example

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

If two strings of differing lengths are identical up to the last character in the shorter string, BASIC–PLUS–2 pads the shorter string with spaces (ASCII value 32) to generate strings of equal length, unless the operator is the double equal sign (==). If the operator is the double equal sign, BASIC–PLUS–2 does not pad the shorter string.

In the following program, BASIC–PLUS–2 compares "ABCDE" to "ABC " to determine which string comes first in the collating sequence. "ABC" comes before "ABCDE" because the ASCII value for space (32) is lower than the ASCII value of *D* (68). Then BASIC–PLUS–2 compares "ABC " with "ABC" using the double equal sign and determines that the strings do not match exactly without padding. The third comparison uses the single equal sign. BASIC–PLUS–2 pads "ABC" with spaces and determines that the two strings match with padding.

Example

```
10  A$ = 'ABCDE'
    B$ = 'ABC'
    PRINT 'B$ comes before A$' IF B$ < A$
    PRINT 'A$ comes before B$' IF A$ < B$
    C$ = 'ABC '

    IF B$ == C$
          THEN PRINT 'B$ exactly matches C$'
          ELSE PRINT 'B$ does not exactly match C$'
    END IF

    IF B$ = C$
          THEN PRINT 'B$ matches C$ with padding'
          ELSE PRINT 'B$ does not match C$'
    END IF
```

Output

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

Table 1–8 shows how numeric operators work in string relational expressions.

Table 1–8 String Relational Operators

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

BASIC–PLUS–2 treats unquoted strings typed in response to the INPUT statement differently from quoted strings; it does so by ignoring leading and trailing spaces and tabs. For example, it evaluates the quoted strings "ABC" and "ABC " as equal but not identical because the == operator does not pad the shorter string with spaces. When you input those same strings as unquoted strings in response to the INPUT prompt, BASIC–PLUS–2 evaluates them

as equal and identical because it ignores the trailing spaces. The LINPUT statement, on the other hand, treats unquoted strings as string literals, so the trailing spaces are part of the string, and BASIC–PLUS–2 evaluates the strings as equal but not identical.

1.6.3.3 Logical Expressions

A logical expression can have one of the following formats:

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

Logical expressions are valid only when the operands are integers. If the expression contains two integer operands of differing data types, the resulting integer has the same data type as that of the higher integer operand. For instance, the result of an expression that contains a BYTE integer and a WORD integer would be a WORD integer. Table 1–6 shows how integer data types interact with each other in expressions.

BASIC–PLUS–2 determines whether the condition is true or false by testing the result of the logical expression to see whether any bits are set. If no bits are set, the value of the expression is zero and it is evaluated as false; if any bits are set, the value of the expression is nonzero and the expression is evaluated as true. BASIC–PLUS–2 generally accepts any nonzero value in logical expressions as true. However, logical operators can return unanticipated results unless –1 is specified for true values and zero for false. Table 1–9 lists the logical operators.

Note

It is recommended that you use logical operators on the results of relational expressions to avoid obtaining unanticipated results.

Table 1–9 Logical Operators

Operator	Example	Meaning
NOT	NOT A%	The bit-by-bit complement of A%. If A% is true (–1), NOT A% is false (0).
AND	A% AND B%	The logical product of A% and B%. A% AND B% is true only if both A% and B% are true.
OR	A% OR B%	The logical sum of A% and B%. A% OR B% is false only if both A% and B% are false; otherwise, A% OR B% is true.
XOR	A% XOR B%	The logical exclusive OR of A% and B%. A% XOR B% is true if either A% or B% is true but not if both are true.
EQV	A% EQV B%	The logical equivalence of A% and B%. A% EQV B% is true if A% and B% are both true or both false; otherwise, the value is false.
IMP	A% IMP B%	The logical implication of A% and B%. A% IMP B% is false only if A% is true and B% is false; otherwise, the value is true.

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

Figure 1–2 Truth Tables

A%	NOT A%
0	–1
–1	0

A%	B%	A% OR B%
0	0	0
0	–1	–1
–1	0	–1
–1	–1	–1

A%	B%	A% AND B%
0	0	0
0	–1	0
–1	0	0
–1	–1	–1

A%	B%	A% EQV B%
0	0	–1
0	–1	0
–1	0	0
–1	–1	–1

A%	B%	A% XOR B%
0	0	0
0	–1	–1
–1	0	–1
–1	–1	0

A%	B%	A% IMP B%
0	0	–1
0	–1	–1
–1	0	0
–1	–1	–1

NU-2200A-RA

The operators XOR and EQV are logical complements.

In logical expressions any nonzero value is evaluated as true, while in relational expressions a –1 is generated as a true value. Logical operators set bits in the result of the expression; any bit set is a nonzero value and is evaluated as true. For this reason, it is important to use logical operators on the results of relational expressions (the values of –1 and zero) to avoid unanticipated results. In the following example, the values of *A%* and *B%* both test as true because they are nonzero values. However, the logical AND of these two variables returns an unanticipated result of false.

Example

```
A% = 2%
B% = 4%

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

Output

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

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

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

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

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

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

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

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

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

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

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

The result of –1% AND –1% is –1% because BASIC–PLUS–2 sets bits in the result for each corresponding bit that is set in the operands. The result tests as true because it is a nonzero value.

Example

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

Output

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

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

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

Example

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

Output

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

1.6.4 Evaluating Expressions

BASIC–PLUS–2 evaluates expressions according to operator precedence. Each arithmetic, relational, and string operator in an expression has a position in the hierarchy of operators. The operator's position informs BASIC–PLUS–2 of the order in which to perform the operation. Parentheses can change the order of precedence.

Table 1–10 lists all operators as BASIC–PLUS–2 evaluates them. Note the following:

- Operators with equal precedence are evaluated logically from left to right.

- BASIC–PLUS–2 evaluates expressions enclosed in parentheses first, even when the operator in parentheses has a lower precedence than that outside the parentheses.
- The addition (+) and multiplication (*) operators are evaluated in algebraic order.

Table 1–10 Numeric Operator Precedence

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

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

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

There is one exception to this order of precedence: When an operator that does not require operands on either side of it (such as NOT) immediately follows an operator that does require operands on both sides (such as the addition operator (+)), BASIC–PLUS–2 evaluates the second operator first. For example:

```
A% + NOT B% + C%
```

This expression is evaluated as follows:

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

BASIC–PLUS–2 evaluates the expression NOT B before it evaluates the expression A + NOT B. When the NOT expression does not follow the addition (+) expression, the normal order of precedence is followed:

```
NOT A% + B% + C%
```

This expression is evaluated as follows:

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

BASIC–PLUS–2 evaluates the two expressions (A% + B%) and ((A% + B%) + C%) because the + operator has a higher precedence than the NOT operator.

BASIC–PLUS–2 evaluates nested parenthetical expressions from the inside out.

In the following program, BASIC–PLUS–2 evaluates the parenthetical expression *A* quite differently from expression *B*. For expression *A*, BASIC–PLUS–2 evaluates the innermost parenthetical expression (25 + 5) first, then the second inner expression (30 / 5), then (6 * 7), and finally (42 + 3). For expression *B*, BASIC–PLUS–2 evaluates (5 / 5) first, then (1 * 7), then (25 + 7 + 3) to obtain a different value.

Example

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

Output

```
45
35
```

2

Environment Commands

BASIC–PLUS–2 environment commands are commands that you can use while in the BASIC–PLUS–2 environment. With environment commands, you can display, edit, and merge BASIC–PLUS–2 programs, set compiler defaults, move BASIC–PLUS–2 source programs to and from storage, and execute programs. This chapter lists alphabetically all of the compiler commands that you can use within the BASIC–PLUS–2 environment. For information on immediate mode statements, see the *BASIC–PLUS–2 User's Guide*.

$ system-command

You can execute a system command while in the BASIC–PLUS–2 environment by typing a dollar sign ($) before the command. BASIC–PLUS–2 passes the command to the operating system for execution. On RSX systems, the context of the BASIC–PLUS–2 environment and the program currently in memory do not change. On RSTS/E systems, the system command executes and control returns to the default run-time system, not to BASIC–PLUS–2.

Format

```
$ system-command
```

Syntax Rules

None.

Remarks

1. BASIC–PLUS–2 passes the *system-command* directly to the operating system without checking for validity.
2. Your terminal displays any error messages or output that the command generates.
3. On RSX systems, control returns to the BASIC–PLUS–2 environment after the command executes. The context (source file status, loaded modules, and so on) of the BASIC–PLUS–2 environment and the program currently in memory do not change unless the command causes the operating system to abort BASIC–PLUS–2 or log you out.
4. On RSTS/E systems, the context of the environment and the program currently in memory are lost. After the system command executes, control passes to monitor level, not to BASIC–PLUS–2.
5. If you have made changes to the program currently in memory and enter a system command without first entering either the SCRATCH or REPLACE command, BASIC–PLUS–2 displays the message "Unsaved change has been made—type EXIT or CTRL/Z to exit."

Example

```
BASIC2

$ DIR STOCK.B2S
%Unsaved change has been made - type EXIT or CTRL/Z to exit.

BASIC2

REPLACE

BASIC2

$ DIR STOCK.B2S
```

APPEND

The APPEND command merges an existing BASIC–PLUS–2 source program with the program currently in memory.

Format

APPEND [file-spec]

Syntax Rules

File-spec is the file specification of the BASIC–PLUS–2 program you want to merge with the program currently in memory. If you do not supply a file type, the default is B2S.

Remarks

1. If you type APPEND without specifying a file name, BASIC–PLUS–2 displays the following prompt:

   ```
   Append file name--
   ```

   If you do not supply a file name in response to this prompt, BASIC–PLUS–2 searches for the file NONAME.B2S. If NONAME.B2S does not exist, BASIC–PLUS–2 signals the error "Can't find file or account."

2. You can append the contents of *file-spec* to a source program that is either called into memory with the OLD command or created in the BASIC environment. If there is no program in memory, BASIC–PLUS–2 appends the file to an empty program with the default file name NONAME.

3. If *file-spec* contains a BASIC–PLUS–2 line with the same line number as a line of the program in memory, the line in the appended file replaces the line of the program in memory and BASIC–PLUS–2 signals the warning "%Duplicate line number *n* encountered." If no line numbers are duplicates, BASIC–PLUS–2 inserts the appended lines into the program in memory in sequential, ascending line number order.

4. The APPEND command does not change the name of the program in memory.

5. If you do not save the appended version of the program and attempt to exit from the BASIC–PLUS–2 environment, BASIC–PLUS–2 signals the warning message "Unsaved change has been made, CTRL/Z or EXIT to exit."

Example

```
BASIC2

New FIRST_TRY.B2S

BASIC2
10  PRINT "First program"

APPEND NEW_PROG.B2S

BASIC2

LISTNH

10  PRINT "First Program"

20  PRINT "This section has been appended"

    .
    .
    .
```

BRLRES

The BRLRES command allows you to specify a memory-resident or user-created library to be used when you task-build a program. When you use the BUILD command, BASIC–PLUS–2 includes the specified library in the Task Builder command file. Your system manager selects the default library for the BRLRES command when installing BASIC–PLUS–2.

Format

BRLRES [lib-param]

lib-param: { file-spec / NONE }

Syntax Rules

1. *File-spec* is the library file specification. The file specification can either be a library supplied by BASIC–PLUS–2 or a user-created library.
2. *NONE* tells the Task Builder not to link your task to the default memory-resident library. Therefore, the Task Builder links your task to the BASIC–PLUS–2 object module library, BP2OTS.OLB.
3. If you do not supply a *lib-param*, BASIC–PLUS–2 displays the following prompt:

   ```
   File spec [NONE]--
   ```

   If you press the Return key in response to this prompt, NONE is the default. NONE indicates that the Task Builder will not link your task to the default memory-resident library.

Remarks

1. BASIC–PLUS–2 supplies the following memory-resident libraries:
   - BP2RES
   - BP2SML

   These BASIC–PLUS–2 memory-resident libraries are optional. Your system manager decides whether to install them during installation. For information about the memory-resident libraries available on your system, see your system manager.

2. The BUILD command includes the library you specify with the BRLRES command in the Task Builder command file. Therefore, you must specify the BRLRES command before you specify the BUILD command so the new library specification is entered into the Task Builder command file. Otherwise, the Task Builder command file remains unchanged and the existing library in the command file is used.
3. The BRLRES library you specify remains in effect until you either specify a new library with the BRLRES command or exit from the BASIC–PLUS–2 environment. Once you exit from the BASIC–PLUS–2 environment, the default memory-resident library is used.
4. You can override a library specified with the BRLRES command by using the /BRLRES qualifier to the BUILD command. The library you specify remains in effect only for that particular build operation.
5. If you specify a memory-resident library that is not available, the Task Builder signals an error message.
6. For more information on BASIC–PLUS–2 memory-resident libraries, see the *BASIC–PLUS–2 User's Guide*.

Examples

1.
```
! On RSX-11M/M-PLUS Systems
BRLRES LB:[1,1]BP2RES
```

2.
```
! On RSTS/E Systems
BRLRES LB:BASIC2
```

BUILD

The BUILD command generates a command (CMD) file and an overlay description language (ODL) file for the Task Builder. The CMD file contains instructions that enable the Task Builder to link your program module or modules with libraries and other routines. The ODL file specifies how program segments should be organized in memory during program execution.

Format

BUILD [prog-nam [,sub-nam, . . .]] [/qualifier] . . .

Command Qualifiers	Defaults
/BRLRES sep lib-param	See text.
/[NO]CLUSTER[sep lib-param]	/NOCLUSTER
/DSKLIB sep file-spec	See text.
/[NO]DUMP	/NODUMP
/EXTEND sep int-const	/EXTEND=512
/[NO]IDS	/NOIDS
/[NO]INDEX	/NOINDEX
/LIBRARY sep lib-param	See text.
/[NO]MAP	/NOMAP
/ODLRMS sep odl-param	See text.
/[NO]RELATIVE	/NORELATIVE
/RMSRES sep lib-param	See text.
/[NO]SEQUENTIAL	/NOSEQUENTIAL
/[NO]VIRTUAL	/NOVIRTUAL

Syntax Rules

1. *Prog-nam* is the name of the program you want to build. If you do not specify a program, BASIC–PLUS–2 creates CMD and ODL files for the current program in memory, or for NONAME.B2S if there is no current program in memory.
2. *Sub-nam* is the name of the subprogram you want to link to the main program.
3. If you specify a subprogram name you must also specify a program name.
4. The BUILD command line must fit on a single 80-character line.

Remarks

1. The BUILD command does not change the current context of the BASIC–PLUS–2 environment.
2. The BUILD command generates the CMD and ODL files. It does not cause the Task Builder to begin operation.
   - The BUILD command generates a CMD file with the same name as the program and a file type of CMD.
   - The BUILD command generates an ODL file with the same name as the program and a file type of ODL.
3. If you do not specify any BUILD command qualifiers, the BUILD command accepts defaults from previously specified BRLRES, DSKLIB, ODLRMS, RMSRES, LIBRARY, and SET commands.
4. The qualifiers to the BUILD command tell the Task Builder to perform special operations on object modules when you task-build the program. You can abbreviate all qualifiers to the first three letters of the qualifier keyword.
5. When you exit from the BASIC–PLUS–2 environment, all options set with qualifiers return to the system default values. Use the SHOW command to display your system defaults before setting any qualifiers.
6. BASIC–PLUS–2 will not cluster a BASIC–PLUS–2 memory-resident library when using Instruction and Data space (I- and D-Space). If you attempt to use I- and D-Space and use the BASIC–PLUS–2 memory-resident library, BASIC–PLUS–2 does not use the library and does not signal an error.

Command Qualifiers

/BRLRES { **=** | **:** } { **file-spec** | **NONE** }

The /BRLRES qualifier lets you specify a memory-resident library to be linked to your task. *File-spec* can be either a library supplied by BASIC–PLUS–2 or a user-created library. *NONE* tells the Task Builder not to link your task to the default memory-resident library. Instead, the Task Builder links your task to BASIC–PLUS–2 object module library BP2OTS.OLB. If you do not specify the /BRLRES qualifier, the Task Builder links your task to the current default memory-resident library by default. See the description of the BRLRES command for more information.

/[NO]CLUSTER [{ = } file-spec]
{ : } NONE

The /CLUSTER qualifer causes the Task Builder to cluster memory-resident libraries to increase the space available for your task. Before you use the /CLUSTER qualifier, at least two memory-resident libraries must be linked to the task: the BASIC–PLUS–2 memory-resident library, and one other memory-resident library.

- *File-spec* specifies the memory-resident library to be clustered. The specified library must be in the account LB: on RSTS/E systems or the account LB:[1,1] on RSX systems.
- *NONE* specifies that only the BASIC–PLUS–2 and RMS–11 libraries are clustered.
- /CLUSTER with no argument causes the Task Builder to cluster the default memory-resident library. If there is no default cluster library, /CLUSTER with no argument acts the same as /CLUSTER:NONE.

The /NOCLUSTER qualifier tells the Task Builder not to cluster memory-resident libraries to increase the space available for your task. /NOCLUSTER is the default. See the *BASIC–PLUS–2 User's Guide* for more information on using RMS–11 libraries.

/DSKLIB { = } { file-spec }
{ : } { NONE }

The /DSKLIB qualifier lets you specify a disk-resident object module library to be linked to your program. *File-spec* can be either a library supplied by BASIC–PLUS–2 or a user-created library. *NONE* tells the Task Builder not to link your task to the default object module library. If you do not specify the /DSKLIB qualifier, BASIC–PLUS–2 links your task to the current default disk-resident library. See the description of the DSKLIB command for more information.

/[NO]DUMP

If your program aborts with a fatal error, the /DUMP qualifier causes the Task Builder to generate a memory dump. The /NODUMP qualifier causes the Task Builder not to generate a memory dump when the program aborts. The default is /NODUMP.

/EXTEND { = } int-const
{ : }

The /EXTEND qualifier specifies the amount of space to be added to the initial task size when the task is started. The Task Builder rounds the extension up

to the nearest 32-word boundary. The maximum allowed extension is 32000. The default is /EXTEND=512.

/[NO]IDS
The /IDS qualifier causes the Task Builder to build your task with I- and D-space support. I- and D-space provides a faster program execution time and also allows you to execute larger programs than usual. It is recommended that you do not use the BASIC–PLUS–2 memory-resident library in an I- and D-space task. By default, when you specify the /IDS qualifier, BASIC–PLUS–2 removes all references to the BASIC–PLUS–2 memory-resident library from the Task Builder command file. The /NOIDS qualifier tells the Task Builder not to build the task with I- and D-space support. /NOIDS is the default. See the *BASIC–PLUS–2 User's Guide* for more information on enabling I- and D-space for program execution.

[NO]INDEX
The /INDEX qualifier causes the Task Builder to include the code needed for indexed file operations. BASIC–PLUS–2 enables this qualifier automatically for programs containing an OPEN statement with the ORGANIZATION INDEXED clause. If you specify /NOINDEX, the Task Builder does not include the code needed for indexed file operations. The default is /NOINDEX.

/LIBRARY { **=** } { **lib-name** }
{ **:** } { **file-spec** }
{ **NONE** }

The /LIBRARY qualifier lets you specify a memory-resident library to be linked to your program. *File-spec* and *lib-name* can be either a library supplied with BASIC–PLUS–2 or a user-created library. If you specify a *lib-name* with no device, BASIC–PLUS–2 assumes LB: on RSTS/E systems and LB:[1,1] on RSX systems. *NONE* tells the Task Builder not to link your task to the default memory-resident library. Therefore, the Task Builder links to BASIC–PLUS–2 disk-resident object module library BP2OTS.OLB. If you do not specify the /LIBRARY qualifier, the Task Builder links your task to the current default memory-resident library. See the description of the LIBRARY command for more information.

/[NO]MAP
The /MAP qualifier causes the Task Builder to generate an allocation map. The /NOMAP qualifier causes the Task Builder to not generate an allocation map. /NOMAP is the default.

/ODLRMS { = } { **file-spec** }
{ : } { **NONE** }

The /ODLRMS qualifier lets you specify an Overlay Description Language (ODL) file for the Task Builder to use when task-building your program. The ODL file describes how the Task Builder should overlay the task in memory. When BASIC–PLUS–2 executes the BUILD command, the ODL file is included in the Task Builder command file. *File-spec* can be either an ODL file supplied by RMS or a user-created file. *NONE* tells the Task Builder not to link your task to the default ODL file. If you do not specify the /ODLRMS qualifier, the Task Builder uses the current default ODL file to task-build your program. See the description of the ODLRMS command for more information.

/[NO]RELATIVE

The /RELATIVE qualifier causes the Task Builder to include the code needed for relative file operations. BASIC–PLUS–2 sets this qualifier automatically for programs containing an OPEN statement with the ORGANIZATION RELATIVE clause. If you specify /NORELATIVE, the Task Builder does not include the code necessary for relative file operations. /NORELATIVE is the default.

/RMSRES { = } { **file-spec** }
{ : } { **NONE** }

The /RMSRES qualifier lets you specify an RMS memory-resident library to be linked to your program. The RMS library supplies RMS code for file and record operations. *File-spec* can be either a library supplied by RMS or a user-created library. *NONE* tells the Task Builder not to link your task to the default RMS library. Therefore the Task Builder links your task to the RMS object module library RMSLIB.OLB. If you do not specify the /RMSRES qualifier, the Task Builder links your task to the current default memory-resident library. See the description of the RMSRES command for more information.

/[NO]SEQUENTIAL

The /SEQUENTIAL qualifier causes the Task Builder to include the RMS–11 code needed for sequential file operations. BASIC–PLUS–2 sets this qualifier automatically for programs containing an OPEN statement with the ORGANIZATION SEQUENTIAL clause. If you specify /NOSEQUENTIAL, the Task Builder does not include the RMS–11 code necessary for sequential file operations. /NOSEQUENTIAL is the default.

/[NO]VIRTUAL
The /VIRTUAL qualifier causes BASIC–PLUS–2 to include the RMS code needed for virtual array and block I/O file operations. BASIC–PLUS–2 sets this qualifier automatically when you compile a program containing an OPEN statement with an ORGANIZATION VIRTUAL clause. If you specify /NOVIRTUAL, BASIC–PLUS–2 does not include the RMS code necessary for virtual array and block I/O file operations. /NOVIRTUAL is the default.

Example

```
BUILD MAIN,SUB1,SUB2/DUMP/REL
```

COMPILE

The COMPILE command converts a BASIC–PLUS–2 source program to an object module and writes the object file to disk.

Format

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

Command Qualifiers	Defaults
/[NO]BOUND	/BOUND
/BYTE	/WORD
/[NO]CHAIN	See text.
/[NO]CROSS_REFERENCE [sep [NO]KEYWORDS]	/NOCROSS_REFERENCE
/[NO]DEBUG	/NODEBUG
/DOUBLE	/SINGLE
/[NO]FLAG sep [[NO]DECLINING]	/FLAG=DECLINING
/[NO]LINE	/LINE
/[NO]LIST	/NOLIST
/LONG	/WORD
/[NO]MACRO	/NOMACRO
/[NO]OBJECT	/OBJECT
/PAGE_SIZE sep int-const	/PAGE_SIZE=60
/[NO]SCALE sep const	/NOSCALE
/SINGLE	/SINGLE
/[NO]SYNTAX_CHECK	/NOSYNTAX_CHECK
/TYPE_DEFAULT sep default-clause	/TYPE_DEFAULT=REAL
/VARIANT sep int-const	/VARIANT=0
/[NO]WARNINGS	/WARNINGS
/WIDTH sep int-const	/WIDTH=132
/WORD	/WORD

Syntax Rules

1. *File-spec* is the file specification of an output file or files.
2. If you do not provide a file specification, the BASIC–PLUS–2 compiler uses the name of the program currently in memory for the file name, a default file type of OBJ for the object file, and a default file type of LST for the listing file, if a listing file is requested. If a macro file is requested, BASIC–PLUS–2 uses a default file type of MAC for the macro source code file.

3. You should not specify both a file name and file type. For example, if you enter the following command line, BASIC–PLUS–2 creates only the object file and names it NEWOBJ.FIL:

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

4. *File-spec* must precede all qualifiers.
5. */Qualifier* specifies a qualifier keyword that sets a BASIC–PLUS–2 default.
6. You can abbreviate all positive qualifiers to the first three letters of the qualifier keyword. You can abbreviate a negative qualifier to NO and the first three letters of the qualifier keyword.
7. You cannot specify the /OBJECT and /MACRO qualifiers together.

Remarks

1. If you specify an invalid qualifier, BASIC–PLUS–2 signals the error "Illegal switch," and the program does not compile. When qualifiers conflict, BASIC–PLUS–2 compiles the program using the last specified conflicting qualifier. For example, the following command line causes BASIC–PLUS–2 to compile the program currently in memory but does not cause BASIC–PLUS–2 to create an OBJ file.

   ```
   COMPILE/OBJ/NOOBJ
   ```

2. If there is no program in memory, or the COMPILE command does not execute, BASIC–PLUS–2 does not signal an error or warning.
3. On RSX systems, if an object file for the program already exists in your directory, BASIC–PLUS–2 creates a new version of the OBJ file. On RSTS/E systems BASIC–PLUS–2 overwrites it with the new object file.
4. Use the COMPILE/NOOBJECT command to check your program for errors without producing an object file.
5. When you exit from the BASIC–PLUS–2 environment, all options set with qualifiers return to the system default values. Use the SHOW command to display your system defaults before setting any qualifiers.

Command Qualifiers

/[NO]BOUND

The /NOBOUND qualifier eliminates the overhead of checking array boundaries when referencing memory-resident arrays of one or two dimensions. This can improve run time performance. Specifying or defaulting the /BOUND qualifier results in full array boundary checking.

Warning

When you specify /NOBOUND, the compiler generates array threads that omit boundary checking. If you incorrectly index beyond array limits, the OTS does not trap your errors. The consequences of such misuse are unpredictable. The user is responsible for ensuring the array handling integrity of the program before taking advantage of the /NOBOUND compilation option.

/BYTE

The /BYTE qualifier causes BASIC–PLUS–2 to allocate 8 bits of storage as the default for all integer data not explicitly typed in the program. Untyped integer values are treated as BYTE values and must be in the BYTE range or BASIC–PLUS–2 signals the error "Integer error or overflow." Table 1-2 in this manual lists BASIC–PLUS–2 data types and ranges. The default is /WORD.

/[NO]CHAIN

The /CHAIN qualifer can be used on RSTS/E systems only. The /CHAIN qualifier enables other programs to CHAIN into the program using the LINE clause of the CHAIN statement. If the program has more than 200 line numbers, the /NOCHAIN qualifier reduces the memory needs of the output program by disabling storage of line numbers in memory. You cannot chain from one DECNET node to another. The default is determined at installation.

/[NO]CROSS_REFERENCE[{ = : } [NO]KEYWORDS]

If you use the /CROSS_REFERENCE qualifier with the /LIST qualifier when you compile your program, the BASIC–PLUS–2 compiler includes cross-reference information in the program listing file. If you specify /CROSS_REFERENCE=KEYWORDS, BASIC–PLUS–2 also cross-references BASIC–PLUS–2 keywords used in the program. If you specify /NOCROSS_REFERENCE, BASIC–PLUS–2 does not include a cross reference section in the compiler listing. The default is /NOCROSS_REFERENCE.

/[NO]DEBUG

The /DEBUG qualifier appends to the object file information on symbolic references and line numbers. This information is used by the BASIC–PLUS–2 Debugger to debug your program. You must specify the /LINE qualifier when you specify the /DEBUG qualifier on the COMPILE command; otherwise, BASIC–PLUS–2 signals an error.

When you specify /DEBUG, control is passed to the debugger when the program is executed in the BASIC–PLUS–2 environment. If you specify /NODEBUG, information on program symbols and line numbers is not included in the object file and control is not passed to the debugger when the program executes. The default is /NODEBUG.

See the *BASIC–PLUS–2 User's Guide* for more information on using the BASIC–PLUS–2 Debugger.

/DOUBLE

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

/[NO]FLAG { = : } [NO]DECLINING

The /FLAG qualifier causes BASIC–PLUS–2 to provide compile-time information about program elements that are not recommended for new program development. For example, if you specify the DECLINING clause, BASIC–PLUS–2 flags the following source code as declining:

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

The default is /FLAG=DECLINING.

/[NO]LINE

The /LINE qualifier includes line number information in object modules. If you specify /NOLINE, BASIC–PLUS–2 does not include line number information in object modules. If you specify /NOLINE in a program containing the run-time ERL function, BASIC–PLUS–2 issues a warning that the /NOLINE qualifier has been overridden. The default is /LINE.

/[NO]LIST
The /LIST qualifier causes BASIC–PLUS–2 to produce a compiler listing file. The name of the listing file is the same as the name of the first program module specified, or the name of the program currently in memory if no file specification is provided. The listing file has a default file type of LST. If you specify /NOLIST, BASIC–PLUS–2 does not generate a compiler listing. /NOLIST is the default.

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

/[NO]MACRO
The /MACRO qualifier converts the program into MACRO–11 source code and saves it in a file with the same name as that of the program and a file type of MAC. A MACRO–11 file can be assembled. If you specify /NOMACRO, a MACRO–11 source code file is not generated. You cannot specify the /OBJECT qualifier with the /MACRO qualifier. The default is /NOMACRO.

/[NO]OBJECT
The /OBJECT qualifier generates an object module with the same file name as that of the program and a default file type of OBJ. The /NOOBJECT qualifier allows you to check your program for errors without creating an object file. If your program contains one or more fatal errors, an object module is not generated. You cannot specify the /MACRO qualifier with the /OBJECT qualifier. /OBJECT is the default.

/PAGE_SIZE { = | : } **int-const**

The /PAGE_SIZE qualifier sets the page size of the listing file. *Int-const* must be greater than zero or BASIC–PLUS–2 signals the warning message "Listing length out of range—ignored." The default is /PAGE_SIZE=60.

/[NO]SCALE { = | : } **const**

The /SCALE qualifier allows control of accumulated round-off errors when double-precision numbers (values typed DOUBLE) are used. Numbers are stored as multiples of 10 by setting *const* (the scale factor) from 0 through 6. A scale factor larger than 6 causes BASIC–PLUS–2 to signal the error message "Scale factor out of range–ignored." /NOSCALE is the default.

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

/[NO]SYNTAX_CHECK
The /SYNTAX_CHECK qualifier causes BASIC–PLUS–2 to perform syntax checking after each program line is typed. If you specify /NOSYNTAX_CHECK, BASIC–PLUS–2 does not perform syntax checking. The default is /NOSYNTAX_CHECK.

/TYPE_DEFAULT { = | : } { **REAL** | **INTEGER** | **EXPLICIT** }

The /TYPE_DEFAULT qualifier sets the default data type (REAL or INTEGER) for all data not explicitly typed in your program or specifies that all data must be explicitly typed (EXPLICIT).

- REAL specifies that all data not explicitly typed is floating-point data of the default size (SINGLE or DOUBLE).
- INTEGER specifies that all data not explicitly typed is integer data of the default size (BYTE, WORD, or LONG).
- EXPLICIT specifies that all data in a program must be explicitly typed. Implicitly declared variables cause BASIC–PLUS–2 to signal an error.

The default is TYPE_DEFAULT=REAL.

/VARIANT { = | : } **int-const**

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

/[NO]WARNINGS
The /WARNINGS qualifier causes BASIC–PLUS–2 to display warning messages during program compilation. The /NOWARNINGS qualifier causes BASIC–PLUS–2 to disable warning messages during program compilation. The default is /WARNINGS.

/WIDTH { = | : } int-const

The /WIDTH qualifier sets the width of the listing file. *Int-const* must be an integer from 72 through 132, or BASIC–PLUS–2 signals the message "Listing width out of range—ignored." The default is /WIDTH=132.

/WORD

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

Example

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

DELETE

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

Format

DELETE line-num [{ - , } line-num . . .]

Syntax Rules

1. The separator characters (comma or hyphen) allow you to delete individual lines or a block of lines.
2. If you separate line numbers with a comma (,), BASIC–PLUS–2 deletes each specified line number.
3. If you separate line numbers with a hyphen (-), BASIC–PLUS–2 deletes the inclusive range of lines. The lower line number must be specified first. If it is not specified first, BASIC–PLUS–2 signals an error: "BAD LINE NUMBER PAIR".
4. You can combine individual line numbers and line ranges in a single DELETE command. Note, however, that a line number range must be followed by a comma and not another hyphen, or BASIC–PLUS–2 signals an error. The following example deletes lines 70 through 80, line 110, and line 124:

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

Remarks

1. If you do not specify a line number, BASIC–PLUS–2 signals the error "Illegal Delete command."
2. BASIC–PLUS–2 signals an error if there are no lines in the specified range or if you specify an illegal line number.

Examples

1. DELETE 50

2. DELETE 50,60,90-110

DSKLIB

The DSKLIB command lets you select a disk-resident, object module library to be used when you build your program. When you use the BUILD command, BASIC–PLUS–2 includes the specified library in the Task Builder command file.

Format

DSKLIB [file-spec]

Syntax Rules

1. *File-spec* can be a disk-resident, object module library supplied with BASIC–PLUS–2 or a user-created library.
2. If you specify the DSKLIB command without a *file-spec*, BASIC–PLUS–2 prompts for one and displays the name of the current default disk-resident library. If you press the Return key without specifying a library file specification, the current default disk-resident library is used.

Remarks

1. The disk-resident object module libraries supplied by BASIC–PLUS–2 are as follows:
   - LB:[1,1]BP2OTS.OLB (on RSX systems)
   - LB:BP2OTS.OLB (on RSTS/E systems)

   Here, LB: is a RSTS/E logical name for the library account on disk.

   The BASIC–PLUS–2 object module libraries contain the Object Time System (OTS) files. If your system does not have memory-resident libraries, the Task Builder extracts all BASIC–PLUS–2 routines from the disk-resident object module libraries by default.
2. The library you specify with the DISKLIB command is included in all Task Builder command files until you either specify a new library with the DSKLIB command or exit from the BASIC–PLUS–2 environment. Once you exit from the BASIC–PLUS–2 environment, the default object module library set at installation is restored as the default disk-resident library.
3. To include the specified library in the Task Builder command file, you must use the DSKLIB command before you use the BUILD command.

4. You can override the DSKLIB command with the /DSKLIB qualifier to the BUILD command. The library you specify remains in effect for only that particular build operation.
5. If you specify a disk-resident library that is not available, the Task Builder signals an error message.
6. See your system manager for more information about the BASIC–PLUS–2 disk-resident libraries available on your system.
7. See the *BASIC–PLUS–2 User's Guide* for more information on object module libraries.

Examples

1.
```
! On RSX-11M/M-PLUS Systems
DSKLIB LB:[1,1]BP2OTS
```

2.
```
! On RSTS/E Systems
DSKLIB LB:BP2OTS
```

EDIT

The EDIT command allows you to edit individual program lines while in the BASIC–PLUS–2 environment by invoking an editor. EDIT with no parameters places you in the BASIC–PLUS–2 editing mode, where you can enter BASIC–PLUS–2 editing mode commands. The BASIC–PLUS–2 editing mode commands are described in C.

Format

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

search-clause: *delim unq-str1 delim*

replace-clause: *[unq-str2] delim [int-const]*

Syntax Rules

1. *Line-num* specifies the number of the line to be edited.
2. *Search-clause* specifies the text you want to remove or replace. *Unq-str1* is the search string you want to remove or replace.
3. *Replace-clause* specifies the replacement text and the occurrence of the search string you want to replace.
   - *Unq-str2* is the replacement string.
   - *Int-const* specifies the occurrence of *unq-str1* you want to replace. If you do not specify an occurrence, BASIC–PLUS–2 replaces the first occurrence of *unq-str1*.
4. *Delim* can be any printing character not used in *unq-str1* or *unq-str2*. The examples in this and the following sections use the slash (/) as a delimiter.
5. The *delim* characters in *search-clause* must match, or BASIC–PLUS–2 signals an error message.
6. The *delim* character you use to signal the end of *replace-clause* must match the *delim* you use in the *search-clause*, or BASIC–PLUS–2 does not signal an error and treats the end delimiter as part of *unq-str2*.
7. BASIC–PLUS–2 replaces or removes text in a program line as follows:
   - If *unq-str1* is found, BASIC–PLUS–2 replaces it with *unq-str2*.
   - If *unq-str1* is not found, BASIC–PLUS–2 signals an error.

- If *unq-str1* is null, BASIC–PLUS–2 replaces the first character of the last edited line with *unq-str2* and does not signal an error.
- If *unq-str2* is null, BASIC–PLUS–2 deletes *unq-str1*.
- BASIC–PLUS–2 matches and replaces strings exactly as you type them. If *unq-str1* is uppercase, BASIC–PLUS–2 searches for an uppercase string. If it is lowercase, BASIC–PLUS–2 searches for a lowercase string.

8. If you enter the EDIT command without an argument, BASIC–PLUS–2 places you in editing mode, where you can enter editing mode commands. The BASIC–PLUS–2 editing mode commands are as follows:
   - DEFINE
   - EXECUTE
   - EXIT or Ctrl/Z
   - FIND
   - INSERT
   - SUBSTITUTE

   See C for a description of the BASIC–PLUS–2 editing mode commands.
9. BASIC–PLUS–2 sets a specified line number as the current edit line, even when the editing operation fails. That line number remains set as the curent edit line until you specify another line number or exit from the BASIC–PLUS–2 environment.
10. You can edit a range of lines by separating two line numbers with a hyphen (–). BASIC–PLUS–2 signals an error and does not edit the specified range if there are spaces between the hyphen and the line numbers.
11. If you specify a range of lines and an occurence, BASIC–PLUS–2 replaces each occurrence of *unq-str1* in each line of the range beginning with the specified occurrence. For example:

```
10 PRINT DISPLAY$, DISPLAY$, DISPLAY$
20 PRINT DISPLAY$, DISPLAY$, DISPLAY$
EDIT 10-20 /DISPLAY$/NEW$/2

10 PRINT DISPLAY$, NEW$, NEW$
20 PRINT DISPLAY$, NEW$, NEW$

"DISPLAY$" replaced by "NEW$".
4 substitutions
```

Remarks

1. BASIC–PLUS–2 displays the edited line or lines with changes after the EDIT command successfully executes. It also displays a message showing the search string, replacement string, and number of replacements made.
2. If you want to edit a range of numbers, you must specify both the beginning and the end of the range. BASIC–PLUS–2 does not default to the last edited line or to the last line number in the program.
3. When you specify a line number with no text parameters, BASIC–PLUS–2 displays the message "Current edit line is x," where x is the specified line number.
4. When you type EDIT with no parameters, BASIC–PLUS–2 checks the last edited line number to make sure that it still exists in the current program. If it has been deleted, BASIC–PLUS–2 displays the error "?No current line."

Example

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

EXIT

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

Format

EXIT

Syntax Rules

None.

Remarks

If you forget to save or replace your program before attempting to exit from the BASIC–PLUS–2 environment, BASIC–PLUS–2 signals the warning "Unsaved change has been made, CTRL/Z or EXIT to exit." This message warns you that your program will be lost if you do not use either the SAVE or REPLACE command before exiting. If you do not save or replace the program and exit from the BASIC–PLUS–2 environment, the program changes are lost.

Example

```
EXIT
%Unsaved change has been made, CTRL/Z or EXIT to exit

BASIC2
```

EXTRACT

The EXTRACT command extracts a specified line or range of lines from the program currently in memory and deletes the remaining program lines.

Format

EXTRACT line-num [{ - , } line-num . . .]

Syntax Rules

1. The separator characters (comma or hyphen) allow you to extract individual lines or a range of lines while deleting all others. All extracted lines remain in memory.
2. If you separate line numbers with a comma (,), BASIC–PLUS–2 extracts each specified line number.
3. If you separate line numbers with a hyphen (-), BASIC–PLUS–2 extracts the inclusive range of lines. The lower line number must be specified first or the EXTRACT command has no effect.
4. You can combine individual line numbers and line ranges in a single EXTRACT command. Note, however, that a line number range must be followed by a comma and not another hyphen, or BASIC–PLUS–2 signals an error. The following example extracts lines 70 through 80, line 110, and line 124, and deletes the rest of the program.

```
EXTRACT 70-80, 110, 124
```

Remarks

1. If you do not specify a line number, BASIC–PLUS–2 signals an error.
2. BASIC–PLUS–2 signals an error if there are no lines in the specified range or if you specify an illegal line number.

Example

```
EXTRACT 300 - 1000
```

HELP

The HELP command displays online documentation for BASIC–PLUS–2 commands, qualifiers, statements, functions, and conventions.

Format

HELP [unq-str] . . .

Syntax Rules

1. *Unq-str* is a BASIC–PLUS–2 topic, qualifier, command, statement, function, or convention.
2. The first *unq-str* must be one of the topics described in the HELP file.
3. You can specify a subtopic after the topic. Separate one *unq-str* from another with a space.
4. You can use the asterisk (*) wildcard character in *unq-str*. BASIC–PLUS–2 then matches any portion of the specified topic.
5. If you type HELP with no parameters, BASIC–PLUS–2 displays a list of topics for you to choose from.

Remarks

1. If the *unq-str* you specify is not a unique topic or subtopic, BASIC–PLUS–2 displays information on all topics or subtopics beginning with *unq-str*.
2. An asterisk (*) indicates that you want to display information that matches any portion of the topic you specify. For example, if you type "Help statements GO*," BASIC–PLUS–2 displays information on the GOSUB statement and the GOTO statement.
3. When information on a particular topic or subtopic is not available, BASIC–PLUS–2 signals the message "Sorry, no help on that subject."
4. To exit from HELP, press the Return key until you get the BASIC2 prompt.

Example

```
BASIC2
Help statements GO* RET
STATEMENTS

  GOSUB

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

  Format

       { GO SUB }
       { GOSUB  } target

  Example

  200 GOSUB 1100

STATEMENTS
Press RETURN for more...

  GOTO

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

  Format

       { GO TO }
       { GOTO  } target

  Example

  20 GOTO 200

  Topic? RET
BASIC2
```

IDENTIFY

The IDENTIFY command displays an identification header on the controlling terminal. The header contains the name and version number of BASIC–PLUS–2.

Format

IDENTIFY

Syntax Rules

None.

Remarks

The message displayed by the IDENTIFY command includes the name of the BASIC–PLUS–2 compiler and the version number.

Example

```
IDENTIFY
PDP-11 BASIC-PLUS-2 V2.7-00

BASIC2
```

INQUIRE

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

LIBRARY

The LIBRARY command allows you to specify a BASIC–PLUS–2 memory-resident or user-created library to be used when you task-build the program. When you use the BUILD command, BASIC–PLUS–2 includes the specified library in the Task Builder command file. Your system manager selects the default library for the LIBRARY command when installing BASIC–PLUS–2.

Format

```
LIBRARY   [ lib-param ]

lib-param:  { file-spec }
            { lib-nam  }
            { NONE     }
```

Syntax Rules

1. *Lib-nam* and *file-spec* can be a memory-resident library supplied by BASIC–PLUS–2 or a user-created library.
2. If you specify a *lib-nam* with no device, the default device is LB: on RSTS/E systems and LB:[1,1] on RSX systems.
3. NONE tells the Task Builder not to link your task to the default memory-resident library. Therefore, the Task Builder links your task to the BASIC–PLUS–2 disk-resident object module library. The BASIC–PLUS–2 object module library is BP2OTS.OLB.
4. If you specify the LIBRARY command without a *lib-param*, BASIC–PLUS–2 prompts for one and displays the name of the current default memory-resident library. If you press the Return key in response to this prompt, the current default memory-resident library is used.

Remarks

1. The memory-resident libraries supplied by BASIC–PLUS–2 are as follows:
   - BP2RES
   - BP2SML

Because memory-resident libraries are optional, your system manager can select none, one, or both during the BASIC–PLUS–2 installation. See the *BASIC–PLUS–2 User's Guide* for information on using BASIC–PLUS–2 memory-resident libraries. See your system manager for information on the memory-resident libraries available on your system.

2. On RSTS/E systems, the LIBRARY command does not require the LB: logical name. BASIC–PLUS–2 automatically searches this account for the memory-resident library symbol table.
3. On RSX systems, the LIBRARY command automatically references libraries on LB:[1,1] unless you specify another device and directory.
4. To include the specified library in the Task Builder command file, you must use the LIBRARY command before you use the BUILD command.
5. The library you specify is included in all the Task Builder command files until you either specify a new library with the LIBRARY command or exit from the BASIC–PLUS–2 environment. When you exit from the BASIC–PLUS–2 environment, the memory-resident library set at installation is restored as the default.
6. You can override the LIBRARY command with the /LIBRARY qualifier to the BUILD command. The library you specify remains in effect for only that particular build operation.
7. If the specified library is not available, the Task Builder signals an error message.

Example

```
LIBRARY BP2RES
```

LIST and LISTNH

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

Format

LIST[NH] [[–] line-num [{ – , } line-num . . .]]

Syntax Rules

1. *Line-num* specifies a line number.
2. A hyphen (–) between the LIST command and the *line-num* displays all lines from the beginning of the program up to and including the line number you specify.
3. The separator characters (comma or hyphen) allow you to display individual lines or a block of lines.
   - A line number followed by a comma (,) or hyphen (–) and a carriage return, displays only the specified line.
   - If you separate line numbers with commas, BASIC–PLUS–2 displays each specified line number.
   - If you separate line numbers with hyphens, BASIC–PLUS–2 displays the inclusive range of lines. The lower line number must come first. If it does not, LIST has no effect.
   - If there are no lines in the specified range, BASIC–PLUS–2 signals an error.
   - You can combine individual line numbers and line ranges in a single LIST command. Note, however, that a line number range must be followed by a comma and not another hyphen, or BASIC–PLUS–2 signals an error.

Remarks

1. The LIST command without parameters displays the entire program.
2. The LIST command displays program lines, along with a header containing the program name, the current time, and the date. To suppress the program header, type LISTNH.
3. BASIC–PLUS–2 displays the source program lines in the order you specify in the command line. BASIC–PLUS–2 displays line 100 before line 10 if you type LIST 100,10.

Example

```
LIST 200-300

200 %IF %VARIANT = 2% %THEN %ABORT
250 %END %IF
300 PRINT A
```

LOAD

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

Format

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

Syntax Rules

File-spec must be the file specification of a BASIC–PLUS–2 object module or BASIC–PLUS–2 signals an error. OBJ is the default file type. If you specify only the file name, BASIC–PLUS–2 searches for an OBJ file in the current default directory.

Remarks

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

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

Example

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

LOCK

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

NEW

The NEW command clears BASIC–PLUS–2 memory and allows you to assign a name to a new program.

Format

NEW [prog-name]

Syntax Rules

1. *Prog-name* is the name of the program you want to create.
2. BASIC–PLUS–2 on RSX systems allows program names to contain a maximum of nine characters. If the program name exceeds nine characters, BASIC–PLUS–2 truncates the program name to nine characters and does not signal an error.
3. BASIC–PLUS–2 on RSTS/E systems allows program names to contain a maximum of six characters. If the program name exceeds six characters, BASIC–PLUS–2 truncates the program name to six characters and does not signal an error.
4. If you specify a file type with the program name, BASIC–PLUS–2 ignores the file type but does not signal an error.

Remarks

1. If you do not specify a *prog-name*, BASIC–PLUS–2 displays the following prompt:

   ```
   New file name--
   ```

   Enter a program name and press Return.
2. If you do not provide a program name BASIC–PLUS–2 assigns the file name NONAME to your program by default.

3. When you enter the NEW command, the program currently in memory is cleared. Program modules loaded with the LOAD command remain unchanged.

Example

```
NEW PROG1
```

ODLRMS

The ODLRMS command lets you specify an overlay description (ODL) file for the Task Builder to use when task-building your program. The ODL file describes how the Task Builder should overlay your task in memory. When you use the BUILD command, BASIC–PLUS–2 includes the specified ODL file in the Task Builder command file. Your system manager selects the default ODL file for your system during installation.

Format

ODLRMS [odl-param]

odl-param: { file-spec | NONE }

Syntax Rules

1. *File-spec* can be an ODL file supplied by RMS or a user-created file. Table 2–1 lists and describes the RMS ODL files.
2. NONE tells the Task Builder not to link your task to any RMS ODL file.
3. If you specify the ODLRMS command without an *odl-param*, BASIC–PLUS–2 prompts you for one and displays the name of the current default ODL file. If you press the Return key without specifying an ODL file, the current default ODL file is used.

Remarks

1. Because new versions of RMS can change ODL file names, consult the RMS distribution kit for current ODL names.
2. The default BASIC–PLUS–2 ODL files are usually located in the account LB:[1,1] on RSX systems and LB: on RSTS/E systems. (On RSTS/E systems, LB: is a logical name for the library account on disk.) See your system manager for more information.
3. The ODL file you specify is included in all Task Builder command files until you either specify a new ODLRMS command or exit from the BASIC–PLUS–2 environment. When you exit from the BASIC–PLUS–2 environment, the ODL file set during installation is restored as the default.

4. You can override the ODLRMS command with the /ODL qualifier to the BUILD command. When you use this qualifier, the ODL file you specify remains in effect for only that particular build operation.
5. See the description of the RMSRES command to see which ODL files are required for each RMS library.
6. If the ODL file you specify is unavailable, the Task Builder signals an error message.
7. See the *BASIC–PLUS–2 User's Guide* for more information on using the RMS libraries.

Table 2–1 Overlay Description Files

ODL File Name	File Organization			Type of Library	Overlay Segments
	Sequential	Relative	Indexed		
DAP11X	Yes	Yes	Yes	Disk	16
DAPRLX	Yes	Yes	Yes	Memory	None
RMS11S	Yes	Yes	No	Disk	11
RMS12S	Yes	Yes	No	Disk	5
RMS11X	Yes	Yes	Yes	Disk	35
RMS12X	Yes	Yes	Yes	Disk	13
RMSRLX	Yes	Yes	Yes	Memory	None

Examples

1.
```
! On RSX-11M/M-PLUS Systems
ODLRMS LB:[1,1]RMSRLX.ODL
```

2.
```
! On RSTS/E Systems
ODLRMS LB:RMSRLX.ODL
```

OLD

The OLD command brings a previously created BASIC–PLUS–2 program into memory.

Format

OLD [file-spec]

Syntax Rules

1. *File-spec* is the file specification of the program.
2. If you do not supply a file specification, BASIC–PLUS–2 prompts for one. If you do not enter a file specification in response to the prompt, BASIC–PLUS–2 searches for a file named NONAME.B2S in the current default directory.
3. If you do not specify a file type, the default file type is B2S.

Remarks

1. If the BASIC–PLUS–2 compiler cannot find the file you specify, BASIC–PLUS–2 signals the error "Can't find file or account."
2. When the specified file is found, it is placed in memory and any program currently in memory is erased. If BASIC–PLUS–2 does not find the specified file, the program currently in memory does not change.
3. If you specify a file that does not begin with a line number, BASIC–PLUS–2 discards all text up to the first line number, brings the file into memory, and signals the error "Illegal line format or missing continuation at line *n*." BASIC–PLUS–2 also signals this error if program text begins in column one without a line number.
4. If a file contains two lines with the same line number, BASIC–PLUS–2 signals the error "%Duplicate line number *n* encountered."
5. If a file contains program lines that are numbered out of sequential order, BASIC–PLUS–2 signals the warning error "%Line number *n* follows line number *n*." BASIC–PLUS–2 includes the line, but places it in ascending numeric order.
6. If a program contains more than 32767 characters associated with a single line number, BASIC–PLUS–2 signals the error "?Line too long."

7. On RSX systems, if a program contains more than 132 characters on a single line, BASIC–PLUS–2 signals the error "?Line too long."
8. On RSTS/E systems, if a program contains more than 256 characters on a single line, BASIC–PLUS–2 signals the error "?Line too long or invalid file format."
9. A source file you bring into memory with the OLD command must be a RSTS/E native-mode file or BASIC–PLUS–2 signals the error "Line too long or invalid file format." If the file you want to bring into memory is an RMS file, you can convert it before bringing it into memory by using the PIP command with the /RMS qualifier.

Example

```
OLD CHECK

BASIC2
```

RENAME

The RENAME command allows you to assign a new name to the program currently in memory. BASIC–PLUS–2 does not write the renamed program to a file until you save the program with the REPLACE or SAVE command.

Format

RENAME [prog-name]

Syntax Rules

1. *Prog-name* specifies the new program name.
2. BASIC–PLUS–2 on RSX systems allows program names to contain a maximum of nine characters. If the program name exceeds nine characters, BASIC–PLUS–2 truncates the program name to nine characters and does not signal an error.
3. BASIC–PLUS–2 on RSTS/E systems allows program names to contain a maximum of six characters. If the program name exceeds six characters, BASIC–PLUS–2 truncates the program name to six characters and does not signal an error.
4. If you do not specify a new program name with the RENAME command, BASIC–PLUS–2 prompts you for one. If you do not specify a program name in response to the prompt, the name of the program currently in memory remains unchanged.
5. If you specify a file type, BASIC–PLUS–2 ignores the file type, does not signal an error, and assigns the B2S file type to the file when you save it.

Remarks

1. You must enter SAVE or REPLACE to write the renamed program to a file. If you do not enter SAVE or REPLACE, BASIC–PLUS–2 does not save the renamed program.
2. The RENAME command does not affect the original saved version of the program.

Example

```
OLD TEST
BASIC2

RENAME NEWTEST
BASIC2

LIST
NEWTEST   29-APR-1991 13:50
PRINT "This program is a simple test"
   .
   .
   .

BASIC2

SAVE

BASIC2
```

REPLACE

The REPLACE command writes the current program to a storage medium.

Format

REPLACE [file-spec]

Syntax Rules

1. *File-spec* is the file specification of the program.
2. If you do not supply a *file-spec*, BASIC–PLUS–2 writes the program to the default disk with the file name of the program currently in memory.
   - BASIC–PLUS–2 on RSX systems creates and saves a new version of the file, incrementing the version number by one. Previous versions of the file remain unchanged.
   - BASIC–PLUS–2 on RSTS/E systems overwrites the original version of the file with the new version.

Remarks

1. If you specify a file specification, it does not have to match the file specification of the program currently in memory. You can differentiate a changed program from the original version by specifying a new file specification.
2. The program currently in memory does not change.

Example

```
REPLACE PROGA.NEW
```

RMSRES

The RMSRES command allows you to specify an RMS memory-resident library for the Task Builder to use when task-building your program. An RMS library supplies RMS code for file and record operations. When you use the BUILD command, BASIC–PLUS–2 includes the specified library in the Task Builder command file. Your system manager selects the default RMS library for your system during installation.

Format

RMSRES lib-param

lib-param: { file-spec / NONE }

Syntax Rules

1. *File-spec* can either be an RMS memory-resident library or a user-created resident library. Table 2–2 lists the RMS libraries.
2. NONE tells the Task Builder not to link your task to the default RMS resident library. Therefore the Task Builder links your task to the RMS object module library RMSLIB.OLB.
3. If you do not supply a *lib-param*, BASIC–PLUS–2 prompts for one and displays the name of the current default RMS library. If you press the Return key in response to this prompt, the current default memory-resident library is used.

Remarks

1. On RSX systems, the RMS libraries are usually located in LB:[1,1]. On RSTS/E systems, the RMS libraries are usually located on device LB:. LB: is a logical name for the library account on disk. See your system manager for more information on the location of the RMS libraries.
2. To include the specified library in the Task Builder command file, you must use the RMSRES command before you use the BUILD command.
3. If you use an RMS library other than the default, you must specify one of the RMS ODL files listed in Table 2–2. See the description of the ODLRMS command for more information.

4. The RMSRES library you specify is included in all the Task Builder command files until you either specify a new RMSRES library with the RMSRES command or exit from the BASIC–PLUS–2 environment. When you exit from the BASIC–PLUS–2 environment, the RMS library set at installation is restored as the default.
5. You can override the RMSRES command with the /RMSRES qualifier to the BUILD command. The specified library remains in effect for only that particular build operation.
6. If the specified library is not available, the Task Builder signals an error message.
7. See the *BASIC–PLUS–2 User's Guide* for more information on using RMS libraries.

Table 2–2 RMS–11 Libraries

Library Name	File Organization			Type of Library	ODL File Required
	Sequential	Relative	Indexed		
DAPRES	Yes	Yes	Yes	Memory	DAPRLX.ODL
RMSLIB	Yes	Yes	Yes	Disk	RMS11S.ODL RMS12S.ODL RMS11X.ODL RMS12X.ODL DAP11X.ODL
RMSRES	Yes	Yes	Yes	Memory	RMSRLX.ODL

Examples

1.
```
! On RSX-11M/M-PLUS Systems
RMSRES LB:[1,1]RMSRES
```

2.
```
! On RSTS/E Systems
RMSRES LB:RMSRES
```

RUN

The RUN command allows you to execute a program from the BASIC–PLUS–2 environment without first invoking the PDP–11 Task Builder to construct an executable image. Support for the RUN command is an installation option. Use the SHOW command to see whether your system supports the RUN command. The RUNNH command is identical to RUN, except that it does not display the program header, current date, and time.

Format

RUN[NH] [file-spec] [/qualifier] . . .

Command Qualifiers	Defaults
/BYTE	/WORD
/[NO]CHAIN	See text.
/[NO]DEBUG	/NODEBUG
/DOUBLE	/SINGLE
/[NO]FLAG [sep [NO]DECLINING]	/FLAG=DECLINING
/[NO]LINE	/LINE
/LONG	/WORD
/[NO]SCALE sep const	/NOSCALE
/SINGLE	/SINGLE
/[NO]SYNTAX_CHECK	/NOSYNTAX_CHECK
/TYPE_DEFAULT sep default-clause	/TYPE_DEFAULT=REAL
/VARIANT sep int-const	/VARIANT=0
/WORD	/WORD

Syntax Rules

1. *File-spec* is the file specification of the program you want to execute.
2. If you do not supply a *file-spec*, BASIC–PLUS–2 executes the program currently in memory.
3. If you specify only a file name, BASIC–PLUS–2 searches for a file with a B2S file type in the current default directory.
4. */Qualifier* specifies a qualifier that sets a BASIC–PLUS–2 default.

Remarks

1. BASIC–PLUS–2 signals the warning message "No main program" if you do not have a main program in memory or do not specify the file specification of a main program with the RUN command.
2. When you specify a file specification with the RUN command, BASIC–PLUS–2 brings the program into memory and then executes it. You do not have to bring a program into memory with the OLD command to run it. After program execution is complete, the program remains in memory.
3. If your program calls a subprogram, the subprogram must be compiled and placed in memory with the LOAD command. If your program calls a subprogram that has not been compiled and loaded, BASIC–PLUS–2 signals an error.
4. The RUN command does not create an object module file or a list file.
5. The RUN command executes a program starting at the lowest line number.
6. When BASIC–PLUS–2 encounters a STOP statement, the program stops executing.
   - If you used the RUN command to execute the program, BASIC–PLUS–2 displays a number sign (#) prompt. You can then either enter the debugger command CONTINUE to resume program execution, or EXIT to end the program.
   - If you used the RUN command with the /DEBUG qualifier to execute the program, control passes to the BASIC–PLUS–2 debugger. You can then use the BASIC–PLUS–2 debugger commands to display and change program values and to analyze your program. When you are finished debugging your program, enter the debugger command CONTINUE to resume program execution. See B for a description of the BASIC–PLUS–2 debugger commands.
7. When you exit from the BASIC–PLUS–2 environment, all options set with qualifiers return to the system default values. Use the SHOW command to display your system defaults before setting any qualifiers.

Command Qualifiers

/BYTE

The /BYTE qualifier causes BASIC–PLUS–2 to allocate eight bits of storage as the default for all integer data not explicitly typed in the program. Untyped integer values are treated as BYTE values and must be in the BYTE range or

BASIC–PLUS–2 signals the error "Integer error or overflow." Table 1-2 in this manual lists BASIC–PLUS–2 data types and ranges. The default is /WORD.

/[NO]CHAIN
The /CHAIN qualifer can be used on RSTS/E systems only. The /CHAIN qualifier enables other programs to CHAIN into the program using the LINE clause of the CHAIN statement. If the program has more than 200 line numbers, the /NOCHAIN qualifier reduces the memory needs of the output program by disabling storage of line numbers in memory. You cannot chain from one DECNET node to another. The default is determined at installation.

/[NO]DEBUG
The /DEBUG qualifier appends to the object file information on symbolic references and line numbers. This information is used by the BASIC–PLUS–2 Debugger to debug your program. You must specify the /LINE qualifier when you specify the /DEBUG qualifier on the COMPILE command; otherwise, BASIC–PLUS–2 signals an error.

When you specify /DEBUG, control is passed to the debugger when the program is executed in the BASIC–PLUS–2 environment. If you specify /NODEBUG, information on program symbols and line numbers is not included in the object file and control is not passed to the debugger when the program executes. The default is /NODEBUG.

See the *BASIC–PLUS–2 User's Guide* for more information on using the BASIC–PLUS–2 Debugger.

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

/[NO]FLAG { = : } [NO]DECLINING

The /FLAG qualifier causes BASIC–PLUS–2 to provide compile-time information about program elements that are not recommended for new program development.

For example, if you specify the DECLINING clause, BASIC–PLUS–2 flags the following source code as declining:

- CVT$$ (use EDIT$)

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

The default is /FLAG=DECLINING.

/[NO]LINE
The /LINE qualifier includes line number information in object modules. If you specify /NOLINE, BASIC–PLUS–2 does not include line number information in object modules. If you specify /NOLINE in a program containing the run-time ERL function, BASIC–PLUS–2 issues a warning that the /NOLINE qualifier has been overridden. The default is /LINE.

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

/[NO]SCALE { = : } const

The /SCALE qualifier allows control of accumulated round-off errors when double-precision numbers (values typed DOUBLE) are used. Numbers are stored as multiples of 10 by setting *const* (the scale factor) from 0 through 6. A scale factor larger than 6 causes BASIC–PLUS–2 to signal the error message "Scale factor out of range–ignored." /NOSCALE is the default.

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

/[NO]SYNTAX_CHECK
The /SYNTAX_CHECK qualifier causes BASIC–PLUS–2 to perform syntax checking after each program line is entered. If you specify /NOSYNTAX_CHECK, BASIC–PLUS–2 does not perform syntax checking. The default is /NOSYNTAX_CHECK.

/TYPE_DEFAULT { = | : } { **REAL** | **INTEGER** | **EXPLICIT** }

The /TYPE_DEFAULT qualifier sets the default data type (REAL or INTEGER) for all data not explicitly typed in your program or specifies that all data must be explicitly typed (EXPLICIT).

- REAL specifies that all data not explicitly typed is floating-point data of the default size (SINGLE or DOUBLE).
- INTEGER specifies that all data not explicitly typed is integer data of the default size (BYTE, WORD, or LONG).
- EXPLICIT specifies that all data in a program must be explicitly typed. Implicitly declared variables cause BASIC–PLUS–2 to signal an error.

The default is TYPE_DEFAULT=REAL.

/VARIANT { = | : } **int-const**

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

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

Example

```
RUN PROG1
PROG1   29-APR-1991 13:52

 1
 3
 6
 10
BASIC2

RUNNH PROG1
 1
 3
 6
 10
BASIC2
```

SAVE

The SAVE command writes the BASIC–PLUS–2 source program currently in memory to a file on the default or specified device.

Format

SAVE [file-spec]

Syntax Rules

1. *File-spec* is the file specification of the program you want to save.
2. If you do not supply a file specification, BASIC–PLUS–2 saves the file with the name of the program currently in memory and a file type of B2S.
3. If you specify only the file name, BASIC–PLUS–2 saves the program with the default file type in the current default directory.

Remarks

1. If you specify a file specification and the file already exists, BASIC–PLUS–2 signals the warning "File exists–rename or replace."
2. BASIC–PLUS–2 stores the sorted program in ascending line number order.
3. You can store the program on a specified device. For example:

   ```
   SAVE DB1:[4,5]NEWTEST.PRO
   ```

   BASIC–PLUS–2 saves the file NEWTEST.PRO in DB1:[4,5].

Example

```
SAVE PROG_SAMP.B2S
```

SCALE

The SCALE command allows you to control accumulated round-off errors by multiplying DOUBLE numeric values by 10 raised to the scale factor, truncating them to an INTEGER value and then storing them.

Format

SCALE int-const

Syntax Rules

1. *Int-const* specifies the power of 10 you want to use as the scaling factor.
2. *Int-const* can be an integer from 0 through 6.
3. If the specified value is greater than 6, BASIC–PLUS–2 signals the error "Scale factor of *n* is out of range," where *n* is the specified value.
4. If you do not supply an *int-const*, BASIC–PLUS–2 signals the error "Illegal number."

Remarks

1. SCALE affects only values of the data type DOUBLE.
2. BASIC–PLUS–2 multiplies values using the scale factor you specify and then truncates the value at the decimal point. For example:

```
10  DECLARE DOUBLE X
    X = "2.488888"
    PRINT USING "#.#######";X
```

The value 2.488888 is rounded as follows:

Scale	Value Produced for 2.488888
0	2.488888
1	2.4
2	2.48
3	2.488

Scale	Value Produced for 2.488888
4	2.4888
5	2.48888
6	2.488888

3. The SCALE command does not improve accuracy; however, it does provide near exact accuracy for the number of digits specified.
4. The SCALE command influences the representation of DOUBLE numeric values in output files that were written using MOVE statements, MAP /COMMON statements, or virtual arrays. If a program compiled without SCALE (or a different SCALE factor) accesses such data written by a SCALE-set program, the data values will be skewed by the power of 10 specified as the scaling factor. You must use consistent scaling amoung programs that share scaled data.

Example

```
SCALE 2
```

SCRATCH

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

Format

SCRATCH

Syntax Rules

None.

Remarks

None.

Example

```
SCRATCH
```

SEQUENCE

The SEQUENCE command causes BASIC–PLUS–2 to automatically generate line numbers for your program text. BASIC–PLUS–2 supplies line numbers for your text until you end the procedure or reach the maximum line number of 32767.

Format

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

Syntax Rules

1. *Line-num* specifies the line number where sequencing begins.
2. If you do not specify a line number, the default is line number 100.
3. *Int-const* specifies the line number increment for your program. If you do not specify an increment, the default is 10.

Remarks

1. If you specify a line number that already contains a statement, or if the sequencing operation generates a line number that already contains a statement, BASIC–PLUS–2 signals the error "Attempt to sequence over existing statement," and returns to normal input mode.
2. Enter your program text in response to the line number prompt; the carriage return ends each line and causes BASIC–PLUS–2 to generate a new line number.
3. If you press Ctrl/Z in response to the line number prompt, BASIC–PLUS–2 terminates the sequencing operation and prompts for another command.
4. When the maximum line number of 32767 is reached, BASIC–PLUS–2 terminates the sequencing process and returns to normal input mode.
5. BASIC–PLUS–2 does not check syntax during the sequencing process.

Example

```
BASIC2

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

SET

The SET command allows you to specify BASIC–PLUS–2 defaults for all BASIC–PLUS–2 qualifiers. Qualifiers control the compilation process and the run-time environment. The defaults you set remain in effect for all subsequent operations until they are reset or until you exit from the compiler.

Format

SET [/qualifier] . . .

Command Qualifiers	Defaults
/BYTE	/WORD
/[NO]CHAIN	See text.
/[NO]CLUSTER[sep lib-param]	/NOCLUSTER
/[NO]CROSS_REFERENCE [sep [NO]KEYWORDS]	/NOCROSS_REFERENCE
/[NO]DEBUG	/NODEBUG
/DOUBLE	/SINGLE
/[NO]DUMP	/NODUMP
/EXTEND sep int-const	/EXTEND=512
/[NO]FLAG sep [NO]DECLINING	/FLAG=DECLINING
/[NO]IDS	/NOIDS
/[NO]INDEX	/NOINDEX
/[NO]LINE	/LINE
/[NO]LIST	/NOLIST
/LONG	/WORD
/[NO]MACRO	/NOMACRO
/[NO]OBJECT	/OBJECT
/PAGE_SIZE sep int-const	/PAGE_SIZE=60
/[NO]RELATIVE	/NORELATIVE
/[NO]SEQUENTIAL	/NOSEQUENTIAL
/SINGLE	/SINGLE
/[NO]SYNTAX_CHECK	/NOSYNTAX_CHECK
/TYPE_DEFAULT sep default-clause	/TYPE_DEFAULT=REAL
/VARIANT sep int-const	/VARIANT=0
/[NO]VIRTUAL	/NOVIRTUAL
/[NO]WARNINGS	/WARNINGS
/WIDTH sep int-const	/WIDTH=132
/WORD	/WORD

Syntax Rules

1. /*Qualifier* specifies a qualifier keyword that sets a BASIC–PLUS–2 default.
2. BASIC–PLUS–2 signals the error "Illegal switch" if you do not separate multiple qualifiers with commas (,) or slashes (/). The same error is signaled if you separate qualifiers with a slash but do not prefix the first qualifier with a slash.

Remarks

If you do not specify any qualifiers, BASIC–PLUS–2 resets all qualifiers except those set with the /BRLRES, /DSKLIB, /LIBRARY, /ODLRMS, /RMSRES, or /EXTEND qualifiers to the installation defaults. The SCALE value set with the SCALE command is also not reset to the installation default.

Command Qualifiers

/BYTE

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

/[NO]CHAIN

The /CHAIN qualifer can be used on RSTS/E systems only. The /CHAIN qualifier enables other programs to CHAIN into the program using the LINE clause of the CHAIN statement. If the program has more than 200 line numbers, the /NOCHAIN qualifier reduces the memory needs of the output program by disabling storage of line numbers in memory. You cannot chain from one DECNET node to another. The default is determined at installation.

/[NO]CLUSTER [{ **=** } **file-spec** **]**
{ **:** } **NONE**

The /CLUSTER qualifer causes the Task Builder to cluster memory-resident libraries to increase the space available for your task. Before you use the /CLUSTER qualifier, at least two memory-resident libraries must be linked to the task: the BASIC–PLUS–2 memory-resident library, and one other memory-resident library.

- *File-spec* specifies the memory-resident library to be clustered. The specified library must be in the account LB: on RSTS/E systems or the account LB:[1,1] on RSX systems.

- *NONE* specifies that only the BASIC–PLUS–2 and RMS–11 libraries are clustered.
- /CLUSTER with no argument causes the Task Builder to cluster the default memory-resident library. If there is no default cluster library, /CLUSTER with no argument acts the same as /CLUSTER:NONE.

The /NOCLUSTER qualifier tells the Task Builder not to cluster memory-resident libraries to increase the space available for your task. /NOCLUSTER is the default. See the *BASIC–PLUS–2 User's Guide* for more information on using RMS–11 libraries.

/[NO]CROSS_REFERENCE[{ = } [NO]KEYWORDS]
{ : }

If you use the /CROSS_REFERENCE qualifier with the /LIST qualifier when you compile your program, the BASIC–PLUS–2 compiler includes cross-reference information in the program listing file. If you specify /CROSS_REFERENCE=KEYWORDS, BASIC–PLUS–2 also cross-references BASIC–PLUS–2 keywords used in the program. If you specify /NOCROSS_REFERENCE, BASIC–PLUS–2 does not include a cross reference section in the compiler listing. The default is /NOCROSS_REFERENCE.

/[NO]DEBUG

The /DEBUG qualifier appends to the object file information on symbolic references and line numbers. This information is used by the BASIC–PLUS–2 Debugger to debug your program. You must specify the /LINE qualifier when you specify the /DEBUG qualifier on the COMPILE command; otherwise, BASIC–PLUS–2 signals an error.

When you specify /DEBUG, control is passed to the debugger when the program is executed in the BASIC–PLUS–2 environment. If you specify /NODEBUG, information on program symbols and line numbers is not included in the object file and control is not passed to the debugger when the program executes. The default is /NODEBUG.

See the *BASIC–PLUS–2 User's Guide* for more information on using the BASIC–PLUS–2 Debugger.

/DOUBLE

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

/[NO]DUMP
If your program aborts with a fatal error, the /DUMP qualifier causes the Task Builder to generate a memory dump. The /NODUMP qualifier causes the Task Builder not to generate a memory dump when the program aborts. The default is /NODUMP.

/EXTEND { = | : } int-const

The /EXTEND qualifier specifies the amount of space to be added to the initial task size when the task is started. The Task Builder rounds the extension up to the nearest 32-word boundary. The maximum allowed extension is 32000. The default is /EXTEND=512.

/[NO]FLAG { = | : }[NO]DECLINING

The /FLAG qualifier causes BASIC–PLUS–2 to provide compile-time information about program elements that are not recommended for new program development.

For example, if you specify the DECLINING clause, BASIC–PLUS–2 flags the following source code as declining:

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

The default is /FLAG=DECLINING.

/[NO]IDS
The /IDS qualifier causes the Task Builder to build your task with I- and D-space support. I- and D-space provides a faster program execution time and also allows you to execute larger programs than usual. It is recommended that you do not use the BASIC–PLUS–2 memory-resident library in an I- and D-space task. By default, when you specify the /IDS qualifier, BASIC–PLUS–2 removes all references to the BASIC–PLUS–2 memory-resident library from the Task Builder command file. The /NOIDS qualifier tells the Task Builder not to build the task with I- and D-space support. /NOIDS is the default. See the *BASIC–PLUS–2 User's Guide* for more information on enabling I- and D-space for program execution.

[NO]INDEX
The /INDEX qualifier causes the Task Builder to include the code needed for indexed file operations. BASIC–PLUS–2 enables this qualifier automatically for programs containing an OPEN statement with the ORGANIZATION INDEXED clause. If you specify /NOINDEX, the Task Builder does not include the code needed for indexed file operations. The default is /NOINDEX.

/[NO]LINE
The /LINE qualifier includes line number information in object modules. If you specify /NOLINE, BASIC–PLUS–2 does not include line number information in object modules. If you specify /NOLINE in a program containing the run-time ERL function, BASIC–PLUS–2 issues a warning that the /NOLINE qualifier has been overridden. The default is /LINE.

/[NO]LIST
The /LIST qualifier causes BASIC–PLUS–2 to produce a compiler listing file. By default, the compiler listing generated by the /LIST qualifier contains a memory allocation map. The name of the listing file is the same as the name of the first program module specified, or the name of the program currently in memory if no file specification is provided. The listing file has a default file type of LST. If you specify /NOLIST, BASIC–PLUS–2 does not generate a compiler listing. /NOLIST is the default.

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

/[NO]MACRO
The /MACRO qualifier converts the program into MACRO–11 source code and saves it in a file with the same name as the program and a file type of MAC. A MACRO–11 file can be assembled. If you specify /NOMACRO, a MACRO–11 source code file is not generated. You cannot specify the /OBJECT qualifier with the /MACRO qualifier. The default is /NOMACRO.

/[NO]OBJECT
The /OBJECT qualifier generates an object module with the same file name as that of the program and a default file type of OBJ. The /NOOBJECT qualifier allows you to check your program for errors without creating an object file. If your program contains one or more fatal errors, an object module is

not generated. You cannot specify the /MACRO qualifier with the /OBJECT qualifier. /OBJECT is the default.

/PAGE_SIZE { = : } **int-const**

The /PAGE_SIZE qualifier sets the page size of the listing file. *Int-const* must be greater than zero or BASIC–PLUS–2 signals the warning message "Listing length out of range—ignored." The default is /PAGE_SIZE=60.

/[NO]RELATIVE

The /RELATIVE qualifier causes the Task Builder to include the code needed for relative file operations. BASIC–PLUS–2 sets this qualifier automatically for programs containing an OPEN statement with the ORGANIZATION RELATIVE clause. If you specify /NORELATIVE, the Task Builder does not include the code necessary for relative file operations. /NORELATIVE is the default.

/[NO]SEQUENTIAL

The /SEQUENTIAL qualifier causes the Task Builder to include the RMS–11 code needed for sequential file operations. BASIC–PLUS–2 sets this qualifier automatically for programs containing an OPEN statement with the ORGANIZATION SEQUENTIAL clause. If you specify /NOSEQUENTIAL, the Task Builder does not include the RMS–11 code necessary for sequential file operations. /NOSEQUENTIAL is the default.

/SINGLE

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

/[NO]SYNTAX_CHECK

The /SYNTAX_CHECK qualifier causes BASIC–PLUS–2 to perform syntax checking after each program line is entered. If you specify /NOSYNTAX_CHECK, BASIC–PLUS–2 does not perform syntax checking. The default is /NOSYNTAX_CHECK.

/TYPE_DEFAULT { = : } { **REAL** **INTEGER** **EXPLICIT** }

The /TYPE_DEFAULT qualifier sets the default data type (REAL or INTEGER) for all data not explicitly typed in your program or specifies that all data must be explicitly typed (EXPLICIT).

- REAL specifies that all data not explicitly typed is floating-point data of the default size (SINGLE or DOUBLE).
- INTEGER specifies that all data not explicitly typed is integer data of the default size (BYTE, WORD, or LONG).
- EXPLICIT specifies that all data in a program must be explicitly typed. Implicitly declared variables cause BASIC–PLUS–2 to signal an error.

The default is TYPE_DEFAULT=REAL.

/VARIANT { = | : } int-const

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

/[NO]VIRTUAL

The /VIRTUAL qualifier causes BASIC–PLUS–2 to include the RMS code needed for virtual array and block I/O file operations. BASIC–PLUS–2 sets this qualifier automatically when you compile a program containing an OPEN statement with an ORGANIZATION VIRTUAL clause. If you specify /NOVIRTUAL, BASIC–PLUS–2 does not include the RMS code necessary for virtual array and block I/O file operations. /NOVIRTUAL is the default.

/[NO]WARNINGS

The /WARNINGS qualifier causes BASIC–PLUS–2 to display warning messages during program compilation. The /NOWARNINGS qualifier causes BASIC–PLUS–2 to disable warning messages during program compilation. The default is /WARNINGS.

/WIDTH { = | : } int-const

The /WIDTH qualifier sets the width of the listing file. *Int-const* must be an integer from 72 through 132, or BASIC–PLUS–2 signals the message "Listing width out of range—ignored." The default is /WIDTH=132.

/WORD

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

Example

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

On the SET command, the slash character can be omitted from a qualifier. However, qualifiers must be separated by either slashes or commas. The following is equivalent to the previous example:

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

In all other commands, the slash character is always required to denote a qualifier.

SHOW

The SHOW command displays the current defaults for the BASIC–PLUS–2 compiler on your terminal.

Format

SHOW

Syntax Rules

None.

Remarks

None.

Example

```
SHOW
PDP-11 BASIC-PLUS-2 V2.7-00 using EIS with run support

ENVIRONMENT INFORMATION:              RMS FILE ORGANIZATION:
    Current edit line : 0                NO Index
    NO Modules loaded                    NO Relative
    NO Main module loaded                NO Sequential
                                         NO Virtual

DEFAULT DATA TYPE INFORMATION:       LISTING FILE INFORMATION:
    Data type : REAL                     NO Source
    Real size : SINGLE                   NO Cross Reference
    Integer size : WORD                  NO Keywords
    Scale factor : 0                        60 lines by 132 columns

COMPILATION QUALIFIERS:              BUILD QUALIFIERS:
       Object                            NO Dump
    NO Macro                             NO Map
       Lines                             NO Cluster
       Warnings                          NO I- and D-Space
    NO Debug records                     Task extend  : 512
    NO Syntax checking                   RMS ODL file : LB:[1,1]RMS11X
    Flag : Declining                     BP2 Disk lib : LB:[1,1]V24OTS
    Variant : 0                          RMS Resident lib : NONE
                                         BP2 Resident lib : NONE
BASIC2
```

UNSAVE

The UNSAVE command deletes a specified file from storage.

Format

UNSAVE [file-spec]

Syntax Rules

File-spec is the file specification of a program.

Remarks

1. If you supply only a file name, BASIC–PLUS–2 deletes the file with the specified name and a file type of B2S.
2. If you do not supply a file specification, BASIC–PLUS–2 deletes the file that has the file name of the program currently in memory and a file type of B2S.
3. If you do not supply a file specification and do not have a program in memory, BASIC–PLUS–2 searches for the default file NONAME.B2S.

Example

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

3

Compiler Directives

A compiler directive is an instruction that causes BASIC–PLUS–2 to perform a certain operation as it translates the source program. This chapter describes the compiler directives supported by BASIC–PLUS–2. Each compiler directive is listed and discussed alphabetically.

%ABORT

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

Format

%ABORT [str-lit]

Syntax Rules

1. Only a line number or a comment field can appear on the same physical line as the %ABORT directive.
2. *Str-lit* is the error message text. It must be a string literal enclosed in quotation marks.

Remarks

When BASIC–PLUS–2 encounters a %ABORT directive, it stops the compilation and terminates the listing file if a listing file has been requested. If you specified an error message, BASIC–PLUS–2 displays the message text on your terminal screen and in the compilation listing.

Example

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

%CROSS

The %CROSS directive causes BASIC–PLUS–2 to resume accumulating cross-reference information for the listing file which was previously suspended by the %NOCROSS directive.

Format

%CROSS

Syntax Rules

Only a line number or a comment field can appear on the same physical line as the %CROSS directive.

Remarks

1. The %CROSS directive has no effect unless you request a cross-reference section for the compilation listing with the /CROSS_REFERENCE qualifier. For more information on listing file format, see the *BASIC–PLUS–2 User's Guide*.
2. When a cross-reference section is requested, the BASIC–PLUS–2 compiler resumes accumulating cross-reference information immediately after encountering the %CROSS directive.

Example

```
10  %CROSS
```

%IDENT

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

Format

%IDENT str-lit

Syntax Rules

1. Only a line number or a comment field can appear on the same physical line as an %IDENT directive.
2. *Str-lit* is the identification text. It must be a string literal enclosed in quotation marks.
   - The identification text can consist of up to six RAD-50 characters.
   - If the identification text contains more than six RAD-50 characters, BASIC–PLUS–2 signals a warning message and truncates the extra characters.
   - If the identification text contains characters other than RAD-50 characters, BASIC–PLUS–2 signals a warning message and the %IDENT directive is ignored.

Remarks

1. The BASIC–PLUS–2 compiler inserts the identification text in the first 31 character positions of the second line on each listing page. BASIC–PLUS–2 also includes the identification text in the object module, if the compilation produces one, and in the map file created by the Task Builder, if a map file is requested.
2. The %IDENT directive should appear at the beginning of your program if you want the identification text to appear on the first page of your listing. If the %IDENT directive appears after the first program statement, the text will appear on the next page of the listing file.
3. You can use the %IDENT directive only once in a module. If you specify more than one %IDENT directive in a module, BASIC–PLUS–2 signals a warning and uses the identification text specified in the first directive.

4. The default identification text is a 6-digit number. The first two digits represent the compiler base level, while the last four digits represent the month and day. For example, the identification text 100712 represents base level 10, and a date of July 12.

Example

```
40  %IDENT "V10"
    .
    .
    .
```

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

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

Format

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

Syntax Rules

1. *Lex-exp* is always a WORD integer.
2. *Lex-exp* can be any of the following:
   - A lexical constant named in a %LET directive.
   - An integer literal, with or without the percent sign suffix.
   - A lexical built-in function (%VARIANT).
   - Any combination of the above, separated by valid lexical operators. Lexical operators include logical operators, relational operators, and the arithmetic operators for addition (+), subtraction (–), multiplication (*), and division (/).
3. *Code* is BASIC–PLUS–2 program code. It can be any BASIC–PLUS–2 statement or another compiler directive, including another %IF directive. You can nest %IF directives to eight levels.
4. The %IF directive can appear anywhere in a program where a space is allowed, except in column one or within a quoted string. This means that you can use the %IF directive to make a whole statement, part of a statement, or a block of statements conditional.
5. %THEN, %ELSE, and %END %IF do not have to be on the same physical line as %IF.

Remarks

1. If *lex-exp* is true, BASIC–PLUS–2 processes the %THEN clause. If *lex-exp* is false, BASIC–PLUS–2 processes the %ELSE clause. If there is no %ELSE clause, BASIC–PLUS–2 processes the %END %IF clause. The BASIC–PLUS–2 compiler includes statements in the %THEN or %ELSE clause in the source program and executes directives in order of occurrence.

2. You must include the %END %IF clause. Otherwise, BASIC–PLUS–2 assumes the remainder of the program is part of the last %THEN or %ELSE clause and signals the error "Missing %END %IF" when compilation ends.

Example

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

%INCLUDE

The %INCLUDE directive lets you include BASIC–PLUS–2 source text from another program file in the current program compilation.

Format

%INCLUDE str-lit

Syntax Rules

1. Only a line number or a comment field can appear on the same physical line as the %INCLUDE directive.
2. *Str-lit* specifies the file to be included. It must be a string literal enclosed in quotation marks.
3. If you do not specify a complete file specification, BASIC–PLUS–2 uses the default device and directory and the file type B2S.

Remarks

1. Any statement that appears after an END statement inside an included file causes BASIC–PLUS–2 to signal an error.
2. The BASIC–PLUS–2 compiler includes the specified source file in the program compilation at the point of the %INCLUDE directive and prints the included code in the program listing file if the compilation produces one.
3. The included file cannot contain line numbers. If it does, BASIC–PLUS–2 signals the error "Line number may not appear in %INCLUDE file."
4. All statements in the accessed file are associated with the line number of the program line that contains the %INCLUDE directive.
5. A file accessed by %INCLUDE can itself contain a %INCLUDE directive.
6. All %IF directives in an included file must have a matching %END %IF directive or BASIC–PLUS–2 signals the error "IF directive in INCLUDE directive needs END IF directive in same file."

Example

```
100  %INCLUDE "CHECKIT"
```

%LET

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

Format

```
%LET   %lex-var = lex-exp
```

Syntax Rules

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

Remarks

You cannot change the value of a lexical variable within a program unit once it has been named in a %LET directive. For more information on coding conventions see the *BASIC–PLUS–2 User's Guide*.

Example

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

%LIST

The %LIST directive causes the BASIC–PLUS–2 compiler to resume accumulating compilation information for the program listing file which was previously suspended by the %NOLIST directive.

Format

%LIST

Syntax Rules

Only a line number or a comment field can appear on the same physical line as the %LIST directive.

Remarks

1. The %LIST directive has no effect unless you requested a listing file. For more information on listing file format, see the *BASIC–PLUS–2 User's Guide*.
2. As soon as it encounters the %LIST directive, the BASIC–PLUS–2 compiler resumes accumulating information for the program listing file. Thus, the directive itself appears as the next line in the listing file.

Example

```
100  %LIST
```

%NOCROSS

The %NOCROSS directive causes the BASIC–PLUS–2 compiler to stop accumulating cross-reference information for the program listing file.

Format

%NOCROSS

Syntax Rules

Only a line number or a comment field can appear on the same physical line as the %NOCROSS directive.

Remarks

1. The BASIC–PLUS–2 compiler stops accumulating cross-reference information for the program listing file immediately after encountering the %NOCROSS directive.
2. The %NOCROSS directive has no effect unless you request a listing file and cross-reference information.
3. It is recommended that you not embed a %NOCROSS directive within a statement. Embedding a %NOCROSS directive within a statement makes the accumulation of cross-reference information unpredictable. For more information on listing file format, see the *BASIC–PLUS–2 User's Guide*.

Example

```
100  %NOCROSS
```

%NOLIST

The %NOLIST directive causes the BASIC–PLUS–2 compiler to stop accumulating compilation information for the program listing file.

Format

%NOLIST

Syntax Rules

Only a line number or a comment field can appear on the same physical line as the %NOLIST directive.

Remarks

1. As soon as it encounters the %NOLIST directive, the BASIC–PLUS–2 compiler stops accumulating information for the program listing file. Thus, the directive itself does not appear in the listing file.
2. The %NOLIST directive has no effect unless you requested a listing file.
3. For more information on the listing file format, see the *BASIC–PLUS–2 User's Guide*.

Example

```
100  %NOLIST
```

%PAGE

The %PAGE directive causes BASIC–PLUS–2 to begin a new page in the program listing file immediately after the line that contains the %PAGE directive.

Format

%PAGE

Syntax Rules

Only a line number or a comment field can appear on the same physical line as the %PAGE directive.

Remarks

The %PAGE directive has no effect unless you request a listing file.

Example

```
100  %PAGE
```

%PRINT

The %PRINT directive lets you insert a message into your source code that the BASIC–PLUS–2 compiler prints during compilation.

Format

%PRINT str-lit

Syntax Rules

1. Only a line number or a comment field can appear on the same physical line as the %PRINT directive.
2. *Str-lit* is the message text. It must be a string literal enclosed in quotation marks.

Remarks

1. %PRINT displays an informational error message containing text that you specify.
2. The message text you specify is displayed on the terminal screen. If you request a listing, the message appears in the compilation listing as well.

Example

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

The output is:

```
Error on line 10,
          %PRINT "This is a debug compilation"
          User PRINT message:  This is a debug compilation
```

%SBTTL

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

Format

%SBTTL str-lit

Syntax Rules

1. Only a line number or a comment field can appear on the same physical line as the %SBTTL directive.
2. *Str-lit* is the subtitle text. It must be a string literal enclosed in quotation marks and have no more than 48 characters.
3. If you specify more than 48 characters in the subtitle, BASIC–PLUS–2 truncates the extra characters but does not signal a warning or error.

Remarks

1. The specified subtitle appears under the title of all pages of source code in the listing file until the BASIC–PLUS–2 compiler encounters another %SBTTL or %TITLE directive. BASIC–PLUS–2 clears the subtitle field before the allocation map section of the listing is generated. This way, you only get a subtitle on the listing pages that contain source code.
2. Because BASIC–PLUS–2 associates a subtitle with a title, a new %TITLE directive sets the current subtitle to the null string. In this case, no subtitle appears in the listing until BASIC–PLUS–2 encounters another %SBTTL directive.
3. If you want a subtitle to appear on the first page of your listing, the %SBTTL directive should appear at the beginning of your program, immediately after the %TITLE directive. Otherwise, the subtitle will start to appear only on the second page of the listing.
4. If you want the subtitle to appear on the page of the listing that contains the %SBTTL directive, the %SBTTL directive should follow a %PAGE directive, or follow a %TITLE directive which follows a %PAGE directive.
5. The %SBTTL directive has no effect unless you request a listing file.

Example

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

%TITLE

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

Format

%TITLE str-lit

Syntax Rules

1. Only a line number or a comment field can appear on the same physical line as the %TITLE directive.
2. *Str-lit* is the title text. It must be a string literal enclosed in quotation marks and can have no more than 48 characters.
3. If you specify more than 48 characters in the title, BASIC–PLUS–2 truncates the extra characters but does not signal a warning or error.

Remarks

1. The specified title appears on the first line of every page of the listing file until BASIC–PLUS–2 encounters another %TITLE directive in the program.
2. The %TITLE directive should appear on the first line of your program, before the first statement, if you want the specified title to appear on the first page of your listing.
3. If you want the specified title to appear on the page that contains the %TITLE directive, the %TITLE directive should immediately follow a %PAGE directive.
4. Because BASIC–PLUS–2 associates a subtitle with a title, a new %TITLE directive sets the current subtitle to the null string.
5. The %TITLE directive has no effect unless you request a listing file.

Example

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

%VARIANT

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

Format

%VARIANT

Syntax Rules

The %VARIANT function can appear only in a lexical expression.

Remarks

The %VARIANT function returns the integer value specified with /VARIANT qualifier to the COMPILE, SET, or RUN commands. The returned integer always has a data type of WORD.

Example

```
40  %LET %RSTS = 2
50  %LET %RSX = 1
60  %LET %VAX = 0

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

4

Statements and Functions

This chapter provides reference material on all of the BASIC–PLUS–2 statements and functions. The statements and functions are described in alphabetical order and each statement or function begins on a separate page.

ABS

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

Format

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

Syntax Rules

None.

Remarks

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

Example

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

The output is:

```
39             512.73
```

ABS%

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

Format

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

Syntax Rules

None.

Remarks

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

Example

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

The output is:

```
39            512
```

ASCII

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

Format

int-var = { **ASC** / **ASCII** } (str-exp)

Syntax Rules

None.

Remarks

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

Example

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

The output is

```
70
```

ATN

The ATN function returns the angle of a specified tangent in radians.

Format

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

Syntax Rules

None.

Remarks

1. ATN returns a value from –PI/2 through PI/2.
2. The argument of the ATN function must be a real expression. When the argument is a real expression, BASIC–PLUS–2 returns a value of the same floating-point size. When the argument is not a real expression, BASIC–PLUS–2 converts the argument to the default floating-point size and returns a value of the default floating-point size.

Example

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

The output is:

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

BUFSIZ

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

Format

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

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number. You cannot precede the *chnl-exp* with a number sign (#).
2. The value assigned to *int-var* is a WORD integer.

Remarks

1. If the specified channel is closed, BUFSIZ returns a value of zero.
2. BUFSIZ of channel #0 always returns the current terminal width or, in a batch stream, returns a value of 512.

Example

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

The output is:

```
Buffer size equals 132
```

CALL

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

Format

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

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

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

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

Syntax Rules

1. *Routine* is the name of a BASIC–PLUS–2 SUB subprogram or another callable program module. It cannot be a variable name.

   Note

   Although you can call routines written in other languages, BASIC–PLUS–2 supports calls only to BASIC–PLUS–2 routines.

2. *Pass-mech* specifies how arguments are passed to a non-BASIC routine.
   - Specify a BY REF *pass-mech* after the routine name if you are invoking a MACRO subprogram. If you invoke a MACRO subprogram but do not specify BY REF, errors in the MACRO subprogram will not be handled correctly.

- Do not specify a *pass-mech* if you are invoking a BASIC–PLUS–2 routine. If you specify a *pass-mech* when invoking a BASIC–PLUS–2 routine, BASIC–PLUS–2 does not save the BASIC–PLUS–2 internal variables and the results are unpredictable.

3. If you do not specify a *pass-mech*, BASIC–PLUS–2 passes arguments as indicated in Table 4–1.
   - BY VALUE specifies that BASIC–PLUS–2 passes the argument's 16-bit value. Only BYTE and WORD values can be passed BY VALUE. BYTE values passed BY VALUE are converted to WORD values.
   - BY REF specifies that BASIC–PLUS–2 passes the argument's address. This is the default for all arguments except strings and entire arrays.
   - BY DESC specifies that BASIC–PLUS–2 passes the address of a BASIC–PLUS–2 descriptor. You can pass only string values and entire arrays by descriptor. See the *BASIC–PLUS–2 User's Guide* for a description of BASIC–PLUS–2 descriptors.

Table 4–1 BASIC–PLUS–2 Parameter-Passing Mechanisms

Parameter	BY VALUE	BY REF	BY DESC
	Integer and Real Data		
Variables	Yes[2]	Yes[1]	No
Constants	Yes[2]	Local copy[1]	No
Expressions	Yes[2]	Local copy[1]	No
Elements of a nonvirtual array	Yes[2]	Local copy[1]	No
Virtual array elements	Yes[2]	Local copy[1]	No
Nonvirtual entire array	No	Yes	Yes[1]
Virtual entire array	No	No	Yes

[1]Specifies the default parameter-passing mechanism.

[2]Two asterisks indicate that the value can have 16 bits, at most.

(continued on next page)

Table 4–1 (Cont.) BASIC–PLUS–2 Parameter-Passing Mechanisms

Parameter	BY VALUE	BY REF	BY DESC
	String Data		
Variables	No	Yes	Yes[1]
Constants	No	Local copy	Local copy[1]
Expressions	No	Local copy	Local copy[1]
Nonvirtual array elements	No	Local copy	Local copy
Virtual array elements	No	Local copy	Local copy[1]
Nonvirtual entire arrays	No	Yes	Yes[1]
Virtual entire arrays	No	No	Yes[1]
	Other Parameters		
RFA variables	No	Yes[1]	No

[1]Specifies the default parameter-passing mechanism.

4. If you call a non-BASIC routine and a passing mechanism appears before the parameter list, it applies to all arguments passed to the called routine and the routine is called with the R5 (fifth register) calling sequence. You can override this passing mechanism by specifying a *pass-mech* for individual arguments in the *actual-param* list.
5. *Actual-param* lists the arguments to be passed to the called routine.
6. You can pass expressions or entire arrays. Optional commas in parentheses after the array name specify the dimensions of the array. The number of commas is equal to the number of dimensions –1. Thus, no comma specifies a one-dimensional array, one comma specifies a two-dimensional array, two commas specify a three-dimensional array, and so on.

7. You cannot pass entire virtual arrays. Instead, share the data in a virtual array between a calling program and a subprogram by opening a virtual file in either program and dimensioning the array (using the same channel number) in both programs.
8. The name of the routine can consist of 1 through 6 characters and must conform to the following rules:
   - The first character of an unquoted name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), or periods (.).
   - A quoted name can consist of any combination of alphabetic characters, digits, dollar signs ($), periods (.), or spaces.
   - The routine can be a BASIC–PLUS–2 subprogram or a subprogram written in another language.
   - BASIC–PLUS–2 allows you to pass up to 32 parameters to a BASIC–PLUS–2 subprogram and up to 255 parameters to a MACRO–11 subprogram.

Remarks

1. BASIC–PLUS–2 does not allow recursion. That is, once a subprogram is called, it cannot be called again until the SUBEND or SUBEXIT statement for that routine has executed or until an error has been trapped with ON ERROR GO BACK.
2. You can specify a null argument as an *actual-param* for non-BASIC routines by omitting the argument and the *pass-mech*, but not the commas or parentheses. This forces BASIC–PLUS–2 to pass a null argument and allows you to access system routines from BASIC–PLUS–2.
3. Arguments in the *actual-param* list must agree in data type and number with the formal parameters specified in the subprogram.
4. Modifiable parameters are parameters passed by reference or descriptor. An argument is modifiable when changes to it are evident in the calling program. Changing a modifiable parameter in a subprogram means the parameter is changed for the calling program as well. Only variables and entire arrays passed by descriptor or by reference are modifiable.

5. Nonmodifiable parameters are parameters passed by value or by reference with a local copy. An argument is nonmodifiable when changes to it are not evident in the calling program. Changing a nonmodifiable argument in a subprogram does not affect the value of that argument in the calling program. Arguments passed by value, constants, and expressions are nonmodifiable. Passing an argument as an expression (by placing it in parentheses) changes it from a modifiable to a nonmodifiable argument. Virtual array elements passed as parameters are non-modifiable. For further information, refer to the section in the *BASIC–PLUS–2 User's Guide*, "Passing Parameters to a BASIC–PLUS–2 Subprogram."
6. BASIC–PLUS–2 automatically converts numeric actual parameters to match the data type declared by the EXTERNAL statement for the routine being called. If the actual parameter is a variable, BASIC–PLUS–2 signals the informational message "Mode for parameter <n> of routine <name> changed to match declaration" and passes the argument by local copy. This prevents the called routine from modifying the contents of the variable.
7. For expressions and virtual array elements passed by reference, BASIC–PLUS–2 makes a local copy of the value, and passes the address of this local copy. For dynamic string arrays, BASIC–PLUS–2 passes a descriptor of the array of string descriptors. The compiler passes the address of the argument's actual value for all other arguments passed by reference.
8. If you attempt to call an external function, BASIC–PLUS–2 treats the function as if it were invoked normally and validates all parameters. Note that you cannot call a STRING or RFA function. See the EXTERNAL statement for more information on how to invoke functions.
9. Files are not closed when the CALL statement executes.

Example

```
10  EXTERNAL SUB OUTPUT (string)
    DECLARE STRING msg_str
    msg_str = "Successful call to OUTPUT!"
    CALL OUTPUT (msg_str)
```

The output is:

```
Successful call to OUTPUT!
```

CAUSE ERROR

The CAUSE ERROR statement allows you to artificially generate a BASIC–PLUS–2 run-time error and transfer program control to a BASIC–PLUS–2 error handler.

Format

CAUSE ERROR err-num

Syntax Rules

Err-num must be a valid BASIC–PLUS–2 run-time error number.

Remarks

See the *BASIC–PLUS–2 User's Guide* for a list of errors and their corresponding numbers.

Example

```
10  End_of_file_error = 11%
    CAUSE ERROR End_of_file_error
    .
    .
    .
3200    SELECT ERR
            CASE = 11
                 PRINT "End of file"
                 RESUME 32767
            CASE ELSE
                 RESUME 32767
        END SELECT
```

CCPOS

The CCPOS function returns the current character position or cursor position of the output record on a specified channel.

Format

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

Syntax Rules

Chnl-exp is a numeric expression that specifies a channel number of an open file or terminal. You cannot precede the *chnl-exp* with a number sign (#).

Remarks

1. If *chnl-exp* is zero, CCPOS returns the current character position of the controlling terminal.
2. The *int-var* returned by the CCPOS function is of the default integer size.
3. The CCPOS function counts only characters. If you use cursor addressing sequences such as escape sequences, the value returned will not be the cursor position.
4. The first character position on a line is zero.

Example

```
10 DECLARE LONG curs_pos
   PRINT "Hello";
   curs_pos = CCPOS (0)
   PRINT curs_pos
```

The output is:

```
Hello 5
```

CHAIN

The CHAIN statement transfers control from the current program to another executable image. CHAIN closes all files, then requests that the new program begin execution. Control does not return to the original program when the new image finishes executing.

Note

The CHAIN statement is not recommended for new program development. It is recommended that you use subprograms and external functions for program segmentation.

There are two formats. The first format is for BASIC–PLUS–2 on RSX–11M and RSX–11M–PLUS systems. The second format is for BASIC–PLUS–2 on RSTS/E systems.

Format

1. For RSX–11M/M–PLUS Systems:

CHAIN str-exp

2. For RSTS/E Systems:

CHAIN str-exp [**LINE** lin-num]

Syntax Rules

1. *Str-exp* represents the file specification of the program to which control is passed. It can be a quoted or unquoted string.
2. On RSTS/E systems, *lin-num* specifies a line number in another BASIC–PLUS–2 program. The line number must exist and be in the range from 1 through 32767. The program containing the line number must have been compiled with the /CHAIN qualifier.

Remarks

1. *Str-exp* must refer to an executable image or BASIC–PLUS–2 signals an error.
2. If you do not specify a file type, BASIC–PLUS–2 searches for a file type of TSK.
3. You cannot chain to a program on another node.
4. Execution starts at the first line number of the specified program.
5. On RSTS/E systems, if you specify a *lin-num*, execution starts at the specified line number.
6. On RSTS/E systems, if you specify a line number and the line number does not exist, BASIC–PLUS–2 signals an error.
7. Before chaining takes place, all active output buffers are written, all open files are closed, and all storage is released. On RSTS/E systems, the last buffer (512 bytes) of a terminal-format file does not get written unless the file is closed before the CHAIN statement executes.
8. Because a CHAIN statement passes control from the executing image, the values of any program variables are lost. This means that you can pass parameters to a chained program only by using files or a system-specific feature such as GET/PUT Core Common on RSTS/E systems.
9. See the *BASIC–PLUS–2 User's Guide* for information about how the CHAIN statement is implemented on your system.

Examples

1.
```
10  CHAIN PROG2
    .
    .
    .
100 CHAIN "PROG5.COM"
```

2.
```
190 !BASIC--PLUS--2 on RSTS/E systems only
200 CHAIN "NUMBER.TSK" LINE 390
```

CHANGE

The CHANGE statement either converts a string of characters to an array of their ASCII integer values or converts an array of numbers to a string of ASCII characters.

The first format converts a string variable to an array. The second format converts an array to a string variable.

Format

1. String Variable to Array

CHANGE str-exp **TO** num-array-name

2. Array to String Variable

CHANGE num-array-name **TO** str-var

Syntax Rules

1. *Str-exp* is a string expression.
2. *Num-array-name* should be a one-dimensional array. If you specify a two-dimensional array, BASIC–PLUS–2 converts only the first row of that array. BASIC–PLUS–2 does not support conversion to or from arrays of more than two dimensions.

Remarks

1. String Variable to Array
   - This format converts each character in the string to its ASCII value.
   - BASIC–PLUS–2 assigns the value of the string's length to the first element of the array.
   - BASIC–PLUS–2 assigns the ASCII value of the first character in the string to the second element, (1) or (0,1), of the array, the ASCII value of the second character to the third element, (2) or (0,2), and so on.
   - If the string is longer than the bounds of the array, BASIC–PLUS–2 does not translate the excess characters and signals the error "Subscript out of range" (ERR=55). The first element of the array still contains the length of the string.

2. Array to String Variable

   - This format converts the elements of the array to a string of characters.
   - The length of the string is determined by the value in the zero element, (0) or (0,0), of the array. If the value of element zero is greater than the array bounds, BASIC–PLUS–2 signals the error "Subscript out of range" (ERR=55).
   - BASIC–PLUS–2 changes the first element, (1) or (0,1), of the array to its ASCII character equivalent, the second element, (2) or (0,2), to its ASCII equivalent, and so on. The length of the returned string is determined by the value in the zero element of the array. For example, if the array is dimensioned as (10), but the zero element (0) contains the value 5, BASIC–PLUS–2 changes only elements (1), (2), (3), (4), and (5) to string characters.
   - BASIC–PLUS–2 truncates floating-point values to integers before converting them to characters.
   - Values in array elements are treated modulo 256.

Example

```
10  DECLARE STRING ABCD, A
    DIM INTEGER array_changes(6)
    ABCD = "ABCD"
    CHANGE ABCD TO array_changes
    FOR I% = 0 TO 4
    PRINT array_changes(I%)
    NEXT I%
    CHANGE array_changes TO A
    PRINT A
```

The output is:

```
 4
 65
 66
 67
 68
ABCD
```

CHR$

The CHR$ function returns a 1-character string that corresponds to the ASCII value you specify.

Format

str-var = **CHR$**(int-exp)

Syntax Rules

None.

Remarks

1. CHR$ returns the character whose ASCII value equals *int-exp*. If *int-exp* is greater than 255, BASIC–PLUS–2 treats it modulo 256. For example, CHR$(325) is the same as CHR$(69).
2. All arguments between 0 and 255 are considered unsigned 8-bit integers. For example, –1 is treated as 255.
3. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to an integer of the default size.

Example

```
10  DECLARE INTEGER num_exp
    INPUT "Enter the ASCII value you wish to be converted";num_exp
    PRINT "The equivalent character is ";CHR$(num_exp)
```

The output is:

```
Enter the ASCII value you wish to be converted? 89
The equivalent character is Y
```

CLOSE

The CLOSE statement ends I/O processing to a device or file on the specified channel.

Format

CLOSE [#]chnl-exp, . . .

Syntax Rules

Chnl-exp is a numeric expression that specifies a channel number associated with a file. It can be preceded by an optional number sign (#).

Remarks

1. BASIC–PLUS–2 writes the contents of any active output buffers to the file or device before it closes that file or device.
2. Channel #0 (the controlling terminal) cannot be closed. An attempt to do so has no effect.
3. If you close a magnetic tape file that is open for output, BASIC–PLUS–2 writes an end-of-file on the magnetic tape at the current tape position.
4. If you try to close a channel that is not currently open, BASIC–PLUS–2 does not signal an error and the CLOSE statement has no effect.

Example

```
10  OPEN "COURSE.DAT" FOR INPUT AS #2
    INPUT #2, course_nam, course_num, course_desc, course_instr
    .
    .
    .
    CLOSE #2
```

COMMON

The COMMON statement defines a named, shared storage area called a COMMON block or program section (PSECT). BASIC–PLUS–2 program modules can access the values stored in the COMMON block by specifying a COMMON block with the same name.

Format

{ **COMMON** / **COM** } [(com-name)] {[data-type] com-item}, . . .)

com-item: { *num-unsubs-var* / *num-array-name (int-const, . . .)* / *str-unsubs-var = int-const* / *str-array-name (int-const, . . .) [= int-const]* / ***FILL*** *[(int-const)][= int-const]* / ***FILL%*** *[(int-const)]* / ***FILL$*** *[(int-const)][= int-const]* }

Syntax Rules

1. A COMMON block can have the same name as that of a program variable, but cannot have the same name as that of a subprogram within the same task image.
2. A COMMON block and a map in the same program module cannot have the same name.
3. All COMMON elements must be separated with commas.
4. *Com-name* names the COMMON. *Com-name* is optional. If you specify a common name, it must be in parentheses. If you do not specify a common name, the default is ".$$$$.".
5. *Com-name* can be from 1 through 6 characters. The first character of the name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), or periods (.).
6. *Data-type* can be any BASIC–PLUS–2 data type keyword. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.

7. When you specify a data type, all following *com-items*, including FILL items, are of that data type until you specify a new data type.
8. If you do not specify any data type, *com-items* take the current default data type and size.
9. *Com-item* declares the name and format of the data to be stored.
   - *Num-unsubs-var* and *num-array-name* specify a numeric variable or a numeric array.
   - *Str-unsubs-var* and *str-array-name* specify a fixed-length string variable or array. You can specify the number of bytes to be reserved for the variable with the =*int-const* clause. The default string length is 16.
   - The FILL, FILL%, and FILL$ keywords allow you to reserve parts of the record buffer within or between data elements and to define the format of the storage. *Int-const* specifies the number of FILL items to be reserved. The =*int-const* clause allows you to specify the number of bytes to be reserved for string FILL items. Table 4–2 describes FILL item format and storage allocation.

Note

In the applicable formats of FILL, *(int-const)* represents a repeat count, not an array subscript. FILL *(n)* represents *n* elements, not *n* + 1.

Table 4–2 FILL Item Formats and Storage Allocations

FILL Format	Storage Allocation
FILL	Allocates storage for one element of the default data type unless preceded by a *data-type*; the number of bytes allocated depends on the default or the specified data type.
FILL(int-const)	Allocates storage for the number of floating-point elements specified by *int-const* unless preceded by a *data type*; the number of bytes allocated for each element depends on the default floating-point data size or the specified *data type*.
FILL%	Allocates storage for one integer element; the number of bytes allocated depends on the default integer size.
FILL%(int-const)	Allocates storage for the number of integer elements specified by *int-const*; the number of bytes allocated for each element depends on the default integer size.
FILL$	Allocates 16 bytes of storage for a string element. The dollar sign can be omitted if the FILL keyword is preceded by the STRING data type.
FILL$(int-const)	Allocates 16 bytes of storage for the number of string elements specified by *int-const*. The dollar sign can be omitted if the FILL keyword is preceded by the STRING data type.

(continued on next page)

Table 4–2 (Cont.) FILL Item Formats and Storage Allocations

FILL Format	Storage Allocation
FILL$=int-const	Allocates the number of bytes of storage specified by *int-const* for a string element. The dollar sign can be omitted if the FILL keyword is preceded by the STRING data type.
FILL$(int-const1)=int-const2	Allocates the number of bytes of storage specified by *int-const2* for the number of string elements specified by *int-const1*. The dollar sign can be omitted if the FILL keyword is preceded by the STRING *data type*.

Remarks

1. You should know how your program overlays memory if the data stored in a COMMON area is to be shared by several program modules. The COMMON should be named in an overlay unit that will remain in memory as long as program units need to reference the COMMON data. If the overlay that names the COMMON is forced out of memory, BASIC–PLUS–2 reinitializes the COMMON area to zero when the overlay is brought back into memory. See the *BASIC–PLUS–2 User's Guide* for information on overlay structures.
2. A COMMON area and a MAP area with the same name, in different program modules, specify the same storage area.
3. BASIC–PLUS–2 does not execute COMMON statements. The COMMON statement allocates and defines the data storage area at compilation time.
4. When you link your program, the size of the COMMON area is the size of the largest COMMON area with that name. BASIC–PLUS–2 concatenates COMMON statements with the same *com-name* within a single program module into a single PSECT. The total space allocated is the sum of the space allocated in the concatenated COMMON statements.

   If you specify the same *com-name* in several program modules, the size of the PSECT will be determined by the program module that has the greatest amount of space allocated in the concatenated COMMON statements.

5. The COMMON statement must lexically precede any reference to variables declared in it.
6. A COMMON area can be accessed by more than one program module, as long as you define the *com-name* in each module that references the COMMON area.
7. Variable names in a COMMON statement in one program module need not match those in another program module.
8. Variables and arrays declared in a COMMON statement cannot be declared elsewhere in the program by any other declarative statements.
9. The data type specified for *com-items* or the default data type and size determines the amount of storage reserved in a COMMON block:
   - BYTE integers reserve 1 byte.
   - WORD integers reserve 2 bytes.
   - LONG integers reserve 4 bytes.
   - SINGLE floating-point numbers reserve 4 bytes.
   - DOUBLE floating-point numbers reserve 8 bytes.
   - STRING reserves 16 bytes (the default) or the number of bytes you specify with *=int-const*.
10. For multi-dimensional arrays, values are assigned in row-column order.

Example

```
10 COMMON (sales) INTEGER shelf_number,          &
                    STRING row = 2,              &
                           report_name = 24      &
                    DOUBLE FILL,                 &
                    LONG part_bins
```

COMP%

The COMP% function compares two numeric strings and returns a –1, 0, or 1, depending on the results of the comparison.

Format

int-var = **COMP%**(str-exp1, str-exp2)

Syntax Rules

Str-exp1 and *str-exp2* are numeric strings. They can contain up to 60 ASCII digits, an optional minus sign (–), and an optional decimal point (.).

Remarks

1. If *str-exp1* is greater than *str-exp2*, COMP% returns a 1.
2. If the string expressions are equal, COMP% returns a 0.
3. If *str-exp1* is less than *str-exp2*, COMP% returns a –1.
4. The value returned by the COMP% function is an integer of the default size.

Example

```
10  DECLARE STRING num_string, old_num_string, &
          INTEGER result
    num_string = "-24.5"
    old_num_string = "33"
    result = COMP%(num_string, old_num_string)
    PRINT "The value is ";result
```

The output is:

```
The value is -1
```

COS

The COS function returns the cosine of an angle in radians.

Format

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

Syntax Rules

None.

Remarks

1. The returned value is between –1 and 1. This value is expressed in radians.
2. BASIC–PLUS–2 expects the argument of the COS function to be a real expression. When the argument is a real expression, BASIC–PLUS–2 returns a value of the same floating-point size. When the argument is not a real expression, BASIC–PLUS–2 converts the argument to the default floating-point size and returns a value of the default floating-point size.

Example

```
10  DECLARE SINGLE cos_value
    cos_value = 26
    PRINT COS(cos_value)
```

The output is:

```
.646919
```

CTRLC

The CTRLC function enables Ctrl/C trapping. When Ctrl/C trapping is enabled, a Ctrl/C entered at the terminal causes control to be transferred to the error handler currently in effect.

Format

int-var = **CTRLC**

Syntax Rules

None.

Remarks

1. When BASIC–PLUS–2 encounters a Ctrl/C, control passes to the error handler currently in effect. If there is no error handler in a program, the program aborts.
2. Ctrl/C trapping is asynchronous; that is, BASIC–PLUS–2 suspends execution and signals the error "Programmable ^C trap" (ERR=28), as soon as it detects a Ctrl/C. Consequently, a statement can be interrupted while it is executing. A statement so interrupted may be only partially completed and variables may be left in an undefined state.
3. BASIC–PLUS–2 can trap more than one Ctrl/C error in a program as long as the error does not occur while the error handler is executing. If a second Ctrl/C is detected while the error handler is processing the first Ctrl/C, the program aborts.
4. On RSX systems, the task that contains the CTRLC function must be able to attach to a terminal as soon as the CTRLC fucntion is enabled. If another task is attached to the terminal, the task that enabled the CTRLC function terminates with a directive error.
5. The CTRLC function always returns a value of zero.

Example

```
10  ON ERROR GOTO 19000
    Y% = CTRLC
    .
    .
    .
    IF (ERR=28)
         RESUME Ctrlc_handler
    END IF
```

CVT$$

The CVT$$ function is a synonym for the EDIT$ function. See the EDIT$ function for more information.

Note

The CVT$$ function is supported only for compatibility with BASIC-PLUS. It is recommended that you use the EDIT$ function for new program development.

Format

str-var = **CVT$$**(str-exp, int-exp)

CVTxx

The CVT$% function maps the first two characters of a string into a 16-bit integer. The CVT%$ function translates a 16-bit integer into a 2-character string. The CVT$F function maps a 4- or 8-character string into a floating-point variable. The CVTF$ function translates a floating-point number into a 4- or 8-byte character string. The number of characters translated depends on whether the floating-point variable is single- or double-precision.

Note

The CVTxx function is supported only for compatibility with BASIC-PLUS. It is recommended that you use BASIC–PLUS–2 dynamic mapping or multiple MAP statements for new program development.

Format

int-var = **CVT$%**(str-var)

real-var = **CVT$F**(str-var)

str-var = **CVT%$**(int-var)

str-var = **CVTF$**(real-var)

Syntax Rules

None.

Remarks

1. **CVT$%**
   - If the CVT$% *str-var* has fewer than two characters, BASIC–PLUS–2 pads the string with nulls.
   - The value returned by the CVT$% function is an integer of the default size.
2. **CVT%$**
   - Only two bytes of data are inserted into *str-var*.

- If you specify a floating-point variable for *int-var*, BASIC–PLUS–2 truncates it to an integer of the default size. If the default size is BYTE and the value of *int-var* exceeds 127, BASIC–PLUS–2 signals an error.

3. **CVT$F**
   - CVT$F maps four characters when the program is compiled with /SINGLE and eight characters when the program is compiled with /DOUBLE.
   - If *str-var* has fewer than four or eight characters, BASIC–PLUS–2 pads the string with nulls.
   - The *real-var* returned by the CVT$F function is the default floating-point size.
4. **CVTF$**
   - The CVTF$ function maps single-precision numbers to a 4-character string and double-precision numbers to an 8-character string.
   - BASIC–PLUS–2 expects the argument of the CVTF$ function to be a real expression. When the argument is a real expression, BASIC–PLUS–2 returns a value of the same floating-point size. When the argument is not a real expression, BASIC–PLUS–2 converts the argument to the default floating-point size and returns a value of the default floating-point size.

Examples

1.
```
10  DECLARE STRING test_string, another_string
    DECLARE WORD first_number, next_number
    test_string = "AT"
    PRINT CVT$%(test_string)
    another_string = "at"
    PRINT CVT$%(another_string)
    first_number = 16724
    PRINT CVT%$(first_number)
    next_number = 24948
    PRINT CVT%$(next_number)
    END
```

The output is:

```
 16724
 24948
AT
at
```

2.
```
10  DECLARE STRING test_string, another_string
    DECLARE SINGLE first_num, second_num
    test_string = "DESK"
    first_num = CVT$F(test_string)
    PRINT first_num
    another_string = "desk"
    second_num = CVT$F(another_string)
    PRINT second_num
    PRINT CVTF$(first_num)
    PRINT CVTF$(second_num)
    END
```

The output is:

```
 .218256E+12
 .466242E+31
DESK
desk
```

DATA

The DATA statement creates a data block for the READ statement.

Format

DATA [num-lit | str-lit | unq-str] , . . .

Syntax Rules

1. *Num-lit* specifies a numeric literal.
2. *Str-lit* is a character string that starts and ends with double or single quotation marks. The quotation marks must match.
3. *Unq-str* is a character sequence that does not start or end with double quotation marks and does not contain a comma.
4. Commas separate data elements. If a comma is part of a data item, the entire item must be enclosed in quotation marks.

Remarks

1. Because BASIC–PLUS–2 treats comment fields in DATA statements as part of the DATA sequence, you should not include comments.
2. A DATA statement must be the last or the only statement on a physical line.
3. DATA statements must end with a line terminator.
4. When a DATA statement is continued with an ampersand (&), BASIC–PLUS–2 interprets all characters between the keyword DATA and the ampersand as part of the data. Any code that appears on a noncontinued line is considered a new statement.
5. You cannot use the percent sign suffix for integer constants that appear in DATA statements. An attempt to do so causes BASIC–PLUS–2 to signal the error "Data format error" (ERR=50).
6. DATA statements are local to a program module.
7. BASIC–PLUS–2 does not execute DATA statements. Instead, control is passed to the next executable statement.

8. A program can have more than one DATA statement. BASIC–PLUS–2 assembles data from all DATA statements in a single program unit into a lexically ordered single data block.
9. BASIC–PLUS–2 ignores leading and trailing blanks and tabs unless they are in a string literal.
10. Commas are the only valid data delimiters. You must use a quoted string literal if a comma is to be part of a string.
11. BASIC–PLUS–2 ignores DATA statements without an accompanying READ statement.
12. BASIC–PLUS–2 signals the error "Data format error" (ERR=50) if the DATA item does not match the data type of the variable specified in the READ statement or if a data element that is to be read into an integer variable ends with a percent sign (%). If a string data element ends with a dollar sign ($), BASIC–PLUS–2 treats the dollar sign as part of the string.

Example

```
10 DECLARE INTEGER A,B,C
   READ A,B,C
   DATA 1,2,3
   PRINT A + B + C
```

The output is:

```
6
```

DATE$

The DATE$ function returns a string containing a day, month, and year in the form *dd-Mmm-yy*.

Format

str-var = **DATE$**(int-exp)

Syntax Rules

1. *Int-exp* can have up to six digits in the form *yyyddd*, where the characters *yyy* specify the number of years since 1970 and the characters *ddd* specify the day of that year.
2. You must fill all three of the *d* positions with digits or zeros before you can fill the *y* positions. For example:
   - DATE$(121) returns the date 01–May–70, day 121 of the year 1970.
   - DATE$(1201) returns the date 20–Jul–71, day 201 of the year 1971.
   - DATE$(12001) returns the date 01–Jan–82, day one of the year 1982.
   - DATE$(10202) returns the date 21–Jul–80, day 202 of the year 1980.

Remarks

1. If *int-exp* equals zero, DATE$ returns the current date.
2. The *str-var* returned by the DATE$ function consists of nine characters and expresses the day, month, and year in the form *dd-Mmm-yy*.
3. If you specify an invalid date, such as day 385, results are unpredictable.
4. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to an integer of the default size.
5. On RSTS/E systems, the form of the DATE$ function's output can be changed to ISO format, yy.mm.dd, during the installation procedure, or to the format selected by the system manager at system start-up time.

Example

```
10  DECLARE STRING todays_date
    todays_date = DATE$(0)
    PRINT todays_date
```

The output is:

```
26-Apr-91
```

DECLARE

The DECLARE statement explicitly assigns a name and a data type to a variable, an entire array, a function, or a constant.

There are three formats:

- For variables and arrays
- For functions
- For named constants

Format

1. Variables and Arrays

DECLARE data-type decl-item [,[data-type] decl-item], . . .

2. DEF Functions

DECLARE data-type **FUNCTION** { def-name [([def-param], . . .)] }, . . .

3. Named Constants

DECLARE data-type **CONSTANT** { const-name = const-exp }, . . .

decl-item: { array-name (int-const, . . .) | unsubs-var }

def-param: [data-type]

Syntax Rules

1. *Data-type* can be any BASIC–PLUS–2 data type keyword. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.
2. The following rules apply when declaring variables:
   - *Decl-item* names an array or a variable.
   - A *decl-item* named in a DECLARE statement cannot be named in another DECLARE statement, or in a DEF, EXTERNAL, FUNCTION, SUB, COMMON, MAP, or DIM statement.
   - *Inst-const* specifies the upper bounds of the array.
   - Each *decl-item* is associated with the preceding data type. A data type is required for the first *decl-item*.

- *Decl-items* of data type STRING are dynamic strings.

3. The following rules apply when declaring DEF functions:
   - *Def-name* names the DEF function.
   - *Data-type* specifies the data type of the value the function returns.
   - Data type keywords must be separated by commas.
   - *Def-params* specify the number and, optionally, the data type of the DEF parameters. Parameters define the arguments the DEF expects to receive when invoked.
     - When you specify a data type, all following parameters are of that data type until you specify a new data type.
     - If you do not specify any data type, parameters take the current default data type and size.
     - The number of parameters equals the number of commas plus one. For example, empty parentheses specify one parameter of the default type and size, one comma inside the parentheses specifies two parameters of the default type and size; and so on. One data type inside the parentheses specifies one parameter of the specified data type; two data types separated by one comma specifies two parameters of the specified type, and so on.
4. The following rules apply when defining named constants:
   - *Const-name* is the name you assign to the constant.
   - *Data-type* specifies the data type of the constant. The value of the *const* must be numeric if the data type is numeric and string if the data type is STRING. If the data type is STRING, *const* must be a quoted string or another string constant.
   - *Const-exp* cannot be of the RFA data type.
   - String constants cannot exceed 128 characters.
   - BASIC–PLUS–2 allows *const-exp* to be an expression for STRING and INTEGER data types. Expressions are not allowed as values when you name floating-point constants.
   - Allowable operators in DECLARE CONSTANT expressions include all valid arithmetic, relational, and logical operators except exponentiation. Built-in functions cannot be used in DECLARE CONSTANT expressions. The following example uses valid expressions as values:

```
DECLARE STRING CONSTANT left_arrow = "<-----" + LF + CR
```

Remarks

1. The DECLARE statement is not executable.
2. The DECLARE statement must lexically precede any reference to the variables, functions, or constants named in it.
3. To declare a virtual or run-time array, use the DIMENSION statement.
4. When declaring variables:
   - Subsequent *decl-items* are associated with the specified data type until you specify another data type.
   - All variables named in a DECLARE statement are initialized to zero if numeric or to the null string if string.
5. When declaring DEF functions:
   - The DECLARE FUNCTION statement allows you to name a function defined in a DEF or DEF* statement, specify the data type of the value the function returns, and declare the number and data type of the DEF parameters.
   - The first specification of a data type for a *def-param* is the default for subsequent arguments until you specify another *def-param*.
   - *Data-type* keywords must be separated by commas. For example:

     ```
     10  DECLARE DOUBLE FUNCTION interest(DOUBLE,SINGLE,,)
     ```

     This example declares two parameters of the default type and size, one DOUBLE parameter, and three SINGLE parameters for the function named *interest*.
6. When declaring named constants:
   - The DECLARE CONSTANT statement allows you to name a constant value and assign a data type to that value. Note that you can specify only one data type in a DECLARE CONSTANT statement. To declare a constant of another data type, you must use a second DECLARE CONSTANT statement.
   - During program execution, you cannot change the value assigned to the constant.
   - You cannot use a *const-nam* where a variable is required.

- BASIC–PLUS–2 signals an error if the data type of *const-exp* does not match the specified data type.

Note

Data types specified in a DECLARE statement override any defaults specified in COMPILE command qualifiers or OPTION statements.

7. Although the data types LONG, WORD, and BYTE allow the values -21474813648, -32768, and -128, you cannot specify these constants explicitly. If you specify these values explicitly, BASIC–PLUS–2 signals an integer overflow error. You can create integer constants with these values only if you supply their value with an expression. For example:

```
DECLARE WORD CONSTANT X = -32767 - 1
```

Examples

1.
```
10  !DEF functions
    DECLARE INTEGER FUNCTION amount(DOUBLE,BYTE)
```

2.
```
10 !Named Constants
   DECLARE DOUBLE CONSTANT interest_rate = 15.22
```

DEF

The DEF statement lets you define a single- or multi-line function.

The first format defines a single-line function DEF; the second format defines a multiple-line function DEF.

Format

1. Single-line DEF

DEF [data-type] def-name [([data-type] unsubs-var , . . .)] = exp

2. Multiple-line DEF

DEF [data-type] def-name [([data-type unsubs-var], . . .)]
[statement]...
[statement]...

{ **END DEF** / **FNEND** } [exp]

Syntax Rules

1. *Data-type* can be any BASIC–PLUS–2 data type keyword. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.
2. The data type that precedes the *def-name* specifies the data type of the value returned by the DEF function.
3. *Def-name* is the name of the DEF function. The *def-name* can contain from 1 through 31 characters.
4. If the *def-name* also appears in a DECLARE FUNCTION statement, the following rules apply:
   - A function data type is required.
   - The first character of the *def-name* must be an alphabetic character (A through Z). The remaining characters can be any combination of letters, digits (0 through 9), dollar signs ($), underscores (_), or periods (.).
5. If the *def-name* does not appear in a DECLARE FUNCTION statement, but the DEF statement appears before the first reference to the *def-name*, the following rules apply:
   - The function data type is optional.

- The first character of the *def-name* must be an alphabetic letter (A through Z). The remaining characters can be any combination of letters, digits, dollar signs, underscores, or periods.
- If a function data type is not specified, the last character in the *def-name* must be a percent sign for an INTEGER function, or a dollar sign for a STRING function; otherwise, it will take the current default data type.

6. If the *def-name* does not appear in a DECLARE FUNCTION statement, and the DEF statement appears after the first reference to the *def-name*, the following rules apply:
   - The function data type cannot be present.
   - The first two characters of the *def-name* must be FN. The remaining characters can be any combination of letters, digits, dollar signs, or periods, with one restriction: the last character must be a percent sign for an INTEGER function, or a dollar sign for a STRING function.
   - There must be at least one character between the FN characters and the ending dollar sign or percent character. FN$ and FN% are not valid function names.
7. *Unsubs-var* specifies optional formal DEF parameters. Because the parameters are local to the DEF function, any reference to these variables outside the DEF body creates a different variable.
8. You can specify the data type of DEF parameters with a data type keyword. If you do not include a data type, the parameters are of the default type and size. Parameters that follow a data type keyword are of the specified type and size until you specify another data type.
9. You can specify up to 32 parameters in a DEF statement.
10. When you define a single-line function DEF, *exp* specifies the operations the function performs.
11. When you define a multiple-line function DEF, the following rules apply:
    - *Statements* specifies the operations the function performs.
    - The END DEF or FNEND statement is required to end a multi-line DEF.
    - BASIC–PLUS–2 does not allow you to specify any statements that indicate the beginning or end of any SUB, FUNCTION, or PROGRAM in a function definition.

- *Exp* specifies the function result. *Exp* must be compatible with the DEF data type.

Remarks

1. When BASIC–PLUS–2 encounters a DEF statement, control of the program passes to the next executable statement after the DEF.
2. The function is invoked when you use the function name in an expression.
3. You cannot specify how parameters are passed. When you invoke a function, BASIC–PLUS–2 evaluates parameters from left to right and passes parameters to the function so that they cannot be modified. Numeric parameters are passed by value and string parameters are passed by descriptor, where the descriptor points to a local copy. DEF functions can reference variables in the main program, but they cannot reference variables in other DEF or DEF* functions. A DEF function can, therefore, modify other variables in the program, but not variables within another DEF function.
4. A DEF function is local to the program, subprogram, or function that defines it.
5. You can declare a DEF by defining it, by using the DECLARE FUNCTION statement, or by implicitly declaring it with a reference to the function in an expression.
6. If your program invokes a function with a name that does not start with FN before the DEF statement defines the function, BASIC–PLUS–2 signals an error.
7. If the number of parameters, types of parameters, or type of result declared in the invocation disagree with the number or types of parameters defined in the DEF statement, BASIC–PLUS–2 signals an error.
8. DATA statements in a multi-line DEF are not local to the function; they are local to the program module containing the function definition.
9. To return a value in a multi-line DEF, make an assignment to the DEF function name from inside the DEF function, or supply a value in an EXIT DEF or END DEF statement. You can only assign a value to a DEF function name while inside the DEF function.
10. The function value is initialized to zero or the null string each time you invoke the function.
11. ON ERROR GO BACK is the default error handler in a DEF function definition.

12. If an error occurs in a DEF function that is not handled in the DEF function, control passes to the error handler of the main program. If the main program has no error handler, control passes to the default BASIC–PLUS–2 error handler.
13. ON ERROR statements within a DEF function are local to the function.
14. A GOTO, GOSUB, ON ERROR GOTO, or RESUME statement in a multi-line function definition must refer to a line number or label in the same function definition.
15. You cannot transfer control into a multi-line DEF except by invoking the function.
16. DEF functions can be recursive; however, BASIC–PLUS–2 does not detect infinitely recursive DEF functions. If your program invokes an infinitely recursive DEF function, your program will eventually terminate with a fatal error.

Examples

1.
```
10 !Single-Line DEF
   DEF DOUBLE add (DOUBLE A, B, SINGLE C, D, E) = A + B + C + D + E
   INPUT 'Enter five numbers to be added';V,W,X,Y,Z
   PRINT 'The sum is';ADD(V,W,X,Y,Z)
```

The output is:

```
Enter five numbers to be added? 1,2,3,4,5
The sum is 15
```

2.
```
10  ! PROGRAM I_want_a_raise
      OPTION SCALE = 2
      DECLARE DOUBLE CONSTANT Overtime_factor = 0.50
      DECLARE DOUBLE My_hours, My_rate, Overtime

      DEF DOUBLE Calculate_pay (DOUBLE Hours, Rate)

        Overtime = Hours - 40.0
        Overtime = 0.0 IF Overtime < 0.0
        Calculate_pay = (Hours * Rate) + (Overtime * Overtime_factor * Rate)

      END DEF

      INPUT 'Your hours this week';My_hours
      INPUT 'Your hourly rate';My_rate
      PRINT 'My pay this week is';Calculate_pay (My_hours,My_rate)

      END
```

The output is:

```
Your hours this week? 45.7
Your hourly rate? 20.35
My pay this week is 987.95
```

DEF*

The DEF* statement lets you define a single- or multi-line function.

The first format defines a single-line function; the second format defines a multiple-line function.

Note

The DEF* statement is not recommended for new program development. It is recommended that you use the DEF statement for defining single- and multi-line functions.

Format

1. Single-line DEF*

DEF* [data-type] def-name [([data-type] unsubs-var , . . .)] = exp

2. Multiple-line DEF*

DEF* [data-type] def-name [([data-type] unsubs-var , . . .)]
[statement]...
[statement]...

{ **END DEF** / **FNEND** } [exp]

Syntax Rules

1. *Data-type* can be any BASIC–PLUS–2 data type keyword. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.
2. The data type that precedes the *def-name* specifies the data type of the value returned by the DEF* function.
3. *Def-name* is the name of the DEF* function. The *def-name* can contain from 1 to 31 characters.
4. If the *def-name* also appears in a DECLARE FUNCTION statement, the following rules apply:
   - A function data type is required.

- The first character of the *def-name* must be an alphabetic character (A through Z). The remaining characters can be any combination of letters, digits (0 through 9), dollar signs ($), underscores (_), or periods (.).

5. If the *def-name* does not appear in a DECLARE FUNCTION statement, but the DEF* statement appears before the first reference to the *def-name*, the following rules apply:
   - The function data type is optional.
   - The first character of the *def-name* must be an alphabetic character (A through Z). The remaining characters can be any combination of letters, digits, dollar signs, underscores, or periods.
   - If a function data type is not specified, the last character in the *def-name* must be a percent sign for an INTEGER function, or a dollar sign for a STRING function, or the function will take the default data type and size.
6. If the *def-name* does not appear in a DECLARE FUNCTION statement, and the DEF* statement appears after the first reference to the *def-name*, the following rules apply:
   - The function data type cannot be present.
   - The first two characters of the *def-name* must be FN. The remaining characters can be any combination of letters, digits, dollar signs, or periods, with one restriction: the last character must be a percent sign for an INTEGER function, or a dollar sign for a STRING function; otherwise, the function takes the default data type and size.
   - There must be at least one character between the FN characters and the ending dollar sign or percent character. FN$ and FN% are not valid function names.
7. *Unsubs-var* specifies optional formal function parameters.
8. You can specify the data type of function parameters with a data type keyword. If you do not specify a data type, parameters are of the default type and size. Parameters that follow a data type are of the specified type and size until you specify another data type.
9. You can specify up to 32 parameters in a DEF* statement.
10. When you specify a single-line function DEF*, *Exp* specifies the operations the function performs.
11. When you specify a multiple-line function DEF*, the following rules apply:
    - *Statements* specifies the operations the function performs.

- The END DEF or FNEND statement is required to end a multi-line DEF*.
- BASIC–PLUS–2 does not allow you to specify any statements that indicate the beginning of any SUB, FUNCTION, PROGRAM, or DEF, or the end or any SUB, FUNCTION, or PROGRAM, in a function definition.
- *Exp* specifies the function result. *Exp* must be compatible with the DEF data type.

Remarks

1. When BASIC–PLUS–2 encounters a DEF* statement, control of the program passes to the next executable statement after the DEF*.
2. A function defined by the DEF* statement is invoked when you use the function name in an expression.
3. You cannot specify how parameters are passed. When you invoke a DEF* function, BASIC–PLUS–2 evaluates parameters from left to right and passes parameters to the function so that they cannot be modified. Numeric parameters are passed by value, and string parameters are passed by descriptor, where the descriptor points to a local copy. DEF* functions can reference variables in the main program, but they cannot reference variables in other DEF or DEF* functions. A DEF* function can, therefore, modify variables in the program, but not variables within another DEF* function.
4. The following differences exist between DEF* and DEF statements:
   - You can use the GOTO, ON GOTO, GOSUB, and ON GOSUB statements to a branch outside a multi-line DEF*, but they are not recommended.
   - Although other variables used within the body of a DEF* function are not local to the DEF* function, DEF* formal parameters are. If you change the value of formal parameters within a DEF* function and then transfer control out of the DEF* function without executing the END DEF or FNEND statement, variables outside the DEF* that have the same names as DEF* formal parameters are also changed.
   - A DEF* function value is not initialized when the DEF* function is invoked. Therefore, if a DEF* function is invoked, and no new function value is assigned, the DEF* function returns the value of its previous invocation.

- The error handler of the program module that contains the DEF* is the default error handler for a DEF* function. Parameters return to their original values when control passes to the error handler.

5. A DEF* is local to the program or subprogram that defines it.
6. You can declare a DEF* either by defining it, by using the DECLARE FUNCTION statement, or by implicitly declaring it with a reference to the function in an expression.
7. If the number of parameters, types of parameters, or type of result declared in the invocation disagree with the number or types of parameters defined in the DEF* statement, BASIC–PLUS–2 signals an error.
8. DEF* functions can be recursive; however, BASIC–PLUS–2 does not detect infinitely recursive DEF* functions. If your program invokes an infinitely recursive DEF* function, your program will eventually terminate with a fatal error.
9. DATA statements in a multi-line DEF* are not local to the function; they are local to the program module containing the function definition.
10. To return a value in a multi-line DEF*, make an assignment to the DEF function from inside the DEF function, or assign a value to an EXIT DEF or END DEF statement. You can assign a value to a DEF function only while inside the DEF function.
11. You can pass up to 32 parameters to a DEF* function.

Examples

1.
```
10 !Single-Line DEF*
   DEF* STRING CONCAT(STRING A,B) = A + B
   DECLARE STRING word1,word2
   INPUT "Enter two words";word1,word2
   PRINT CONCAT (word1,word2)
```

The output is:

```
Enter two words? TO
? DAY
TODAY
```

2.
```
10  !Multi-Line DEF*
    DEF* DOUBLE example(DOUBLE A, B, SINGLE C, D, E)
         EXIT DEF IF B = 0
         example = (A/B) + C - (D*E)
    END DEF
    INPUT "Enter 5 numbers";V,W,X,Y,Z
    PRINT example(V,W,X,Y,Z)
```

The output is:

```
Enter 5 numbers? 2,4,6,8,1
-1.5
```

DELETE

The DELETE statement removes a record from a relative or indexed file.

Format

DELETE #chnl-exp

Syntax Rules

Chnl-exp is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).

Remarks

1. The DELETE statement removes the current record from a file. Once the record is removed, you cannot access it.
2. The file specified by *chnl-exp* must have been opened with ACCESS MODIFY or WRITE.
3. You can delete a record only if the last I/O statement executed on the specified channel was a successful GET or FIND operation.
4. The DELETE statement leaves the current record pointer undefined and the next record pointer unchanged.
5. BASIC–PLUS–2 signals an error when the following conditions exist:
   - The I/O channel is illegal or not open.
   - No current record exists.
   - The specified record is locked.
   - Record access is illegal or illogical.
   - The operation is illegal.

Example

```
10  DECLARE STRING record_num
    .
    .
    .
    OPEN "CUS.DAT" FOR INPUT AS #1, RELATIVE FIXED       &
          ACCESS MODIFY, RECORDSIZE 40
    .
    .
    .
    INPUT "WHICH RECORD WOULD YOU LIKE TO EXAMINE";record_num
    GET #1, RECORD record_num
    DELETE #1
    .
    .
    .
```

DET

The DET function returns the value of the determinant of the last matrix inverted with the MAT INV statement.

Format

real-var = **DET**

Syntax Rules

None.

Remarks

1. When a matrix is inverted with the MAT INV statement, BASIC–PLUS–2 calculates the determinant as a by-product of the inversion process. The DET function retrieves this value.
2. If your program does not contain a MAT INV statement, the DET function returns a value of zero.
3. The value returned by the DET function is a floating-point value of the default size.

Example

```
10  MAT INPUT first_array(3,3)
    MAT PRINT first_array;
    PRINT
    MAT inv_array = INV (first_array)
    determinant = DET
    MAT PRINT inv_array;
    PRINT
    PRINT determinant
    PRINT
    MAT mult_array = first_array * inv_array
    MAT PRINT mult_array;
```

The output is:

```
? 1,0,0,0,1,0,0,0,1
 1 0 0
 0 1 0
 0 0 1

 1 0 0
 0 1 0
 0 0 1

 1

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

DIF$

The DIF$ function returns a numeric string whose value is the difference between two numeric strings.

Format

str-var = **DIF$**(str-exp1, str-exp2)

Syntax Rules

Each *str-exp* can contain up to 54 ASCII digits, an optional decimal point, and an optional leading sign.

Remarks

1. BASIC–PLUS–2 subtracts *str-exp2* from *str-exp1* and stores the result in *str-var*.
2. The difference between two integers takes the precision of the larger integer.
3. The difference between two decimal fractions takes the precision of the more precise fraction, unless trailing zeros generate that precision.
4. The difference between two floating-point numbers takes precision as follows:
   - The difference of the integer parts takes the precision of the larger part.
   - The difference of the decimal fraction part takes the precision of the more precise part.
5. BASIC–PLUS–2 truncates leading and trailing zeros.

Example

```
10  PRINT DIF$ ("689","-231")
```

The output is:

```
920
```

DIMENSION

The DIMENSION statement creates and names a static, dynamic, or virtual array. The array subscripts determine the dimensions and the size of the array. You can specify the data type of the array and associate the array with an I/O channel. There are three formats:

- For creating nonvirtual and nonexecutable arrays
- For creating executable arrays
- For creating virtual arrays

Format

1. Nonvirtual and Nonexecutable Arrays

{ DIM / DIMENSION } {[data-type] array-name (int-const , . . .) }, . . .

2. Executable Arrays

{ DIM / DIMENSION } {[data-type] array-name (int-var, . . .) }, . . .

3. Virtual Arrays

{ DIM / DIMENSION } #chnl-exp, { [data-type] array-name
(int-const, . . .) [= int-const] }, . . .

Syntax Rules

1. An array name in a DIM statement cannot also appear in a COMMON, MAP, or DECLARE statement.
2. *Data-type* can be any BASIC–PLUS–2 data type keyword. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.
3. If you specify a data type and the array name ends in a percent sign (%) or dollar sign ($) suffix character, the variable must have an INTEGER or STRING data type.

4. If you do not specify a data type, the array name determines the type of data the array holds. If the array name ends in a percent sign, the array stores integer data of the default integer size. If the array name ends in a dollar sign, the array stores string data; otherwise, the array stores data of the default type and size.
5. An array can have up to eight dimensions. Nonvirtual array sizes are limited by the virtual memory limits of your system.
6. Each instance of *int-const* or *int-vbl* within the parentheses specifies the upper bound of an array dimension. BASIC–PLUS–2 array bounds must be in the range 0 to 32767.
7. Although the compiler does not generate an error for subscript values outside of these ranges, there is a limit to the amount of storage your system can allocate. Therefore, very large arrays can cause an internal allocation error or a run-time error.
8. The following rules apply when creating nonvirtual and nonexecutable arrays:
   - When all the dimension specifications are integer constants, as in DIM A(15%,10%,20%), the DIM statement is nonexecutable and the array size is static. A static array cannot appear in another DIM statement because BASIC–PLUS–2 determines storage requirements at compilation time.
   - A nonexecutable DIM statement must lexically precede any reference to the array it dimensions. That is, you must dimension a static array before you can reference any of its elements.
9. The following rules apply when creating virtual arrays:
   - The virtual array must be dimensioned and the file must be open before you can reference the array.
   - When the data type is STRING, the *=int-const* clause specifies the length of each array element. The default string length is 16 characters. Virtual string array lengths are rounded to the next higher power of 2. Therefore, specifying an element length of 12 results in an actual length of 16. For example:

```
DIM #1, STRING vir_array(100) = 12
OPEN "STATS.BAS" FOR OUTPUT as #1, VIRTUAL
```

10. The following rules apply when creating executable arrays:

When any of the dimension specifications are integer variables as in DIM A(10%,20%,*Y*%), the DIM statement is executable and the array is dynamic. A dynamic array can be redimensioned with a DIM statement any number of times because BASIC–PLUS–2 allocates storage at run time when each DIM statement is executed.

Remarks

1. You can create an array implicitly by referencing an array element without using a DIM statement. This causes BASIC–PLUS–2 to create a static array with dimensions of (10), (10,10), (10,10,10), and so on, depending on the number of bounds specifications in the referenced array element. You cannot create virtual or executable arrays implicitly.
2. The lower bound of an array is always zero, rather than 1. Thus, A(10) allocates 11 elements, A(10,10) allocates 121 elements, and A(0,0,0) allocates 1 element.
3. BASIC–PLUS–2 allocates storage for arrays by row, from right to left.
4. The following rules apply when creating nonvirtual and nonexecutable arrays:
   - You can declare arrays with the COMMON, MAP, and DECLARE statements. Arrays so declared cannot be redimensioned with the DIM statement. Furthermore, string arrays declared with a COMMON or MAP statement are always fixed length.
   - If you reference an array element declared in an array whose subscripts are larger than the bounds specified in the DIM statement, BASIC–PLUS–2 signals the error "Subscript out of range" (ERR=55).
   - Arrays that require more than 32767 bytes of storage generate the error "Array <name> too large."
5. The following rules apply when creating virtual arrays:
   - When the rightmost subscript varies faster than the subscripts to the left, fewer disk accesses are necessary to access array elements in virtual arrays.
   - Using the same DIM statement for multiple virtual arrays allocates all arrays in a single disk file. The arrays are stored in the order in which they were declared.

- Any program or subprogram can access a virtual array by declaring it in a virtual DIMENSION statement. For example:

```
DIM #1, A(10)
DIM #1, B(10)
```

  In this example, array *B* overlays array *A*. You must specify the same channel number, data types, and limits in the same order as they occur in the DIM statement that created the virtual array.

- BASIC–PLUS–2 stores a string in a virtual array by padding it with trailing nulls to the length of the array element. It removes these nulls when it retrieves the string from the virtual array. Remember that string array element sizes are always rounded to the next power of 2.
- On RSX systems, the OPEN statement for a virtual array must include the ORGANIZATION VIRTUAL clause for the channel specified in the DIMENSION statement.
- BASIC–PLUS–2 does not initialize virtual arrays and treats them as statically allocated arrays. You cannot redimension virtual arrays.
- Refer to the *BASIC–PLUS–2 User's Guide* for more information on virtual arrays.

6. The following rules apply when creating executable arrays:
   - You create an executable, dynamic array by using integer variables for array bounds, as in DIM A(*Y%,X%*). This eliminates the need to dimension an array to its largest possible size. Array bounds in an executable DIM statement can be constants or variables, but not expressions. At least one bound must be a variable.
   - You cannot reference an array named in an executable DIM statement until after the DIM statement executes.
   - You can redimension a dynamic array to make the bounds of each dimension larger or smaller, but you cannot change the number of dimensions. For example, you cannot redimension a four-dimensional array to be a five-dimensional array.
   - The executable DIM statement cannot be used to dimension virtual arrays, arrays received as formal parameters, or arrays declared in COMMON, MAP, or nonexecutable DIM statements.
   - An executable DIM statement always reinitializes all elements in the array to zero (for numeric arrays) or to the null string if string.

- If you reference an array element declared in an executable DIM statement whose subscripts are larger than the bounds specified in the last execution of the DIM, BASIC–PLUS–2 signals the error "Subscript out of range" (ERR=55).

Examples

1.
```
10  !Nonvirtual, Nonexecutable
    DIM STRING name_list(100), BYTE age(100)
```

2.
```
10  !Virtual
    DIM #1%, STRING name_list(500), REAL amount(10,10)
```

3.
```
10  !Executable
    DIM DOUBLE inventory(base,markup)
    .
    .
    .
    DIM DOUBLE inventory (new_base,new_markup)
```

ECHO

The ECHO function causes characters to be echoed at a terminal that is opened on a specified channel.

Format

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

Syntax Rules

Chnl-exp is a numeric expression that specifies a channel number associated with an open terminal. It cannot be preceded by a number sign (#).

Remarks

1. The ECHO function is the complement of the NOECHO function; each function disables the effect of the other.
2. The ECHO function has no effect on an unopened channel.
3. The ECHO function always returns a value of zero.

Example

```
10  DECLARE INTEGER Y,                    &
          STRING pass_word
    Y = NOECHO(0%)
    SET NO PROMPT
    INPUT "Enter your password: ";pass_word
    Y = ECHO(0%)
    IF pass_word = "Darlene"
    THEN
        PRINT CR+LF+"YOU ARE CORRECT !"
    END IF
```

The output is:

```
Enter your password?
YOU ARE CORRECT !
```

EDIT$

The EDIT$ function performs one or more string editing functions, depending on the value of its integer argument.

Format

str-var = **EDIT$**(str-exp, int-exp)

Syntax Rules

None.

Remarks

1. BASIC–PLUS–2 edits *str-exp* to produce *str-var*.
2. The editing that BASIC–PLUS–2 performs depends on the value of *int-exp*. Table 4–3 describes EDIT$ values and functions.
3. All values are additive. For example, you can perform the editing functions of values 8, 16, and 32 by specifying a value of 56. You can also specify the values in an expression, for example, 8% + 16% + 32%. Specifying editing values as an expression eliminates human error.
4. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to an integer of the default size.

Table 4–3 EDIT$ Values

Value	Function
1	Discards each character's parity bit (bit 7)
2	Discards all spaces and tabs
4	Discards all carriage returns <CR>, line feeds LF, form feeds FF, deletes <DEL>, escapes <ESC>, and nulls <NUL>
8	Discards leading spaces and tabs
16	Converts multiple spaces and tabs to a single space
32	Converts lowercase letters to uppercase letters

(continued on next page)

Table 4–3 (Cont.) EDIT$ Values

Value	Function
64	Converts left bracket to left parenthesis ([to ( ) and right bracket to right parenthesis ( ] to))
128	Discards trailing spaces and tabs (same as TRM$ function)
256	Suppresses all editing for characters within quotation marks; if the string has only one quotation mark, BASIC–PLUS–2 suppresses all editing for the characters following the quotation mark

Example

```
10  DECLARE STRING old_string, new_string
    old_string = "a value of 32 converts lowercase letters to uppercase"
    new_string = EDIT$(old_string,32)
    PRINT new_string
```

The output is:

```
A VALUE OF 32 CONVERTS LOWERCASE LETTERS TO UPPERCASE
```

END

The END statement marks the physical and logical end of a main program, a program module, or a block of statements.

Format

END [block]

block: { **DEF**[*exp*] | **FUNCTION**[*exp*] | **IF** | **PROGRAM** | **SELECT** | **SUB** }

Syntax Rules

1. The END statement with no *block* keyword marks the end of a main program. The END or END PROGRAM statement must be the last statement on the last lexical line of the main program.
2. The END statement followed by a *block* keyword marks the end of a program, a BASIC–PLUS–2 SUB, or FUNCTION subprogram, a DEF, an IF, a PROGRAM, or a SELECT statement block.
3. The END *block* statement must be the lexically last statement in a subprogram or statement block and must match the statement that established the subprogram or statement block.

Remarks

1. END DEF and END FUNCTION
   - When BASIC–PLUS–2 executes an END DEF or an END FUNCTION statement, it returns the function value to the statement that invoked the function and releases all storage associated with the DEF or FUNCTION.
   - If you specify an optional expression with the END DEF or END FUNCTION statement, the expression must be compatible with the DEF or FUNCTION data type. The expression is the function result unless an EXIT DEF or EXIT FUNCTION statement is executed. This expression supersedes all function assignments.

- The END DEF statement restores the error handler in effect when the DEF was invoked. This is not true of the DEF* statement because error handling in a DEF* statement is global.
- The END FUNCTION statement does not affect I/O operations or files.

2. END PROGRAM
   - The END PROGRAM statement allows you to end a program module.
   - You can specify an END PROGRAM statement without a matching PROGRAM statement.
3. END SUB
   - The END SUB statement does not affect I/O operations or files.
   - The END SUB statement releases the storage allocated to local variables and returns control to the calling program.
   - The END SUB statement cannot be executed in an error handler unless the END SUB is in a subprogram called by the error handler of another routine.
   - The END SUB statement restores the error handler in effect when the SUB was invoked.
4. When an END or END PROGRAM statement marking the end of a main program executes, BASIC–PLUS–2 closes all files and releases all program storage.
5. If you use ON ERROR error handling, you must clear any errors with the RESUME statement before executing an END PROGRAM, END SUB, or END FUNCTION statement.
6. Except for the END PROGRAM statement, BASIC–PLUS–2 signals an error when a program contains an END *block* statement with no corresponding and preceding *block* keyword.

Example

```
10   INPUT "Guess a number";A%
     IF A% = 24
     THEN
          PRINT, "YOU GUESSED IT!"
     END IF

     IF A% < 24
     THEN
          PRINT, "BIGGER IS BETTER!"
     GOTO 10
     END IF

     IF A% > 24
     THEN
          PRINT, "SMALLER IS BETTER!"
          GOTO 10
     END IF

     END PROGRAM
```

ERL

The ERL function returns the number of the BASIC line where the last error occurred.

Format

int-var = **ERL**

Syntax Rules

None.

Remarks

1. The value of *int-var* returned by the ERL function is a WORD integer.
2. If the ERL function is used before an error occurs or after an error is handled with a RESUME statement, the results are undefined.
3. The ERL function overrides the /NOLINE qualifier. If a program compiled with the /NOLINE qualifier in effect contains an ERL function, BASIC–PLUS–2 signals the warning message "ERL overrides NOLINE."

Example

```
100     IF ERL = 20
         THEN
             PRINT "Invalid input...try again"
             RETRY
         ELSE
             PRINT "UNEXPECTED ERROR"
```

ERN$

The ERN$ function returns the name of the main program or subprogram that was executing when the last error occurred.

Format

str-var = **ERN$**

Syntax Rules

None.

Remarks

If the ERN$ function executes before an error occurs, ERN$ is undefined. When an error occurs, ERN$ is set to the name of the module that caused the error.

Example

```
10 ON ERROR GOTO 20
   DECLARE LONG int_exp
15 INPUT "Enter an number";int_exp
   PRINT Date$(int_exp)
   GOTO 30
20 PRINT 'Error in module';ERN$
   RESUME 15
30 END
```

The output is:

```
Enter a number? ABCD
Error in module DATE
Enter a number? 0
11-May-91
```

ERR

The ERR function returns the error number of the current run-time error.

Format

int-var = **ERR**

Syntax Rules

None.

Remarks

1. The value of *int-var* returned by the ERR function is always a WORD integer.
2. If the ERR function is used before an error occurs or after an error is handled with a RESUME statement, the results are undefined.
3. See the *BASIC–PLUS–2 User's Guide* for a list of BASIC–PLUS–2 run-time errors and their numbers.

Example

```
10  PRINT "Error number";ERR
    IF ERR = 50  THEN PRINT "DATA FORMAT ERROR"
    ELSE PRINT "UNEXPECTED ERROR"
    END IF
    RESUME
    END
```

ERT$

The ERT$ function returns explanatory text associated with an error number.

Format

str-var = **ERT$**(int-exp)

Syntax Rules

Int-exp is a BASIC–PLUS–2 error number. The error number must be in the range 0 through 255.

Remarks

1. The ERT$ function can be used at any time to return the text associated with a specified error number.
2. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to an integer of the default size.

Example

```
20      PRINT "Error number";ERR
        PRINT ERT$(ERR)
        RETRY
```

EXIT

The EXIT statement lets you exit from a main program, a SUB, or FUNCTION subprogram, a multi-line DEF, or a statement block.

Format

EXIT block

block: { **DEF**[*exp*] | **FUNCTION**[*exp*] | **SUB** | **PROGRAM** | label }

Syntax Rules

1. The DEF, FUNCTION, SUB, and PROGRAM keywords specify the type of subprogram, or multi-line DEF from which BASIC–PLUS–2 is to exit.
2. If you specify an optional expression with the EXIT DEF statement or with the EXIT FUNCTION statement, the expression becomes the function result and supersedes any function assignment. It also overrides any expression specified on the END DEF or END FUNCTION statement. Note that the expression must be compatible with the FUNCTION or DEF data type.
3. *Label* specifies a statement label for an IF, SELECT, FOR, WHILE, or UNTIL statement block.

Remarks

1. An EXIT SUB, EXIT FUNCTION, EXIT PROGRAM, or EXIT DEF statement is equivalent to an unconditional branch to an equivalent END statement. Control then passes to the statement that invoked the DEF or to the statement following the statement that called the subprogram.
2. The EXIT PROGRAM statement causes BASIC–PLUS–2 to exit from a main program module.
3. BASIC–PLUS–2 allows you to specify an EXIT PROGRAM statement without a matching PROGRAM statement.

4. The EXIT *label* statement is equivalent to an unconditional branch to the first statement following the end of the IF, SELECT, FOR, WHILE, or UNTIL statement labeled by the specified label.
5. An EXIT FUNCTION, EXIT SUB or EXIT PROGRAM statement cannot be used within a multi-line DEF function.
6. When the EXIT FUNCTION, EXIT SUB or EXIT PROGRAM statement executes, BASIC–PLUS–2 releases all storage allocated to local variables and returns control to the calling program.

Example

```
10  DEF factorial(X)
    EXIT DEF 1 if X = 0
    END DEF (X * factorial(X -1))
20  PRINT "5! = ";factorial(5)
```

The output is:

```
5! = 120
```

EXP

The EXP function returns the value of the mathematical constant *e* raised to a specified power.

Format

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

Syntax Rules

None.

Remarks

1. The EXP function returns the value of *e* raised to the power of *real-exp*.
2. BASIC–PLUS–2 expects the argument of the EXP function to be a real expression. When the argument is a real expression, BASIC–PLUS–2 returns a value of the same floating-point size. When the argument is not a real expression, BASIC–PLUS–2 converts the argument to the default floating-point size and returns a value of the default floating-point size.
3. EXP allows arguments between –88 and 88. When the argument exceeds the upper limit of this range, BASIC–PLUS–2 signals an error. When the argument exceeds the lower limit of this range, the EXP function returns a zero and BASIC–PLUS–2 does not signal an error.

Example

```
10  DECLARE SINGLE num_val
    num_val = EXP(4.6)
    PRINT num_val
```

The output is:

```
99.4843
```

EXTERNAL

The EXTERNAL statement declares constants, variables, functions, and subroutines external to your program. You can describe parameters for external functions and subroutines.

There are four formats:

- For external constants
- For external variables
- For external functions
- For external subroutines

Format

1. External Constants

EXTERNAL data-type **CONSTANT** const-name, . . .

2. External Variables

EXTERNAL data-type unsubs-var, . . .

3. External Functions

EXTERNAL data-type **FUNCTION** { func-name [pass-mech]
[(external-param , . . .)] }, . . .

4. External Subroutines

EXTERNAL SUB { sub-name [pass-mech]
[(external-param , . . .)] }, . . .

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

external-param: [param-data-type] [**DIM** ([,] . . .)] [= int-const] [pass-mech]

Syntax Rules

1. The name of an external constant, variable, function, or subroutine can be from one through six characters.
2. For all external routine declarations, the name must be a valid BASIC–PLUS–2 identifier and must not be the same as any other SUB, FUNCTION, or PROGRAM name.
3. The first character of an unquoted name must be an alphabetic character. The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), and periods (.).
4. External Constants

   For external constants, *data-type* must be WORD or INTEGER (if the default size is WORD).
5. External Variables

   For external variables, *data type* can be any valid numeric data type.
6. External Functions and Subroutines

   - For external functions and subroutines, the data type can be any valid BASIC–PLUS–2 data type. See Table 1-2 in this manual for more information on data type size, range and precision.
   - For external functions, the data type that precedes the keyword FUNCTION defines the data type of the function result.
   - *External-param* defines the form of the arguments passed to the external function or subprogram. Empty parentheses indicate that the subprogram expects zero parameters. Missing parentheses indicate that the EXTERNAL statement does not define parameters.
   - You can specify up to 32 formal parameters for BASIC–PLUS–2 subprograms and 255 formal parameters for other programs.
   - *Param-data-type* specifies the data type of a parameter. If you do not specify a data type, parameters are of the default data type and size.
   - Quoted names are allowed for external subroutines only. Quoted names can be any combination of letters, digits, dollar signs, periods, and spaces.
   - *Pass-mech* specifies how parameters are to be passed to the function or subroutine.
     - A *pass-mech* clause outside the parentheses applies to all parameters.

- A *pass-mech* clause inside the parentheses overrides the previous *pass-mech* and applies only to the specific parameter.

7. Declaring Array Dimensions

   The DIM keyword indicates that the parameter is an array. Commas specify array dimensions. The number of dimensions is equal to the number of commas plus 1. For example:

```
10 EXTERNAL STRING FUNCTION new (DOUBLE, STRING DIM(,), DIM())
```

   This statement declares a function named *new* that has three parameters. The first is a double-precision floating-point value, the second is a two-dimensional string array, and the third is a one-dimensional string array. The function returns a string result.

Remarks

1. The EXTERNAL statement must precede any program reference to the constant, variable, function, or subroutine declared in the statement.
2. The EXTERNAL statement is not executable.
3. A name declared in an EXTERNAL CONSTANT statement can be used in any nondeclarative statement as if it were a constant.
4. A name declared in an EXTERNAL FUNCTION statement can be used as a function invocation in an expression. In addition, you can invoke a function with the CALL statement unless the function data type is STRING or RFA.
5. A name declared in an EXTERNAL SUB statement can be used in a CALL statement.
6. The optional *pass-mech* clauses in the EXTERNAL FUNCTION and EXTERNAL SUB statements tell BASIC–PLUS–2 how to pass arguments to a non-BASIC function or subprogram. Table 4–1 describes BASIC–PLUS–2 parameter-passing mechanisms. Note that you cannot specify a *pass-mech* when invoking a BASIC–PLUS–2 subprogram.

   - BY VALUE specifies that BASIC–PLUS–2 passes the argument's 16-bit value.
   - BY REF specifies that BASIC–PLUS–2 passes the argument's address. This is the default for all arguments except strings and entire arrays. If you know the size of string parameters and the dimensions of array parameters, you can improve run-time performance by passing strings and arrays by reference.

- BY DESC specifies that BASIC–PLUS–2 passes the address of a BASIC–PLUS–2 descriptor. For information about the format of a BASIC–PLUS–2 descriptor for strings and arrays, see the *BASIC–PLUS–2 User's Guide*.

7. When invoked, the arguments passed to external functions and subroutines should match the external parameters declared in the EXTERNAL FUNCTION or EXTERNAL SUB statement in number, type, and passing mechanism. BASIC–PLUS–2 forces arguments to be compatible with declared parameters. If they are not compatible, BASIC–PLUS–2 signals an error.

Examples

1.
```
10  !External Constant
    EXTERNAL WORD CONSTANT IE.SUC
```

2.
```
10  !External Variable
    EXTERNAL WORD SYSNUM
```

3.
```
10  !External Function
    EXTERNAL DOUBLE FUNCTION USR$2(WORD,LONG)
```

4.
```
10  !External Subroutine
    EXTERNAL SUB calc BY DESC (STRING DIM(,),  BYTE BY REF)
```

FIELD

The FIELD statement dynamically associates string variables with all or parts of a record buffer. FIELD statements do not move data. Instead, they permit direct access through string variables to sections of a specified record buffer.

Note

The FIELD statement is supported only for compatibility with BASIC–PLUS. Because data defined in the FIELD statement can be accessed only as string data, you must use the CVTxx functions to process numeric data; therefore, you must convert string data to numeric after you move it from the record buffer. Then, after processing, you must convert numeric data back to string data before transferring it to the record buffer. It is recommended that you use the BASIC–PLUS–2 dynamic mapping feature or multiple maps instead of the FIELD statement and CVTxx functions.

Format

FIELD #chnl-exp, int-exp **AS** str-var[, int-exp **AS** str-var] . . .

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#). A file must be open on the specified channel or BASIC–PLUS–2 signals an error.
2. *Int-exp* specifies the number of characters in *str-var*.

Remarks

1. A FIELD statement is executable. You can change a buffer description at any time by executing another FIELD statement. For example:

```
FIELD #1%, 40% AS whole_field$
FIELD #1%, 10% AS A$, 10% AS B$, 10% AS C$, 10% AS D$
```

The first FIELD statement associates the first 40 characters of a buffer with the variable *whole_field$*. The second FIELD statement associates the first 10 characters of the same buffer with *A$*, the second 10 characters with *B$*, and so on. Later program statements can refer to any of the variables named in the FIELD statements to access specific portions of the buffer.

2. You cannot define virtual array strings as string variables in a FIELD statement.
3. The FIELD statement is also described in the *BASIC–PLUS Language Manual*.

Example

```
10  FIELD #8%, 2% AS U$, 2% AS CL$, 4% AS X$, 4% AS Y$
    LSET U$ = CVT%$(U%)
    LSET CL$ = CVT%$(CL%)
    LSET X$ = CVTF$(X)
    LSET Y$ = CVTF$(Y)
    U% = CVT$%(U$)
    CL% = CVT$%(CL$)
    X = CVT$F(X$)
    Y = CVT$F(Y$)
```

FIND

The FIND statement locates a specified record in a disk file and makes it the current record for a GET, UPDATE, or DELETE operation. FIND statements are valid on RMS sequential, relative, and indexed files.

Format

FIND #chnl-exp [, position-clause]

position-clause: { **RFA** *rfa-exp* | **RECORD** *num-exp* | **KEY#** *key-clause* }

key-clause: *int-exp1 rel-op key-exp*

rel-op: { **EQ** | **GE** | **GT** }

key-exp: { *int-exp2* | *str-exp* }

Syntax Rules

Chnl-exp is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).

Remarks

1. *Position-clause*

   - *Position-clause* specifies the position of a record in a file. BASIC–PLUS–2 signals an error if you specify a *position-clause* and the channel is not associated with a disk file. The RECORD *position-clause* is invalid for a SEQUENTIAL FIXED file. If you do not specify a *position-clause*, FIND locates records sequentially. Sequential record access is valid on all RMS files.

- The RFA *position-clause* allows you to randomly locate records by specifying the record file address (RFA) of a record. You specify the disk address of a record, and RMS locates the record at that address. All file organizations can be accessed by RFA.

  Rfa-exp in the RFA *position-clause* is a variable of the RFA data type that specifies the record's file address. Note that an RFA expression can only be a variable of the RFA data type or the GETRFA function. Use the GETRFA function to find the RFA of a record.

- The RECORD *position-clause* allows you to randomly locate records in relative files by specifying the record number.
  - *Num-exp* in the RECORD *position-clause* specifies the number of the record you want to locate. It must be between 1 and the number of the record with the highest number in the file.
  - When you specify a RECORD clause, *chnl-exp* must be a channel associated with an open relative file.
- The KEY *position-clause* allows you to randomly locate records in indexed files by specifying a key of reference, a relational test, and a key value.

2. *Key-clause*
   - In a *key-clause*, *int-exp1* is the target key of reference. It must be an integer in the range of zero through the highest-numbered key for the file. The primary key is #0, the first alternate key is #1, the second alternate key is #2, and so on. *Int-exp1* must be preceded by a number sign (#) or BASIC–PLUS–2 signals an error.
   - When you specify a *key-clause*, the specified channel must be a channel associated with an open indexed file.
3. *Rel-op*
   - *Rel-op* is a relational operator that specifies how *key-exp* is to be compared with *int-exp1* in the *key-clause*.
     - EQ means *equal to*
     - GE means *greater than or equal to*
     - GT means *greater than*
   - A successful random FIND operation by key locates the first record whose key satisfies the *key-clause* comparison:
     - With an *equal to* (EQ) key match, a successful FIND locates the first record in the file that equals the key value specified in *key-exp*.

If the characters specified by a *str-exp* key expression are less than the key length, characters specified by *str-exp* are matched approximately rather than exactly. For example, if you specify "ABC" and the key length is six characters, BASIC–PLUS–2 locates the first record that begins with ABC. If you specify "ABCABC", BASIC–PLUS–2 locates only a record with the key "ABCABC". If no match is possible, BASIC–PLUS–2 signals the error "Record not found" (ERR=155).

- If you specify a *greater than or equal to* (GE) key match, a successful FIND locates the first record that equals the key value specified in *key-exp*. If no exact match is possible, BASIC–PLUS–2 selects the next record in the key sort order with a higher key value. If no such record exists, BASIC–PLUS–2 signals the error "Record not found" (ERR=155).
- With a *greater than* (GT) key match, a successful FIND locates the first record with a value greater than the key value specified in *key-exp*. If no such record exists, BASIC–PLUS–2 signals the error "Record not found" (ERR=155).

4. *Key-exp*
   - *Int-exp2* specifies an integer value to be compared with the key value of a record.
   - *Str-exp* specifies a string value to be compared with the key value of a record. *Str-exp* can contain fewer characters than the key of the record you want to locate, but cannot be a null string.
5. FIND does not transfer any data to the record buffer. To access the contents of a record, use the GET statement.
6. The file specified by *chnl-exp* must be open with ACCESS READ or MODIFY before your program can execute a FIND statement.
7. A successful sequential FIND operation updates both the current record pointers and next record pointers.
   - For sequential files, a successful FIND operation locates the next sequential record (the record pointed to by the next record pointer) in the file, changes the current record pointer to the record just found, and sets the next record pointer to the next sequential record. If the current record pointer points to the last record in a file, a sequential FIND operation causes BASIC–PLUS–2 to signal "End of file on device" (ERR=11).

- For relative files, a successful FIND operation locates the record that exists with the next higher record number (or cell number), makes it the current record, and sets the next record pointer to the record with the next higher record number.
- For indexed files, a successful FIND operation locates the next logical record in the current key of reference, makes this the current record, and changes the next record pointer to the next logical record in the current key of reference.

8. A successful random access FIND operation by RFA or by record changes the current record pointer to the record specified by *rfa-exp* or *int-exp*, but leaves the next record pointer unchanged.
9. A successful random access FIND operation by key changes the current record pointer to the first record whose key satisfies the *key-clause* comparison, and the next record pointer to the record with the next value in the key sort order.
10. When a random access FIND operation by RFA, record, or key is not successful, BASIC–PLUS–2 signals the error "Record not found" (ERR=155). The values of the current record pointer and next record pointer are undefined.
11. You should not use a FIND statement on a terminal-format or virtual array file.
12. When you access a shared file, a successful FIND locks the record or bucket and unlocks the previously locked record or bucket.

Example

```
10  DECLARE LONG rec-num
    MAP (cusrec) WORD cus_num                                        &
        STRING cus_nam=20, cus_add=20, cus_city=10,  cus_zip=9
    OPEN "ACCT.DAT" FOR INPUT AS #1,                              &
          RELATIVE FIXED,                                            &
          ACCESS MODIFY                                              &
          MAP cusrec
    INPUT "Which record number would you like to delete";rec_num
    FIND #1, RECORD rec_num
    DELETE #1
    CLOSE #1
    END
```

FIX

The FIX function truncates a floating-point value at the decimal point and returns the integer portion represented as a floating-point value.

Format

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

Syntax Rules

None.

Remarks

1. The FIX function returns the integer portion of a floating-point value, not an integer value.
2. BASIC–PLUS–2 expects the argument of the FIX function to be a real expression. When the argument is a real expression, BASIC–PLUS–2 returns a value of the same floating-point size. When the argument is not a real expression, BASIC–PLUS–2 converts the argument to the default floating-point size and returns a value of the default floating-point size.
3. If *real-exp* is negative, FIX returns the negative integer portion. For example, FIX(–5.2) returns –5.

Example

```
10  DECLARE SINGLE result
    result = FIX(-3.333)
    PRINT FIX(24.566), result
```

The output is:

```
24          -3
```

FNEND

The FNEND statement is a synonym for END DEF. See the END statement for more information.

Format

FNEND [exp]

FNEXIT

The FNEXIT statement is a synonym for the EXIT DEF statement. See the EXIT statement for more information.

Format

FNEXIT [exp]

FOR

The FOR statement repeatedly executes a block of statements, while incrementing a specified control variable for each execution of the statement block. FOR loops can be conditional or unconditional, and can modify other statements.

There are four formats:

- Unconditional
- Conditional
- Unconditional statement modifier
- Conditional statement modifier

Format

1. Unconditional

FOR num-unsubs-var = num-exp1 **TO** num-exp2 [**STEP** num-exp3]
[statement]...

NEXT num-unsubs-var

2. Conditional **FOR**)

num-unsubs-var = num-exp1 [**STEP** num-exp3]

{ **UNTIL** / **WHILE** } cond-exp

[statement]...

NEXT num-unsubs-var

3. Unconditional Statement Modifier

statement **FOR** num-unsubs-var = num-exp1 **TO** num-exp2 [**STEP** num-exp3]

4. Conditional Statement Modifier

statement **FOR** num-unsubs-var = num-exp1 [**STEP** num-exp3]

{ **UNTIL** / **WHILE** } cond-exp

Syntax Rules

1. *Num-unsubs-var* must be a numeric, unsubscripted variable.
2. *Num-unsubs-var* is the loop variable. It is incremented each time the loop executes.
3. In unconditional FOR loops, *num-exp1* is the initial value of the loop variable; *num-exp2* is the maximum value.
4. In conditional FOR loops, *num-exp1* is the initial value of the loop variable, while the *cond-exp* in the WHILE or UNTIL clause is the condition that controls loop iteration.
5. *Num-exp3* in the STEP clause is the value by which the loop variable is incremented after each execution of the loop.

Remarks

1. The default for *num-exp3* is 1 if there is no STEP clause.
2. You can transfer control into a FOR loop only by returning from a function invocation, a subprogram call, a subroutine call, or an error handler that was invoked in the loop.
3. The loop variable can be modified inside the FOR loop.
4. The starting, incrementing, and ending values of a loop cannot be changed during loop execution.
5. When an unconditional FOR loop ends, the loop variable contains the value last used in the loop, not the value that caused loop termination.
6. BASIC–PLUS–2 converts *num-exp1*, *num-exp2*, and *num-exp3* to the data type of the loop variable before storing them.
7. If the loop variable exceeds the allowable range for its data type, BASIC–PLUS–2 signals the error "Integer overflow, FOR loop" (ERR=60).
8. An inner loop must be entirely within an outer loop; the loops cannot overlap.
   - There is a limit to the number of inner loops you can contain within a single outer loop. This number varies according to the complexity of the loops. If you exceed the limit, BASIC–PLUS–2 signals the error message "Program structures nested too deeply."
   - You cannot use the same loop variable in nested FOR loops. For example, if the outer loop uses FOR *I* = 1 TO 10, you cannot use the variable *I* as a loop variable in an inner loop.

9. During each iteration of a conditional loop, BASIC–PLUS–2 tests the value of *cond-exp* before it executes the loop.
   - If you specify a WHILE clause and *cond-exp* is false (value zero), BASIC–PLUS–2 exits from the loop. If the *cond-exp* is true (value nonzero), the loop executes again.
   - If you specify an UNTIL clause and *cond-exp* is true (value nonzero), BASIC–PLUS–2 exits from the loop. If the *cond-exp* is false (value zero), the loop executes again.
10. When FOR is used as a statement modifier, BASIC–PLUS–2 executes the statement until the loop variable equals or exceeds *num-exp2* or until the WHILE or UNLESS condition is satisfied.
11. Each FOR statement must have a corresponding NEXT statement or BASIC–PLUS–2 signals an error. (This is not the case if the FOR statement is used as a statement modifier.)

Examples

1.
```
10 !Unconditional
    DECLARE LONG course_num, STRING course_nam
    FOR I = 3 TO 12 STEP 3
    INPUT "Course number";course_num
    INPUT "Course name";course_nam
    NEXT I

Course number? 221
Course name? Botany
Course number? 231
Course name? Organic Chemistry
Course number? 237
Course name? Life Science II
Course number? 244
Course name? Programming in BASIC-PLUS-2
```

2.
```
10 !Unconditional Statement Modifier
    DECLARE INTEGER counter
    PRINT "This is an unconditional statement modifier"   &
          FOR counter = 1 TO 3
    END
```

The output is:

```
This is an unconditional statement modifier
This is an unconditional statement modifier
This is an unconditional statement modifier
```

3.
```
10 !Conditional Statement Modifier
   DECLARE INTEGER counter, &
           STRING my_name
   INPUT "Try and guess my name";my_name FOR counter = 1    &
         UNTIL my_name = "BASIC"
   PRINT "You guessed it!"
```

```
Try and guess my name? PASCAL
Try and guess my name? FORTRAN
Try and guess my name? BASIC
You guessed it!
```

FORMAT$

The FORMAT$ function converts an expression to a formatted string.

Format

str-var = **FORMAT$**(exp, str-exp)

Syntax Rules

1. The rules for building a format string are the same as those for printing numbers with the PRINT USING statement. See the description of the PRINT USING statement for more information.
2. You cannot specify the FORMAT$ function in a PRINT USING statement.

Remarks

None.

Example

```
10  DECLARE STRING result,          &
          INTEGER num_exp
    num_exp = 12345
    result = FORMAT$(num_exp,"##,###")
    PRINT result
```

The output is:

```
12,345
```

FSP$

The FSP$ function returns a string describing an open file on a specified channel.

Format

str-var = **FSP$**(chnl-exp)

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number. You cannot precede the *chnl-exp* with a number sign (#).
2. A file must be open on *chnl-exp*.
3. The FSP$ function must come immediately after the OPEN statement for the file.
4. The returned string is 28 bytes long.

Remarks

1. Table 4–4 describes the RMS fields and their corresponding BYTE values.
2. Use the FSP$ function with files opened as ORGANIZATION UNDEFINED. Then use multiple MAP statements to interpret the returned data.
3. See the *BASIC–PLUS–2 User's Guide* and the RMS–11 documentation for more information on FSP$ values.

Table 4–4 FSP$ Return Values and Corresponding RMS Fields

Return Value	RMS Field	Description
0	RFM + ORG	File Organization code plus record format code
1	RAT	Record handling mask
2,3	MRS	Maximum record size in bytes
4,5,6,7	ALQ	File allocation size

(continued on next page)

Table 4–4 (Cont.) FSP$ Return Values and Corresponding RMS Fields

Return Value	RMS Field	Description
8,9	BLS or BKS	Magtape block size or file bucket size
10,11	NA	Number of indexed keys
12,13,14,15	MRN	Maximum record number
16,17,18,19	BKT	Virtual block number or relative record number

Example

```
10 MAP (A) STRING A = 28
   MAP (A) BYTE org_rfm, rat, WORD mrs, LONG alq,  &
           WORD bks_bls, num_keys,LONG bkt mrn,
   OPEN "STUDENT.DAT" FOR INPUT AS #1%,        &
         ORGANIZATION UNDEFINED,               &
        RECORDTYPE ANY, ACCESS READ
   A = FSP$(1%)
   PRINT "RMS organization and record format = ";org_rfm
   PRINT "RMS record attributes = ";rat
   PRINT "RMS maximum record size = ";mrs
   PRINT "RMS allocation quantity = ";alq
   PRINT "RMS bucket size = ";bks_bls
   PRINT "Number of keys = ";num_keys
   PRINT "RMS maximum record number = ";mrn
   PRINT "RMS virtual block number = ";bkt
```

The output is:

```
RMS organization = 2
RMS record attributes = 2
RMS maximum record size = 5
RMS allocation quantity = 1
RMS bucket size = 0
Number of keys = 0
RMS maximum record number = 0
RMS virtual block number = 1
```

FSS$

The FSS$ function scans a file name string beginning at a specified position and returns a 30-character string describing the file name and status. Because file specifications differ from system to system, the returned string contains system-specific information.

Format

str-vbl = **FSS$**(str-vbl,int-vbl)

Syntax Rules

1. *Str-vbl* names the file name string to be scanned.
2. *Int-vbl* specifies the character position at which scanning starts.

Remarks

1. If you specify a floating-point variable for *int-vbl*, BASIC–PLUS–2 truncates it to an integer of the default size.
2. *Str-vbl* is a 30-character string. See the *BASIC–PLUS–2 User's Guide* for information on the encoding of *str-vbl*.
3. See the *BASIC–PLUS–2 User's Guide* for more information on the values returned by the FSS$ function.

Example

```
100     Y$ = FSS$(A$,B%)
```

FUNCTION

The FUNCTION statement marks the beginning of a FUNCTION subprogram and defines the subprogram's parameters.

Format

FUNCTION data-type func-name [([formal-param], . . .)]
[statement]...

{ **END FUNCTION** *[exp]* }
{ **FUNCTIONEND** *[exp]* }

formal param: [data-type] { unsubs-var | array-name ([int-const] [,] , . . .) }

Syntax Rules

1. *Func-name* names the FUNCTION subprogram.
2. *Func-name* can have from one through six characters. The first character must be an alphabetic character. The remaining characters can be any combination of letters, digits (0 through 9), dollar signs ($) and periods (.), with the exception that the last character cannot be a dollar sign ($).
3. *Data-type* can be any BASIC–PLUS–2 data type keyword. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.
4. The data type that precedes the *func-name* specifies the data type of the value returned by the function.
5. *Formal-param* specifies the number and type of parameters for the arguments the function expects to receive when invoked.
   - Empty parentheses indicate that the function has no parameters.
   - *Data-type* specifies the data type of a parameter. If you do not specify a data type, parameters are of the default data type and size. When you do specify a data type, all following parameters are of that data type until you specify a new data type.
   - Parameters defined in *formal-param* must agree in number and type with the arguments specified in the function invocation.

- You can specify up to 32 formal parameters.

6. *Exp* specifies the function result which supersedes any function assignment. *Exp* must be compatible with the function's data type.

Remarks

1. The FUNCTION statement must be the first statement in the FUNCTION subprogram.
2. Every FUNCTION statement must have a corresponding END FUNCTION or FUNCTIONEND statement.
3. Any BASIC–PLUS–2 statement except END, PROGRAM, END PROGRAM, SUB, SUBEND, END SUB, or SUBEXIT can appear in a FUNCTION subprogram.
4. FUNCTION subprograms must be declared with the EXTERNAL statement before your BASIC–PLUS–2 program can invoke them.
5. All variables and data, except virtual arrays, COMMON areas, MAP areas, and EXTERNAL variables, in a FUNCTION subprogram are local to the subprogram.
6. BASIC–PLUS–2 initializes local numeric variables to zero and local string variables to the null string each time the FUNCTION subprogram is invoked.
7. ON ERROR GO BACK is the default error handler for FUNCTION subprograms.
8. To return a function value, either assign a value to the function name or specify a value with the END FUNCTION or EXIT FUNCTION statement. Note that you can only assign a value to a function name while inside that function subprogram.

Example

```
10  FUNCTION REAL sphere_volume (REAL R)
    IF R < 0 THEN EXIT FUNCTION
    sphere_volume = 4/3 * PI *R **3
    END FUNCTION
```

FUNCTIONEND

The FUNCTIONEND statement is a synonym for the END FUNCTION statement. See the END statement for more information.

Format

FUNCTIONEND [exp]

FUNCTIONEXIT

The FUNCTIONEXIT statement is a synonym for the EXIT FUNCTION statement. See the EXIT statement for more information.

Format

FUNCTIONEXIT [exp]

GET

The GET statement moves a record from a file to a record buffer and makes the data available for processing. GET statements are valid on sequential, relative, indexed, and block I/O files.

Format

GET #chnl-exp [, position-clause]

position-clause: { **RFA** rfa-exp | **RECORD** num-exp | **KEY#** key-clause }

key-clause: int-exp1 rel-op key-exp

rel-op: { **EQ** | **GE** | **GT** }

key-exp { int-exp2 | str-exp }

Syntax Rules

Chnl-exp is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).

Remarks

1. *Position-clause*

   - *Position-clause* specifies the position of a record in a file. BASIC–PLUS–2 signals an error if you specify a *position-clause* and *chnl-exp* is not associated with a disk file. The RECORD *position-clause* is invalid for accessing SEQUENTIAL FIXED files. If you do not specify a *position-clause*, GET retrieves records sequentially. Sequential record access is valid on all RMS files.

- The RFA *position-clause* allows you to randomly retrieve records by specifying the record file address (RFA); You specify the disk address of a record, and RMS retrieves the record at that address. All file organizations can be accessed by RFA.

  Rfa-exp in the RFA *position-clause* is an expression of the RFA data type that specifies the record's file address. An RFA expression must be a variable of the RFA data type or the GETRFA function. Use the GETRFA function to obtain the RFA of a record.

- The RECORD *position-clause* allows you to randomly retrieve records in relative files by specifying the record number.
  - *Num-exp* in the RECORD *position-clause* specifies the number of the record you want to retrieve. It must be between 1 and the number of the record with the highest number in the file.
  - When you specify a RECORD clause, *chnl-exp* must be a channel associated with an open relative file.
- The KEY *position-clause* allows you to randomly retrieve records in indexed files by specifying a key of reference, a relational test, or a key value.

2. *Key-clause*
   - In a *key-clause*, *int-exp1* is the target key of reference. It must be an integer value in the range of zero through the highest-numbered key for the file. The primary key is #0, the first alternate key is #1, the second alternate key is #2, and so on. *Int-exp1* must be preceded by a number sign (#) or BASIC–PLUS–2 signals an error.
   - When you specify a key clause, *chnl-exp* must be a channel associated with an open indexed file.
3. *Rel-op*
   - *Rel-op* specifies how *key-exp* is to be compared with *int-exp1* in the *key-clause*.
     - EQ means *equal to*
     - GE means *greater than or equal to*
     - GT means *greater than*
   - With an *equal to* (EQ) key match, a successful GET operation retrieves the first record in the file that equals the key value specified in *key-exp*. If the key expression is a *str-exp* whose length is less than the key length, characters specified by the *str-exp* are matched approximately

rather than exactly. That is, if you specify a string expression "ABC" and the key length is six characters, BASIC–PLUS–2 matches the first record that begins with ABC. If you specify "ABCABC", BASIC–PLUS–2 matches only a record with the key "ABCABC." If no match is possible, BASIC–PLUS–2 signals the error "Record not found" (ERR=155).

- If you specify a *greater than or equal to* (GE) key match, a successful FIND locates the first record that equals the key value specified in *key-exp*. If no exact match is possible, BASIC–PLUS–2 selects the next record in the key sort order with a higher key value. If no such record exists, BASIC–PLUS–2 signals the error "Record not found" (ERR=155).
- If you specify a *greater than* (GT) key match, a successful GET operation retrieves the first record with a value greater than *key-exp*. If no such record exists, BASIC–PLUS–2 signals the error "Record not found" (ERR=155).

4. *Key-exp*
   - *Int-exp2* in the key clause specifies an integer value to be compared with the key value of a record.
   - *Str-exp* in the key clause specifies a string value to be compared with the key value of a record. The string expression can contain fewer characters than the key of the record you want to retrieve but it cannot be a null string.
5. The file specified by *chnl-exp* must be open with ACCESS READ or MODIFY before your program can execute a GET statement.
6. If the last I/O operation was a successful FIND operation, a sequential GET operation retrieves the current record located by the FIND operation and sets the next record pointer to the record logically succeeding the pointer.
7. If the last I/O operation was not a FIND operation, a sequential GET operation retrieves the next record and sets the record logically succeeding the record pointer to the current record.
   - For sequential files, a sequential GET operation retrieves the next record in the file.
   - For relative files, a sequential GET operation retrieves the record with the next higher cell number.

- For indexed files, a sequential GET operation retrieves the next record in the current key of reference.

8. A successful random GET operation by RFA or by record retrieves the record specified by *rfa-exp* or *int-exp*.
9. A successful random GET operation by key retrieves the first record whose key satisfies the *key-clause* comparison.
10. A successful random GET operation by RFA, record, or key sets the value of the current record pointer to the record just read. The next record pointer is set to the next logical record.
11. An unsuccessful GET operation leaves the record pointers and the record buffer in an undefined state.
12. If the retrieved record is smaller than the receiving buffer, BASIC–PLUS–2 fills the remaining buffer space with nulls.
13. If the retrieved record is larger than the receiving buffer, BASIC–PLUS–2 truncates the record and signals an error.
14. You should not use a GET statement on a terminal-format or virtual array file.
15. When you access a shared file, a successful GET locks the record or bucket and unlocks the previously locked record or bucket.
16. A successful GET operation sets the value of the RECOUNT variable to the number of bytes transferred from the file to the record buffer.

Example

```
10   DECLARE LONG rec-num
     MAP (CUSREC) WORD cus_num                                              &
            STRING cus_nam = 20, cus_add = 20, cus_city = 10, cus_zip = 9
     OPEN "CUS_ACCT.DAT" FOR INPUT AS  #1                                   &
          RELATIVE FIXED, ACCESS MODIFY,                                    &
          MAP CUSREC
     INPUT "Which record number would you like to view";rec_num
     GET #1, RECORD REC_NUM
     PRINT "The customer's number is ";CUS_NUM
     PRINT "The customer's name is ";cus_nam
     PRINT "The customer's address is ";cus_add
     PRINT "The customer's city is ";cus_city
     PRINT "The customer's zip code is ";cus_zip
     CLOSE #1
     END
```

GETRFA

The GETRFA function returns the record's file address (RFA) of the last record accessed in an RMS file open on a specified channel.

Format

rfa-var = **GETRFA**(chnl-exp)

Syntax Rules

1. *Rfa-var* is a variable of the RFA data type.
2. *Chnl-exp* is the channel number of an open RMS file. You cannot precede the *chnl-exp* with a number sign (#).
3. You must access a record in the file with a GET, FIND, or PUT statement before using the GETRFA function, or BASIC–PLUS–2 signals "No current record" (ERR=131).

Remarks

1. There must be a file open on the specified *chnl-exp* or BASIC–PLUS–2 signals an error.
2. You can use the GETRFA function with RMS sequential, relative, indexed, and (except on RSTS/E systems) block I/O files.
3. The RFA value returned by the GETRFA function can be used only for assignments to and comparisons with other variables of the RFA data type. Comparisons are limited to equal to (=) and not equal to (<>) relational operations.
4. RFA values cannot be printed or used for any arithmetic operations.

Example

```
10  DECLARE RFA R_ARRAY(1 TO 100)
    .
    .
    .
    FOR I% = 1% TO 100%
        PUT #1
        R_ARRAY(I%) = GETRFA(1)
    NEXT I%
```

GOSUB

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

Format

{ **GO SUB** } target
{ **GOSUB** }

Syntax Rules

1. *Target* must refer to an existing line number or label in the same program unit as the GOSUB statement or BASIC–PLUS–2 signals an error.
2. *Target* cannot be inside a block structure such as a FOR . . . NEXT, WHILE, or UNTIL loop or a multi-line function definition unless the GOSUB statement is also within that block or function definition.

Remarks

None.

Example

```
10   GOSUB subroutine_1
   .
   .
   .
200  subroutine_1:
   .
   .
   .
     RETURN
```

GOTO

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

Format

{ **GO TO** } target
{ **GOTO** }

Syntax Rules

1. *Target* must refer to an existing line number or label in the same program unit as the GOTO statement or BASIC–PLUS–2 signals an error.
2. *Target* cannot be inside a block structure such as a FOR . . . NEXT, WHILE, or UNTIL loop or a multi-line function definition unless the GOTO statement is also inside that loop or function definition.

Remarks

None.

Example

```
10  IF answer = 0
        THEN GOTO done
    END IF
    .
    .
    .
    done:
         EXIT PROGRAM
```

IF

The IF statement evaluates a conditional expression and transfers program control depending on the resulting value.

There are two formats:

- Conditional
- Statement modifier

Format

1. Conditional

IF cond-exp **THEN** statement . . . [**ELSE** statement . . .]
[statement]... **END IF**

2. Statement Modifier

statement **IF** cond-exp

Syntax Rules

1. The following rules apply to conditional IF statements:
   - *Cond-exp* can be any valid conditional expression.
   - All statements between the THEN keyword and the next ELSE, line number, or END IF are part of the THEN clause. All statements between the keyword ELSE and the next line number or END IF are part of the ELSE clause.
   - BASIC–PLUS–2 assumes a GOTO statement when the keyword ELSE is followed by a line number. When the target of a GOTO statement is a label, the keyword GOTO is required. The use of this syntax is not recommended for new program development.
   - The END IF statement terminates the most recent unterminated IF statement.
   - A new line number terminates all unterminated IF statements.
2. The following rules apply to statement modifier IF statements:
   - IF can modify any executable statement except a block statement such as FOR, WHILE, UNTIL, or SELECT.
   - *Cond-exp* can be any valid conditional expression.

Remarks

1. The following remarks apply to conditional IF statements:
   - BASIC–PLUS–2 evaluates the conditional expression for truth or falsity. If true (nonzero), BASIC–PLUS–2 executes the THEN clause. If false (zero), BASIC–PLUS–2 skips the THEN clause and executes the ELSE clause, if present.
   - The keyword NEXT cannot be in a THEN or ELSE clause unless the FOR or WHILE statement associated with the keyword NEXT is also part of the THEN or ELSE clause.
   - If a THEN or ELSE clause contains a block statement such as a FOR, SELECT, UNTIL, or WHILE, then a corresponding block termination statement such as a NEXT or END must appear in the same THEN or ELSE clause.
   - IF statements can be nested to 12 levels.
   - Any executable statement is valid in the THEN or ELSE clause, including another IF statement. You can include any number of statements in either clause.
   - Execution continues either at the statement following the END IF clause or at the first line number following an unterminated IF statement.

2. BASIC–PLUS–2 executes a statement modifier IF statement only if the conditional expression is true (nonzero).

Example

```
10  IF Update_flag = True
    THEN
          Weekly_salary = New_rate * 40.0
          UPDATE #1
          IF Dept <> New_dept
          THEN
                GET #1, KEY #1 EQ New_dept
                Dept_employees = Dept_employees + 1
                UPDATE #1
          END IF
          PRINT "Update complete"
    ELSE
          PRINT "Skipping update for this employee"
    END IF
```

INPUT

The INPUT statement assigns values from your terminal or from a terminal-format file to program variables.

Format

INPUT [#chnl-exp,] input-item [{ , ; } input-item] . . .

input-item: [strng-const { , ; }] str-var

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Str-const* is the prompt issued for the input string value.
3. *Str-var* is a program variable to which the input string value is assigned.
4. You can include more than one *input-item* in an INPUT statement by separating them with commas (,) or semicolons (;).
5. The comma or semicolon that follows *str-var* has no formatting effect. BASIC–PLUS–2 always advances to a new line when you terminate input with a carriage return.
6. The separator that directly follows *str-const* determines where the question mark prompt (if requested) is displayed and where the cursor is positioned for input.
   - A comma causes BASIC–PLUS–2 to skip to the next print zone and display the question mark unless a SET NO PROMPT statement has been executed. For example:

     ```
     DECLARE STRING your_name
     INPUT "What is your name",your_name
     ```

     The output is:

     ```
     What is your name          ?
     ```

- A semicolon causes BASIC–PLUS–2 to display the question mark next to *str-const* unless a SET NO PROMPT statement has been executed. For example:

```
DECLARE STRING your_name
INPUT "What is your name";your_name
```

The output is:

```
What is your name?
```

7. BASIC–PLUS–2 always advances to a new line when you terminate input with a carriage return.

Remarks

1. If you do not specify a channel, the default *chnl-exp* is #0 (the controlling terminal). If a *chnl-exp* is specified, a file must be open on that channel with ACCESS READ or MODIFY before the INPUT statement can execute.
2. If input comes from a terminal, BASIC–PLUS–2 displays *str-const*, if present. If the terminal is open on channel #0, BASIC–PLUS–2 also displays a question mark (?).
3. You can disable the question mark prompt by using the SET NO PROMPT statement. See the SET PROMPT statement for more information.
4. When BASIC–PLUS–2 receives a line terminator or a complete record, it checks each data element for correct data type and range limits, then assigns the values to the corresponding variables.
5. If you specify a string variable to receive the input text, and the user enters an unquoted string in response to the prompt, BASIC–PLUS–2 ignores the string's leading and trailing spaces and tabs. An unquoted string cannot contain any commas.
6. If there is not enough data in the current record or line to satisfy the variable list, BASIC–PLUS–2 takes one of the following actions:
   - If the input device is a terminal and you have not specified SET NO PROMPT, BASIC–PLUS–2 repeats the question mark but not the *str-const* on a new line until sufficient data is entered.
   - If the input device is not a terminal, BASIC–PLUS–2 signals the error "Not enough data in record" (ERR=59).
7. If there are more data items than variables in the INPUT response, BASIC–PLUS–2 ignores the excess.

8. If there is an error while data is being converted or assigned (for example, string data being assigned to a numeric variable), BASIC–PLUS–2 takes one of the following actions:
   - If there is no error handler in effect and the input device is a terminal, BASIC–PLUS–2 signals a warning, reexecutes the INPUT statement, and displays *str-const* and the input prompt.
   - If there is an error handler in effect and the input device is not a terminal, BASIC–PLUS–2 signals the error "Illegal number" (ERR=52) or "Data format error" (ERR=50).
9. When a RESUME statement transfers control to an INPUT statement, the INPUT statement retrieves a new record or line regardless of any data left in the previous record or line.
10. After a successful INPUT statement, the RECOUNT variable contains the number of characters transferred from the file or terminal to the record buffer.
11. If you terminate input text with Ctrl/Z, BASIC–PLUS–2 assigns the value to the variable and signals the error "End of file on device" (ERR=11) when the next terminal input statement executes. If you are in the BASIC environment and there is no subsequent INPUT, INPUT LINE, or LINPUT statement in the program, the Ctrl/Z is passed to BASIC–PLUS–2 as a signal to exit the BASIC environment. BASIC–PLUS–2 signals the error "Unsaved changes have been made, Ctrl/Z or EXIT to exit" if you have made changes to your program or are running a program that has never been saved. If you have not made changes, BASIC–PLUS–2 exits from the BASIC environment and does not signal an error.

Example

```
10  DECLARE STRING var_1,   &
            INTEGER var_2
    INPUT "The first variable";var_1, "The second variable";var_2
```

The output is:

```
The first variable? name
The second variable? 4
```

INPUT LINE

The INPUT LINE statement assigns a string value, including the line terminator, from a terminal or terminal-format file to a string variable.

Format

INPUT LINE [#chnl-exp,] input-item [{ , ; } input-item] . . .

input-item: [strng-const { , ; }] str-var

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Str-const* is the prompt issued for the input string value.
3. *Str-var* is a program variable to which the input string value is assigned.
4. You can include more than one *input-item* in an INPUT LINE statement by separating them with commas (,) or semicolons (;).
5. The separator (comma or semicolon) that directly follows *str-var* has no formatting effect. BASIC–PLUS–2 always advances to a new line when you terminate input with a carriage return.
6. The separator that directly follows *str-const* determines where the question mark (if requested) is displayed and where the cursor is positioned for input.
   - A comma causes BASIC–PLUS–2 to skip to the next print zone and display the question mark unless a SET NO PROMPT statement has been executed. For example:

     ```
     10  DECLARE STRING your_name
         INPUT LINE "Name",your_name
     ```

     The output is:

     ```
     Name              ?
     ```

- A semicolon causes BASIC–PLUS–2 to display the question mark next to *str-const* unless a SET NO PROMPT statement has been executed. For example:

```
10  DECLARE STRING your_name
    INPUT LINE "Name";your_name
```

The output is:

```
Name?
```

7. BASIC–PLUS–2 always advances to a new line when you terminate input with a carriage return.

Remarks

1. The default *chnl-exp* is #0 (the controlling terminal). If a channel is specified, a file must be open on that channel with ACCESS READ or ACCESS MODIFY before the INPUT LINE statement can execute.
2. BASIC–PLUS–2 signals an error if the INPUT LINE statement has no argument.
3. If input comes from a terminal, BASIC–PLUS–2 displays the contents of *str-const1*, if present. If the terminal is open on channel #0, BASIC–PLUS–2 also displays a question mark (?).
4. You can disable the question mark prompt by using the SET NO PROMPT statement. See the SET PROMPT statement for more information.
5. The INPUT LINE statement assigns all input characters, including any line terminator, to string variables. Single and double quotation marks, commas, tabs, leading and trailing spaces, and other special characters in the string are part of the data.
6. When a RESUME statement transfers control to an INPUT LINE statement, the INPUT LINE statement retrieves a new record or line regardless of any data left in the previous record or line.
7. After a successful INPUT LINE statement, the RECOUNT variable contains the number of characters transferred from the file or terminal to the record buffer.
8. If you terminate input text with Ctrl/Z, BASIC–PLUS–2 assigns the value to the variable and signals the error "End of file on device" (ERR=11) when the next terminal input statement executes. If you are in the BASIC environment and there is no next INPUT, INPUT LINE, or LINPUT statement in the program, the Ctrl/Z is passed to BASIC–PLUS–2 as a signal to exit the BASIC environment. BASIC–PLUS–2 signals the

error "Unsaved changes have been made, Ctrl/Z or EXIT to exit" if you have made changes to your program. If you have not made changes, BASIC–PLUS–2 exits from the BASIC environment and does not signal an error.

Example

```
10  DECLARE STRING Z,N,record_string
    INPUT LINE "Type two words", Z$,'Type your name';N$
    INPUT LINE #4%, record_string$
```

INSTR

The INSTR function searches for a substring within a string. It returns the position of the substring's starting character.

Format

int-var = **INSTR**(int-exp, str-exp1, str-exp2)

Syntax Rules

1. *Int-exp* specifies the character position in the main string at which BASIC–PLUS–2 starts the search.
2. *Str-exp1* specifies the main string.
3. *Str-exp2* specifies the substring.

Remarks

1. The position returned by the INSTR function is the number of characters from the beginning of the string regardless of the value specified in *int-exp*.
2. If *int-exp* is less than 1, INSTR starts its search at the first character of the string.
3. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to an integer of the default size.
4. The INSTR function searches *str-exp1*, the main string, for the first occurrence of a substring, *str-exp2*, and returns the position of the substring's first character.
5. If you know that the substring is not near the beginning of the string, specifying a starting position greater than 1 speeds program execution by reducing the number of characters BASIC–PLUS–2 must search.
6. INSTR returns the character position in the main string at which BASIC–PLUS–2 finds the substring, except in the following situations:
   - If only the substring is null, and if *int-exp* is less than or equal to zero, INSTR returns a value of 1.
   - If only the substring is null, and if *int-exp* is equal to or greater than 1 and less than or equal to the length of the main string, INSTR returns the value of *int-exp*.

- If only the substring is null, and if *int-exp* is greater than the length of the main string, INSTR returns the main string's length plus 1.
- If the substring is not null, and if *int-exp* is greater than the length of the main string, INSTR returns a value of zero.
- If only the main string is null, INSTR returns a value of zero.
- If both the main string and the substring are null, INSTR returns a 1.

7. If BASIC–PLUS–2 cannot find the substring, INSTR returns a value of zero.

Example

```
10  DECLARE STRING alpha,   &
            INTEGER result
    alpha = "ABCDEF"
    result = INSTR(1,alpha,"DEF")
    PRINT result
```

The output is:

```
4
```

INT

The INT function returns the floating-point value of the largest whole number less than or equal to a specified expression.

Format

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

Syntax Rules

None.

Remarks

1. If *real-exp* is negative, BASIC–PLUS–2 returns the largest whole number less than or equal to *real-exp*. For example, INT(–5.3) is –6.
2. BASIC–PLUS–2 expects the argument of the INT function to be a real expression. When the argument is a real expression, BASIC–PLUS–2 returns a value of the same floating-point size. When the argument is not a real expression, BASIC–PLUS–2 converts the argument to the default floating-point size and returns a value of the default floating-point size.

Examples

1.
```
10  DECLARE SINGLE any_num, result
    any_num = 6.667
    result = INT(any_num)
    PRINT result
```

The output is:

```
6
```

2.
```
10  !This example contrasts the INT and FIX functions
    DECLARE SINGLE test_num
    test_num = -32.7
    PRINT "INT OF -32.7 IS: "; INT(test_num)
    PRINT "FIX OF -32.7 IS: "; FIX(test_num)
```

The output is:

```
INT OF -32.7 IS: -33
FIX OF -32.7 IS: -32
```

INTEGER

The INTEGER function converts a numeric expression or numeric string to a specified or default INTEGER data type.

Format

int-var = **INTEGER**(exp [**,BYTE** | **,WORD** | **,LONG**])

Syntax Rules

Exp can be either numeric or string. A string expression can contain the ASCII digits 0 through 9, a plus sign (+), or a minus sign (–).

Remarks

1. BASIC–PLUS–2 evaluates *exp*, then converts it to the specified INTEGER size. If you do not specify a size, BASIC–PLUS–2 uses the default INTEGER size.
2. If *exp* is a string, BASIC–PLUS–2 ignores leading and trailing spaces and tabs.
3. The INTEGER function returns a value of zero when a string argument contains only spaces and tabs, or when it is null.

Example

```
10  INPUT "Enter a floating-point number";F_P
    PRINT INTEGER(F_P, WORD)
```

The output is:

```
Enter a floating-point number? 76.99
 76
```

ITERATE

The ITERATE statement allows you to explicitly reexecute a loop.

Format

ITERATE [label]

Syntax Rules

Label is the name that identifies the FOR . . . NEXT, WHILE, or UNTIL loop. For more information on labels, see Section 1.1.2.

Remarks

1. ITERATE is equivalent to an unconditional branch to the current loop's NEXT statement. If you supply a label, ITERATE transfers control to the NEXT statement in the specified loop. If you do not supply a label, ITERATE transfers control to the current loop's NEXT statement.
2. The ITERATE statement can be used only within a FOR . . . NEXT, WHILE, or UNTIL loop.

Example

```
10  Date_loop:  WHILE 1% = 1%
                    GET #1
                    ITERATE Date_loop IF Day$ <> Today$
                    ITERATE Date_loop IF Month$ <> This_month$
                    ITERATE Date_loop IF Year$ <> This_year$
                    PRINT Item$
                NEXT
```

KILL

The KILL statement deletes a disk file, removes the file's directory entry, and releases the file's storage space.

Format

KILL file-spec

Syntax Rules

File-spec can be a quoted string constant, a string variable, or a string expression. It cannot be an unquoted string constant.

Remarks

1. The KILL statement marks a file for deletion but does not delete the file until all users have closed it.
2. If you do not specify a complete file specification, BASIC–PLUS–2 uses the default device and directory. If you do not specify a file version, BASIC–PLUS–2 on RSX systems deletes the highest version of the file.
3. The file must exist, or BASIC–PLUS–2 signals an error.
4. You can delete a file in another directory if you have access to that directory and privilege to delete the file.

Example

```
10  KILL "TEMP.DAT"
```

LEFT$

The LEFT$ function extracts a specified substring from a string's left side, leaving the main string unchanged.

Format

str-var = **LEFT[$]** *str-exp, int-exp*

Syntax Rules

1. *Int-exp* specifies the number of characters to be extracted from the left side of *str-exp*.
2. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to an integer of the default size.

Remarks

1. The LEFT$ function extracts a substring from the left of the specified *str-exp* and stores it in *str-var*.
2. If *int-exp* is less than 1, LEFT$ returns a null string.
3. If *int-exp* is greater than the length of *str-exp*, LEFT$ returns the entire string.

Example

```
10  DECLARE STRING sub_string, main_string
    main_string = "1234567"
    sub_string = LEFT$(main_string, 4)
    PRINT sub_string
```

The output is:

```
1234
```

LEN

The LEN function returns an integer value equal to the number of characters in a specified string.

Format

int-var = **LEN**(str-exp)

Syntax Rules

None.

Remarks

1. If *str-exp* is null, LEN returns a value of zero.
2. The length of *str-exp* includes leading, trailing, and embedded blanks. Tabs in *str-exp* are treated as a single space.
3. The value returned by the LEN function is a WORD integer.

Example

```
10  DECLARE STRING alpha, &
            INTEGER length
    alpha = "ABCDEFG"
    length = LEN(alpha)
    PRINT length
```

The output is:

```
7
```

LET

The LET statement assigns a value to one or more variables.

Format

[LET] var, . . . = exp

Syntax Rules

The keyword LET is optional.

Remarks

1. You cannot assign string data to a numeric variable or unquoted numeric data to a string variable.
2. The value assigned to a numeric variable is converted to the variable's data type. For example, if you assign a floating-point value to an integer variable, BASIC–PLUS–2 truncates the value to an integer.
3. For dynamic strings, the destination string's length equals the source string's length.
4. When you assign a value to a fixed-length string variable (a variable declared in a COMMON or MAP statement), the value is left-justified and padded with spaces or truncated to match the length of the string variable. Because of this padding, the length of the string is always the length declared in the COMMON or MAP statement. You cannot easily concatenate these strings because of the padding.
5. To add characters to a static string, you must first strip off the trailing blanks with the TRM$ function. For example:

```
100  COMMON A$ = 16
     A$ = "A"
     PRINT A$
     A$ = A$ + "B"
     PRINT A$
     A$ = TRM$(A$) + "B"
     PRINT A$
```

The output is:

```
A
A
AB
```

6. Virtual array strings are of fixed length as declared in the DIMENSION statement. When you assign a value to a virtual array string, it is left-justified and padded with null characters. Therefore, values stored in virtual arrays cannot contain trailing null characters.

Example

```
10  DECLARE STRING alpha, &
            INTEGER length
    LET alpha = "ABCDEFG"
    LET length = LEN(alpha)
    PRINT length
```

The output is:

```
 7
```

LINPUT

The LINPUT statement assigns a string value, without line terminators, from a terminal or terminal-format file to a string variable.

Format

LINPUT [#chnl-exp,] input-item [{ , ; } input-item] . . .

input-item: [strng-const { , ; }] str-var

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Str-const* is the prompt issued for the input value.
3. *Str-var* is a program string variable to which the input value is assigned.
4. You can include more than one *input-item* in an LINPUT statement by separating them with commas (,) or semicolons (;).
5. The separator (comma or semicolon) that directly follows *str-var* has no formatting effect. BASIC–PLUS–2 always advances to a new line when you terminate input with a carriage return.
6. The separator character that directly follows *str-const* determines where the question mark (if requested) is displayed and where the cursor is positioned for input.
   - A comma causes BASIC–PLUS–2 to skip to the next print zone to display the question mark unless a SET NO PROMPT statement has been executed. For example:

     ```
     10  DECLARE STRING your_name
         LINPUT "Name",your_name
     ```

     The output is:

     ```
     Name              ?
     ```

- A semicolon causes BASIC–PLUS–2 to display the question mark next to *str-const* unless a SET NO PROMPT statement has been executed. For example:

```
10  DECLARE STRING your_name
    LINPUT "What is your name";your_name
```

The output is:

```
What is your name?
```

7. BASIC–PLUS–2 always advances to a new line when you terminate input with a carriage return.

Remarks

1. The default *chnl-exp* is #0 (the controlling terminal). If you specify a channel, the file associated with that channel must have been opened with ACCESS READ or MODIFY.
2. BASIC–PLUS–2 signals an error if the LINPUT statement has no argument.
3. If input comes from a terminal, BASIC–PLUS–2 displays the contents of *str-const1*, if present. If the terminal is open on channel #0, BASIC–PLUS–2 also displays a question mark (?).
4. You can disable the question mark prompt by using the SET NO PROMPT statement. See the SET PROMPT statement for more information.
5. The LINPUT statement assigns all characters, except a line terminator, to *str-var1*. Single and double quotation marks, commas, tabs, leading and trailing spaces, or other special characters in the string are part of the data.
6. If a RESUME statement transfers control to a LINPUT statement, the LINPUT statement retrieves a new record regardless of any data left in the previous record.
7. After a successful LINPUT statement, the RECOUNT variable contains the number of bytes transferred from the file or terminal to the record buffer.
8. If you terminate input text with Ctrl/Z, BASIC–PLUS–2 assigns the value to the variable and signals the error "End of file on device" (ERR=11) when the next terminal input statement executes. If you are in the BASIC environment and there is no next INPUT, INPUT LINE, or LINPUT statement in the program, the Ctrl/Z is passed to BASIC–PLUS–2 as a signal to exit the BASIC environment.

Example

```
10  DECLARE STRING last_name
    LINPUT "ENTER YOUR LAST NAME";Last_name
    LINPUT #2%, Last_name
```

LOG

The LOG function returns the natural logarithm (base *e*) of a specified number. The LOG function is the inverse of the EXP function.

Format

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

Syntax Rules

None.

Remarks

1. *Real-exp* must be greater than zero. An attempt to find the logarithm of zero or a negative number causes BASIC–PLUS–2 to signal "Illegal argument in LOG" (ERR=53).
2. The LOG function uses the mathematical constant *e* as a base. BASIC–PLUS–2 approximates *e* to be 2.718281828459045 (double precision).
3. The LOG function returns the exponent to which *e* must be raised to equal *real-exp*.
4. BASIC–PLUS–2 expects the argument of the LOG function to be a real expression. When the argument is a real expression, BASIC–PLUS–2 returns a value of the same floating-point size. When the argument is not a real expression, BASIC–PLUS–2 converts the argument to the default floating-point size and returns a value of the default floating-point size.

Example

```
10  DECLARE SINGLE exponent
    exponent = LOG(98.6)
    PRINT exponent
```

The output is:

```
4.59107
```

LOG10

The LOG10 function returns the common logarithm (base 10) of a specified number.

Format

real-var = **LOG10**(real-exp)

Syntax Rules

None.

Remarks

1. *Real-exp* must be larger than zero. An attempt to find the logarithm of zero or a negative number causes BASIC–PLUS–2 to signal "Illegal argument in LOG" (ERR=53).
2. The LOG10 function returns the exponent to which 10 must be raised to equal *real-exp*.
3. BASIC–PLUS–2 expects the argument of the LOG10 function to be a real expression. When the argument is a real expression, BASIC–PLUS–2 returns a value of the same floating-point size. When the argument is not a real expression, BASIC–PLUS–2 converts the argument to the default floating-point size and returns a value of the default floating-point size.

Example

```
10  DECLARE SINGLE exp_base_10
    exp_base_10 = LOG10(250)
    PRINT exp_base_10
```

The output is:

```
2.39794
```

LSET

The LSET statement assigns left-justified data to a string variable. LSET does not change the length of the destination string variable.

Format

LSET str-var, . . . = str-exp

Syntax Rules

Str-var is the destination string. *Str-exp* is the string value assigned to *str-var.*

Remarks

1. The LSET statement treats all strings as fixed length. LSET does not change the length of the destination string or create new storage. LSET does, however, overwrite the current storage of *str-var.*
2. If the destination string is longer than *str-exp*, LSET left-justifies *str-exp* and pads it with spaces on the right. If smaller, LSET truncates characters from the right of *str-exp* to match the length of *str-var.*
3. With string virtual arrays, LSET changes the length of *str-exp* to the declared length by padding it with spaces on the right. Note that the LET statement uses null characters for padding.

Example

```
10  DECLARE STRING alpha
    alpha = "ABCDE"
    LSET alpha = "FGHIJKLMN"
    PRINT alpha
```

The output is:

```
FGHIJ
```

MAG

The MAG function returns the absolute value of a specified expression. The returned value has the same data type as that of the expression.

Format

var = **MAG**(exp)

Syntax Rules

None.

Remarks

1. The returned value is always greater than or equal to zero. The absolute value of zero is zero. The absolute value of a positive number equals that number. The absolute value of a negative number equals that number multiplied by –1.
2. The MAG function is similar to the ABS function in that it returns the absolute value of a number. The ABS function, however, takes a floating-point argument and returns a floating-point value. The MAG function takes an argument of any numeric data type and returns a value of the same data type as the argument. It is recommended that you use the MAG function rather than the ABS and ABS% functions, because the MAG function returns a value using the data type of the argument.

Example

```
10  DECLARE SINGLE A
    A = -34.6
    PRINT MAG(A)
```

The output is:

```
34.6
```

MAGTAPE

The MAGTAPE function permits your program to control unformatted magnetic tape files.

Format

int-var1 = **MAGTAPE**(func-code, int-var2, chnl-exp)

Syntax Rules

1. *Func-code* is an integer from 1 through 9 that specifies the code for the MAGTAPE function you want to perform. MAGTAPE function codes are described in Table 4–5.
2. *Int-var1* is the value returned by function codes 4, 5, 7, and 9.
3. *Int-var2* is an integer parameter for function codes 4, 5, and 6.
   - *Int-var2* for function 4 is a value from 1 through 32767 that specifies the number of records to skip.
   - *Int-var2* for function 5 is a value from 1 through 32767 that specifies the number of records to backspace.
   - *Int-var2* for function 6 specifies the density and/or parity of the magnetic tape drive.
4. *Chnl-exp* is a numeric expression that specifies a channel number associated with the magnetic tape file. You cannot precede the *chnl-exp* with a number sign (#).

Table 4–5 MAGTAPE Function Codes

Code	Function
1	Rewind and take tape offline
2	Write EOF
3	Rewind tape
4	Skip records
5	Backspace

(continued on next page)

Table 4–5 (Cont.) MAGTAPE Function Codes

Code	Function
6	Set density or set parity
7	Get status
8 †	Get characteristics of file
9 †	Rewind once file is closed

†RSTS/E systems only.

5. On RSTS/E systems, MAGTAPE function 9 must be specified after the OPEN statement and before the CLOSE statement associated with the specified magnetic tape.

Remarks

1. You cannot use the MAGTAPE function with RMS files.
2. If the specified function code is 1,2,3,6, or 9, *int-var1* always equals zero.
3. If the specified function code is 4, *int-var1* is an integer of the default size that equals the number of records to skip over.
4. If the specified function code is 5, *int-var1* is an integer of the default size that equals the number of records to backspace over.
5. If the specified function code is 7, *int-var1* is a 16-bit integer that reflects the status of the specified magnetic tape. See the *BASIC–PLUS–2 User's Guide* for information on bit values and their meaning.
6. If the specified function code is 8, *int-var1* is a 16-bit integer that describes the file characteristics of the specified magnetic tape. See the *BASIC–PLUS–2 User's Guide* for information on bit values and their meaning.
7. If the specified function code is 9, the tape is rewound when the file is closed.

Example

```
20  I = MAGTAPE (3%,0%,2%)
```

MAP

The MAP statement defines a named area of statically allocated storage called a PSECT, declares data fields in the record, and associates them with program variables.

Format

MAP (map-name) { [data-type] map-item }, . . .

map-item: { num-unsubs-var | num-array-name (int-const, . . .) | str-unsubs-var [= int-const] | str-array-name (int-const1, . . .) [= int-const] | **FILL** [(int-const)] [= int-const] | **FILL%** [(int-const)] | **FILL$** [(int-const)] [= int-const] }

Syntax Rules

1. *Map-name* is global to the program and image. It cannot appear elsewhere in the program unit as a variable name.
2. *Map-name* can be from one through six characters. The first character of the name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), or periods (.).
3. *Data-type* can be any BASIC–PLUS–2 data type keyword. Data type keywords, size, range, and precision are listed in Table 1-2.
4. When you specify a data type, all following *map-items*, including FILL items, are of that data type until you specify a new data type.
5. If you specify a dollar sign ($) or percent sign (%) suffix character, the variable must be a string or integer data type, respectively.
6. If you do not specify a data type, all following *map-items* take the current default data type and size.
7. *Map-item* declares the name and format of the data to be stored.
   - *Num-unsubs-var* and *num-array-name* specify a numeric variable or a numeric array.

- *Str-unsubs-var* and *str-array-name* specify a fixed-length string variable or array. You can specify the number of bytes to be reserved for the variable with the =*int-const* clause. The default string length is 16.
- The FILL, FILL%, and FILL$ keywords allow you to reserve parts of the record buffer within or between data elements and to define the format of the storage. *Int-const* specifies the number of FILL items to be reserved. The =*int-const* clause allows you to specify the number of bytes to be reserved for string FILL items. Table 4–2 describes FILL item format and storage allocation.
- In the applicable formats of FILL, (*int-const*) represents a repeat count, not an array subscript. FILL (*n*), for example, represents *n* elements, not *n* + 1.

8. Variable names, array names, and FILL items, following a data type other than STRING cannot end with a dollar sign. Variable names, array names, and FILL items, following a data type other than BYTE, WORD, LONG, or INTEGER, cannot end with a percent sign.
9. Variables and arrays declared in a MAP statement cannot be declared elsewhere in the program by any other declarative statement.

Remarks

1. BASIC–PLUS–2 does not execute MAP statements. The MAP statement allocates static storage and defines data at compilation time.
2. A program can have multiple maps with the same name. The allocation for each map overlays the others. Thus, data is accessible in many ways. The actual size of the data area is the size of the largest map. When you link your program, the size of the map area is the size of the largest map with that name.
3. *Map-items* with the same name can appear in different MAP statements with the same map name only if they match exactly in attributes such as data type, position, and so forth. If the attributes are not the same, BASIC–PLUS–2 signals an error. For example:

```
10  MAP (ABC) LONG A, B
    MAP (ABC) LONG A, C ! This MAP statement is valid
    MAP (ABC) LONG B, A ! This MAP statement produces an error
    MAP (ABC) WORD A, B ! This MAP statement produces an error
```

The third MAP statement causes BASIC–PLUS–2 to signal the error "variable <name> not aligned in multiple references in MAP <name>," while the fourth MAP statement generates the error "Attributes of overlaid variable <name> don't match."

4. The MAP statement should precede any reference to variables declared in it.
5. Storage space for *map-items* is allocated in order of occurrence in the MAP statement.
6. A MAP area can be accessed by more than one program module, as long as you define the *map-name* in each module that references the MAP.
7. A COMMON area and a MAP area with the same name specify the same storage area and are not allowed in the same program module; however, a COMMON in one module can reference the storage declared by a MAP or COMMON in another module.
8. Variables in a MAP statement are initialized to zero or a null string.
9. A map named in an OPEN statement's MAP clause is associated with that file. The file's records and record fields are defined by that map. The size of the map determines the record size for file I/O, unless the OPEN statement includes a RECORDSIZE clause.
10. The allocation for a MAP cannot exceed 32767 bytes or BASIC–PLUS–2 signals the error "COMMON/MAP <name> is too large."

Example

```
10  MAP (BUF1) BYTE AGE, STRING emp_name = 20          &
        SINGLE emp_num
    MAP (BUF1) BYTE FILL, STRING last_name = 12,       &
        FILL = 8, SINGLE FILL
```

MAP DYNAMIC

The MAP DYNAMIC statement names the variables and arrays whose size and position in a storage area can change at run time. The MAP DYNAMIC statement is used in conjunction with the REMAP statement. The REMAP statement defines or redefines the position in the storage area of variables named in the MAP DYNAMIC statement.

Format

MAP DYNAMIC (map-dyn-name){[data-type]map-item }, . . .

map-dyn-name: { map-name | static-str-var }

map-item: { num-unsubs-var | num-array-name (int-const, . . .) | str-unsubs-var | str-array-name (int-const, . . .) }

Syntax Rules

1. *Map-dyn-name* can be either a map name or a static string variable.
   - *Map-name* is the storage area named in a MAP statement.
   - If you specify a map name, then a MAP statement with the same name must precede both the MAP DYNAMIC statement and the REMAP statement.
   - When you specify a static string variable, the string must be declared before you can specify a MAP DYNAMIC statement or a REMAP statement.
   - *Static-str-var* must specify a static string variable or a string parameter variable.
   - If you specify a *static-str-var*, the following restrictions apply:
     - *Static-str-var* cannot be a string constant.
     - *Static-str-var* cannot be the same as any previously declared *map-item* in a MAP DYNAMIC statement.
     - *Static-str-var* cannot be a subscripted variable.

2. *Map-item* declares the name and data type of the items to be stored in the storage area. All variable pointers point to the beginning of the storage area until the program executes a REMAP statement.
   - *Num-unsubs-var* and *num-array-name* specify a numeric variable or a numeric array.
   - *Str-unsubs-var* and *str-array-name* specify a string variable or array. You cannot specify the number of bytes to be reserved for the variable in the MAP DYNAMIC statement. All string items have a fixed length of zero until the program executes a REMAP statement.
3. *Data-type* can be any BASIC–PLUS–2 data type keyword. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.
4. When you specify a data type, all following *map-items* are of that data type until you specify a new data type.
5. If you do not specify any data type, *map-items* take the current default data type and size.
6. *Map-items* must be separated with commas.
7. If you specify a dollar sign or percent sign suffix, the variable must be either a STRING data type or an integer data type, respectively.

Remarks

1. All variables and arrays declared in a MAP DYNAMIC statement cannot be declared elsewhere in the program by any other declarative statements.
2. The MAP DYNAMIC statement does not affect the amount of storage allocated to the map buffer declared in a previous MAP statement or the storage allocated to a static string. Until your program executes a REMAP statement, all variable and array element pointers point to the beginning of the MAP buffer or static string.
3. BASIC–PLUS–2 does not execute MAP DYNAMIC statements. The MAP DYNAMIC statement names the variables whose size and position in the MAP or static string buffer can change and defines their data type.
4. Before you can specify a map name in a MAP DYNAMIC statement, there must be a MAP statement in the program unit with the same map name; otherwise, BASIC–PLUS–2 signals the error "MAP DYNAMIC <map-name> requires MAP or static string." Similarly, before you can specify a static string variable in the MAP DYNAMIC statement, the string variable must be declared; otherwise, BASIC–PLUS–2 signals the same error message.

5. A static string variable must be either a variable declared in a MAP or COMMON statement or a parameter declared in a SUB or FUNCTION. It cannot be a parameter declared in a DEF or DEF* function.
6. The MAP DYNAMIC statement must lexically precede the REMAP statement or BASIC–PLUS–2 signals the error "MAP variable <name> referenced before declaration."

Example

```
100     MAP (MY.BUF) STRING DUMMY = 512
        MAP DYNAMIC (MY.BUF) STRING LAST, FIRST, MIDDLE,      &
                             BYTE AGE, STRING EMPLOYER,       &
                             STRING CHARACTERISTICS
```

MAT

The MAT statement lets you implicitly create and manipulate one- and two-dimensional arrays. You can use the MAT statement to assign values to array elements, or to redimension a previously dimensioned array. You can also perform matrix arithmetic operations such as multiplication, addition, and subtraction, and other matrix operations such as transposing and inverting matrices.

There are five formats:

- For numeric initialization
- For string initialization
- For array arithmetic
- For scalar multiplication
- For inversion and transposition

Format

1. Numeric Initialization

MAT num-array = { **CON** | **IDN** | **ZER** } [(int-exp1 [, int-exp2])]

2. String Initialization

MAT str-array = **NUL$** [(int-exp1 [, int-exp2])]

3. Array Arithmetic

MAT num-array1 = num-array2 [{ + | – | * } num-array3]

4. Scalar Multiplication

MAT num-array4 = (num-exp) * num-array5

5. Inversion and Transposition

MAT num-array6 = { **TRN** | **INV** } (num-array7)

Syntax Rules

1. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the new dimensions of an existing array.
2. You cannot use the MAT statement on arrays of more than two dimensions.
3. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.
4. If you do not specify bounds, BASIC–PLUS–2 creates the array and dimensions it to (10) or (10,10).
5. If you specify bounds, BASIC–PLUS–2 creates the array with the specified bounds. If the bounds exceed (10) or (10,10), BASIC–PLUS–2 signals the error "Redimensioned array" (ERR=105).

Remarks

1. To perform MAT operations on arrays larger than (10,10), create the input and output arrays with the DIM statement.
2. You cannot increase the number of array elements or change the number of dimensions in an array when you redimension with the MAT statement. For example, you can redimension an array with dimensions (5,4) to (4,5) or (3,2), but you cannot redimension that array to (5,5) or to (10). The total number of array elements includes those in row and column zero.
3. If an array is named in both a DIM statement and a MAT statement, the DIM statement must lexically precede the MAT statement.
4. MAT statements do not operate on elements in the zero element (one-dimensional arrays) or in the zero row or column (two-dimensional arrays). MAT statements use these elements to store results of intermediate calculations. Therefore, you should not depend on values in row and column zero if your program uses MAT statements.
5. When the array exists, the following rules apply:
   - If you specify bounds, BASIC–PLUS–2 redimensions the array to the specified size; however, MAT operations cannot increase the total number of array elements.
   - If you do not specify bounds, BASIC–PLUS–2 does not redimension the array.
6. **Initialization**
   - CON sets all elements of *num-array* to 1, except those in row and column zero.

- IDN creates an identity matrix from *num-array*. The number of rows and columns in *num-array* must be identical. IDN sets all elements to zero except those on the diagonal from *num-array*(1,1) to *num-array*(n,n), which are set to 1.
- ZER sets all array elements to zero, except those in row and column zero.
- NUL$ sets all elements of a string array to the null string, except those in row and column zero.

7. **Array Arithmetic**
   - The equal sign (=) assigns the results of the specified operation to the elements in *num-array1*.
   - If *num-array3* is not specified, BASIC–PLUS–2 assigns the values of *num-array2*'s elements to the corresponding elements of *num-array1*. *Num-array1* must have at least as many rows and columns as *num-array2*.
   - Use the plus sign (+) to add the elements of two arrays. *Num-array2* and *num-array3* must have identical bounds.
   - Use the minus sign (–) to subtract the elements of two arrays. *Num-array2* and *num-array3* must have identical bounds.
   - Use the asterisk (*) to perform matrix multiplication on the elements of *num-array2* and *num-array3* and to assign the results to *num-array1*. This operation gives the dot product of *num-array2* and *num-array3*. All three arrays must be two-dimensional, and the number of columns in *num-array2* must equal the number of rows in *num-array3*. BASIC–PLUS–2 redimensions *num-array1* to have the same number of rows as *num-array2* and the same number of columns as *num-array3*.
   - You cannot perform matrix multiplication with the same array as both the source and destination, or BASIC–PLUS–2 signals an error.
8. **Scalar Multiplication**

   BASIC–PLUS–2 multiplies each element of *num-array5* by *num-exp* and stores the results in the corresponding elements of *num-array4*.
9. **Inversion and Transposition**
   - TRN transposes *num-array7* and assigns the results to *num-array6*. If *num-array7* has *m* rows and *n* columns, *num-array6* will have *n* rows and *m* columns. Both arrays must be two-dimensional.

- You cannot transpose a matrix to itself: MAT A = TRN(A) is invalid.
- INV inverts *num-array7* and assigns the results to *num-array6*. *Num-array7* must be a two-dimensional array that can be reduced to the identity matrix with elementary row operations. The row and column dimensions must be identical.
- You cannot use the MAT statement to invert an array with a datatype of LONG or BYTE, or to invert an array received as a parameter.

Examples

1.
```
10  !Numeric Initialization
    MAT CONVERT = zer(10,10)
```

2.
```
10  !Initialization
    MAT na_me$ = NUL$(5,5)
```

3.
```
!Array Arithmetic
MAT new_int = old_int - rslt_int
```

4.
```
10  !Scalar Multiplication
    MAT Z40 = (4.24) * Z
```

5.
```
10  !Inversion and Transposition
    MAT Q% = INV (Z)
```

MAT INPUT

The MAT INPUT statement assigns values from a terminal or terminal-format file to array elements.

Format

MAT INPUT [#chnl-exp,] { array [(int-exp1 [, int-exp2])] }, . . .

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. The file associated with *chnl-exp* must be an open terminal-format file or terminal. If *chnl-exp* is not specified, BASIC–PLUS–2 takes data from the controlling terminal.
3. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.
4. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.

Remarks

1. You cannot use the MAT INPUT statement on arrays of more than two dimensions.
2. If you do not specify bounds, BASIC–PLUS–2 creates the array and dimensions it to (10,10).
3. If you do specify bounds, BASIC–PLUS–2 creates the array with the specified bounds. If the bounds exceed (10) or (10,10), BASIC–PLUS–2 signals the error "Redimensioned array" (ERR=105).
4. To use the MAT INPUT statement with arrays larger than (10,10), create the input and output arrays with the DIM statement.
5. When the array exists, the following rules apply:
   - If you specify bounds, BASIC–PLUS–2 redimensions the array to the specified size; however, MAT INPUT cannot increase the total number of array elements.
   - If you do not specify bounds, BASIC–PLUS–2 does not redimension the array.

6. Unless a SET NO PROMPT statement has been executed, the MAT INPUT statement prompts with a question mark on terminals open on channel #0 only. See the description of the SET PROMPT statement for more information.
7. Use commas to separate data elements and a line terminator to end the input of data. Use an ampersand (&) before the line terminator to continue data over more than one line.
8. The MAT INPUT statement assigns values by row. For example, it assigns values to all elements in row 1 before beginning row 2.
9. The MAT INPUT statement assigns the row number of the last data element transferred into the array to the system variable NUM.
10. The MAT INPUT statement assigns the column number of the last data element transferred into the array to the system variable NUM2.
11. If there are fewer elements in the input data than there are array elements, BASIC–PLUS–2 does not change the remaining array elements.
12. If there are more data elements in the input stream than there are array elements, BASIC–PLUS–2 ignores the excess.
13. Row zero and column zero are not changed.

Example

```
10  MAT INPUT XYZ(5,5)
    MAT PRINT XYZ;
```

The output is:

```
? 1,2,3,4,5
 1  2  3  4  5
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
```

MAT LINPUT

The MAT LINPUT statement receives string data from a terminal or terminal-format file and assigns it to string array elements.

Format

MAT LINPUT [#chnl-exp,] { str-array [(int-exp1 [, int-exp2])] }, . . .

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file or terminal. It must be immediately preceded by a number sign (#).
2. The file associated with *chnl-exp* must be an open terminal-format file or terminal. If a channel is not specified, BASIC–PLUS–2 takes data from the controlling terminal.
3. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.
4. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.

Remarks

1. You cannot use the MAT LINPUT statement on arrays of more than two dimensions.
2. If you do not specify bounds, BASIC–PLUS–2 creates the array and dimensions it to (10,10).
3. If you do specify upper bounds, BASIC–PLUS–2 creates the array with the specified bounds. If the bounds exceed (10) or (10,10), BASIC–PLUS–2 signals the error "Redimensioned array" (ERR=105).
4. To use MAT LINPUT with arrays larger than (10,10), create the input and output arrays with the DIM statement.
5. When the array exists, the following rules apply:
   - If you specify bounds, BASIC–PLUS–2 redimensions the array to the specified size; however, MAT LINPUT cannot increase the total number of array elements.
   - If you do not specify bounds, BASIC–PLUS–2 does not redimension the array.

6. Unless a SET NO PROMPT statement has been executed, the MAT LINPUT statement prompts with a question mark for each string array element for terminals open on channel zero. BASIC–PLUS–2 assigns values to all elements of row 1 before beginning row 2.
7. The MAT LINPUT statement assigns the row number of the last data element transferred into the array to the system variable NUM.
8. The MAT LINPUT statement assigns the column number of the last data element transferred into the array to the system variable NUM2.
9. Entering only a line terminator in response to the question mark prompt causes BASIC–PLUS–2 to assign a null string to that string array element.
10. Like the MAT INPUT statement, the MAT LINPUT statement normally accepts only a single line of input from an input file. To supply more than one line of input, you must use an ampersand (&) before the line terminator.
11. MAT LINPUT does not change row and column zero.

Example

```
10  DIM cus_rec$(3,3)
    MAT LINPUT cus_rec$(2,2)
    PRINT cus_rec$(1,1)
    PRINT cus_rec$(1,2)
    PRINT cus_rec$(2,1)
    PRINT cus_rec$(2,2)
```

The output is:

```
? Babcock
? Santani
? Lloyd
? Kelly
Babcock
Santani

Lloyd
Kelly
```

MAT PRINT

The MAT PRINT statement prints the contents of a one- or two-dimensional array on your terminal or assigns the value of each array element to a record in a terminal-format file.

Format

MAT PRINT [#chnl-exp,] { array [(int-exp1 [, int-exp2])] [, ;] } . . .

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file or terminal. It must be immediately preceded by a number sign (#).
2. The file associated with *chnl-exp* must be an open terminal-format file or terminal. If you do not specify a channel, BASIC–PLUS–2 takes data from the controlling terminal.
3. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.
4. The separator (comma or semicolon) determines the output format for the array:
   - If you use a comma, BASIC–PLUS–2 prints each array element in a new print zone and starts each row on a new line.
   - If you use a semicolon, BASIC–PLUS–2 separates each array element with a space and starts each row on a new line.
   - If you do not use a separator character, BASIC–PLUS–2 prints each array element on its own line.

Remarks

1. You cannot use the MAT PRINT statement on arrays of more than two dimensions.
2. When you use the MAT PRINT statement to print more than one array, each array name except the last must be followed with either a comma or a semicolon. BASIC–PLUS–2 prints a blank line between arrays.

3. If the array does not exist, the following rules apply:
   - If you do not specify bounds, BASIC–PLUS–2 creates the array and dimensions it to (10,10).
   - If you specify bounds, BASIC–PLUS–2 creates the array with the specified bounds. If the bounds exceed (10) or (10,10), BASIC–PLUS–2 prints the elements (10) or (10,10), and signals the error "Subscript out of range" (ERR=55).
4. When the array exists, the following rules apply:
   - If the specified bounds are smaller than the maximum bounds of a dimensioned array, BASIC–PLUS–2 prints a subset of the array, but does not redimension the array. For example, if you use the DIM statement to dimension A(20,20), and then MAT PRINT A(2,2), BASIC–PLUS–2 prints elements (1,1), (1,2), (2,1), and (2,2) only; array A(20,20) does not change.
   - If you do not specify bounds, BASIC–PLUS–2 prints the entire array.
5. The MAT PRINT statement does not print elements in row or column zero.
6. The MAT PRINT statement cannot redimension an array.

Example

```
10  DIM cus_rec$(3,3)
    MAT LINPUT cus_rec$(2,2)
    MAT PRINT cus_rec$(2,2)
```

The output is:

```
? Babcock
? Santani
? Lloyd
? Kelly
Babcock
Santani

Lloyd
Kelly
```

MAT READ

The MAT READ statement assigns values from DATA statements to array elements.

Format

MAT READ { array [(int-exp1 [, int-exp2])] }, . . .

Syntax Rules

1. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.
2. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.

Remarks

1. If you do not specify bounds, BASIC–PLUS–2 creates the array and dimensions it to (10) or (10,10).
2. If you specify bounds, BASIC–PLUS–2 creates the array with the specified bounds. If the bounds exceed (10) or (10,10), BASIC–PLUS–2 signals the error "Redimensioned array" (ERR=105).
3. To read arrays larger than (10,10), create the array with the DIM statement.
4. When the array exists, the following rules apply:
   - If you specify bounds, BASIC–PLUS–2 redimensions the array to the specified size; however, MAT READ cannot increase the total number of array elements.
   - If you do not specify bounds, BASIC–PLUS–2 does not redimension the array.
5. All the DATA statements must be in the same program unit as the MAT READ statement.
6. The MAT READ statement assigns data items by row. For example, it assigns data items to all elements in row 1 before beginning row 2.
7. The MAT READ statement does not read elements into row or column zero.

8. The MAT READ statement assigns the row number of the last data element transferred into the array to the system variable NUM.
9. The MAT READ statement assigns the column number of the last data element transferred into the array to the system variable NUM2.
10. If you use MAT READ for an existing array without specifying bounds, BASIC–PLUS–2 does not redimension the array. If you use MAT READ for an existing array and specify bounds, BASIC–PLUS–2 redimensions the array.
11. You cannot use the MAT READ statement on arrays of more than two dimensions.

Example

```
10  MAT READ A(3,3)
    MAT READ B(3,3)
    PRINT
    PRINT "Matrix A"
    PRINT
    MAT PRINT A;
    PRINT
    PRINT "Matrix B"
    PRINT
    MAT PRINT B;
    DATA 1,2,3,4,5,6
```

The output is:

```
Matrix A

 1  2  3
 4  5  6
 0  0  0

Matrix B

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

MAX

The MAX function compares the values of two or more numeric expressions and returns the highest value.

Format

num-var = **MAX** *(num-exp1, num-exp2 [,num-exp3 , . . .])*

Syntax Rules

BASIC–PLUS–2 allows you to specify up to eight numeric expressions.

Remarks

1. If you specify values with different data types, BASIC–PLUS–2 performs data type conversions to maintain precision.
2. BASIC–PLUS–2 returns a function result whose data type is compatible with the values you supply.

Example

```
20  DECLARE REAL John_grade, &
                 Bob_grade,  &
                 Joe_grade,  &
                 highest_grade
30  INPUT "John's grade";John_grade
40  INPUT "Bob's grade";Bob_grade
50  INPUT "Joe's grade";Joe_grade
60  highest_grade = MAX(John_grade, Bob_grade, Joe_grade)
70  PRINT "The highest grade is";highest_grade
```

The output is:

```
John's grade? 90
Bob's grade? 95
Joe's grade? 79
The highest grade is 95
```

MID$

The MID$ function extracts a specified substring from a string expression.

Format

str-var = **MID[$]** *(str-exp, int-exp1, int-exp2)*

Syntax Rules

1. *Int-exp1* specifies the position of the substring's first character.
2. *Int-exp2* specifies the length of the substring.

Remarks

1. If *int-exp1* is less than 1, MID$ assumes a starting character position of 1.
2. If *int-exp2* is less than or equal to zero, MID$ assumes a length of zero.
3. If you specify a floating-point expression for *int-exp1* or *int-exp2*, MID$ truncates it to a WORD integer.
4. The MID$ function extracts a substring from *str-exp* and stores it in *str-var*.
5. If *int-exp1* is greater than the length of *str-exp*, MID$ returns a null string.
6. If *int-exp2* is greater than the length of *str-exp*, the returned string begins at *int-exp1* and includes all characters remaining in *str-exp*.
7. If *int-exp2* is less than or equal to zero, MID$ returns a null string.

Example

```
10  !MID$ Function
    DECLARE STRING old_string, new_string
    old_string = "ABCD"
    new_string = MID$(old_string,1,3)
    PRINT new_string
```

The output is:

```
ABC
```

MIN

The MIN function compares the values of two or more numeric expressions and returns the smallest value.

Format

num-var = **MIN** *(num-exp1, num-exp2 [,num-exp3 , . . .])*

Syntax Rules

BASIC–PLUS–2 allows you to specify up to eight numeric expressions.

Remarks

1. If you specify values with different data types, BASIC–PLUS–2 performs data type conversions to maintain precision.
2. BASIC–PLUS–2 returns a function result whose data type is compatible with the values you supply.

Example

```
20  DECLARE REAL John_grade, &
                Bob_grade,  &
                Joe_grade,  &
                lowest_grade
30  INPUT "John's grade";John_grade
40  INPUT "Bob's grade";Bob_grade
50  INPUT "Joe's grade";Joe_grade
60  lowest_grade = MIN(John_grade, Bob_grade, Joe_grade)
70  PRINT "The lowest grade is";lowest_grade
```

The output is:

```
John's grade? 95
Bob's grade? 100
Joe's grade? 84
The lowest grade is 84
```

MOD

The MOD function divides a numeric value by another numeric value and returns the remainder.

Format

num-var = **MOD** *(num-exp1, num-exp2)*

Syntax Rules

None.

Remarks

1. Num-exp1 is divided by num-exp2.
2. If you specify values with different data types, BASIC–PLUS–2 performs data type conversions to maintain precision.
3. BASIC–PLUS–2 returns a function result whose data type is compatible with the values you supply.
4. The function result is either a positive or negative value, depending on the value of the first numeric expression. For example, if the first numeric expression is negative, then the function result will also be negative.

Example

```
10  DECLARE REAL A,B
20  A = 500
30  B = MOD(A,70)
40  PRINT "The remainder equals";B
```

The output is:

```
The remainder equals 10
```

MOVE

The MOVE statement transfers data between a record buffer and a list of variables.

Format

MOVE { **TO** / **FROM** } #chnl-exp, move-item, . . .

move-item: { num-var | num-array ([,] . . .) | str-var [= int-exp] | str-array ([,] . . .) [= int-exp] | [data-type] **FILL** [(int-exp)] [= int-const] | **FILL%** [(int-exp)] | **FILL$** [(int-exp)] [= int-exp] }

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Move-item* specifies the variable or array to which or from which data is to be moved.
3. Parentheses indicate the number of dimensions in a numeric array. The number of dimensions is equal to the number of commas plus 1. Empty parentheses indicate a one-dimensional array, one comma indicates a two-dimensional array, and so on.
4. *Str-var* and *str-array* specify a fixed-length string variable or array. Parentheses indicate the number of dimensions in a string array. The number of dimensions is equal to the number of commas plus 1. You can specify the number of bytes to be reserved for the variable or array elements with the =*int-exp* clause. The default string length for a MOVE FROM statement is 16. For a MOVE TO statement, the default is the string's length.

5. The FILL, FILL%, and FILL$ keywords allow you to transfer fill items of a specific data type. Table 4–2 shows FILL item formats, representations, and storage requirements.

   - If you specify a data type before the FILL keyword, the fill is of that data type. If you do not specify a data type, the fill is of the default data type. *Data-type* can be any BASIC–PLUS–2 data type keyword. Data type keywords, size, range, and precision are listed in Table 1-2 in this manual.
   - FILL items following a data type other than STRING cannot end with a dollar sign. FILL items following a data type other than BYTE, WORD, LONG, or INTEGER cannot end with a percent sign.
   - *Int-exp* specifies the number of FILL items to be moved.
   - FILL% indicates integer fill. FILL$ indicates string fill. The *=int-exp* clause specifies the number of bytes to be moved for string FILL items.
   - In the applicable formats of FILL, *(int-exp)* represents a repeat count, not an array subscript. FILL (*n*), for example, represents *n* elements, not *n* + 1.

6. You cannot use an expression or function reference as a *move-item*.

Remarks

1. Before a MOVE FROM statement can execute, the file associated with *chnl-exp* must be open and there must be a record in the record buffer.
2. A MOVE statement neither transfers data to or from external devices, nor invokes the Record Management Services (RMS). Instead, it transfers data between user areas. Thus, a record should first be fetched with the GET statement before you use a MOVE FROM statement, and a MOVE TO statement should be followed by a PUT or UPDATE statement that writes the record to a file.
3. MOVE FROM transfers data from the record buffer to the *move-item*.
4. MOVE TO transfers data from the *move-item* to the record buffer.
5. The MOVE statement does not affect the record buffer's size. If a MOVE statement partially fills a buffer, the rest of the buffer is unchanged. If there is more data in the variable list than in the buffer, BASIC–PLUS–2 signals the error "MOVE overflows buffer" (ERR=161).

6. Each MOVE statement to or from a channel transfers data starting at the beginning of the buffer. For example:

```
100 MOVE FROM #1%, I%, A$ = I%
```

   In this example, BASIC–PLUS–2 assigns the first value in the record buffer to *I%*; the value of *I%* is then used to determine the length of *A$*.

7. If a MOVE statement operates on an entire array, the following conditions apply:
   - BASIC–PLUS–2 transfers elements of row and column zero (in contrast to the MAT statements).
   - The storage size of the array elements and the size of the array determine the amount of data moved. A MOVE statement that transfers data from the buffer to a longword integer array transfers the first four bytes of data into the first element (for example, (0,0)), the next four bytes of data into element (0,1), and so on.

8. If the MOVE TO statement specifies an explicit string length, the following restrictions apply:
   - If the string is equal to or longer than the explicit string length, BASIC–PLUS–2 moves only the specified number of characters into the buffer.
   - If the string is shorter than the explicit string length, BASIC–PLUS–2 moves the entire string and pads it with spaces to the specified length.

9. BASIC–PLUS–2 does not check the validity of data during the MOVE operation.

Examples

1.
```
20  MOVE FROM #4%, RUNS%, HITS%, ERRORS%, RBI%, BAT_AVERAGE
```

2.
```
100 MOVE TO #9%, FILL$ = 10%, A$ = 10%, B$ = 30%, C$ = 2%
```

NAME . . . AS

The NAME . . . AS statement renames the specified file.

Format

NAME file-spec1 **AS** file-spec2

Syntax Rules

1. *File-spec1* and *file-spec2* must be string expressions.
2. There is no default file type in *file-spec1* or *file-spec2*. If the file to be renamed has a file type, *file-spec1* must include both the file name and the file type.
3. If you specify only a file name, BASIC–PLUS–2 searches for a file with no file type. If you do not specify a file type for *file-spec2*, BASIC–PLUS–2 names the file but does not assign a file type.
4. If you specify a directory name with *file-spec2*, the file will be placed in the specified directory. If you do not specify a directory name, the default is the current directory.
5. On RSX systems, file version numbers are optional. BASIC–PLUS–2 renames the highest version of *file-spec1* if you do not specify a version number.

Remarks

1. If the file specified by *file-spec1* does not exist, BASIC–PLUS–2 signals the error "Can't find file or account" (ERR=5).
2. If you use the NAME . . . AS statement on an open file, BASIC–PLUS–2 does not rename the file until it is closed.
3. You cannot use the NAME . . . AS statement to move a file between devices. Under RSTS/E, you can change only the name, type, or version number. Under RSX, you can can also change the directory. If you specify a new device with the NAME . . . AS statement, BASIC–PLUS–2 signals the error "Illegal usage for device" (ERR=133).

Example

```
400  NAME "OLDPRG.BAS" AS "NEWPRG.BAS"
```

NEXT

The NEXT statement marks the end of a FOR, UNTIL, or WHILE loop.

Format

NEXT [num-unsubs-var]

Syntax Rules

1. *Num-unsubs-var* is required in a FOR . . . NEXT loop and must correspond to the *num-unsubs-var* specified in the FOR statement.
2. *Num-unsubs-var* is not allowed in an UNTIL or WHILE loop.
3. *Num-unsubs-var* must be a numeric, unsubscripted variable.

Remarks

Each NEXT statement must have a corresponding FOR, UNTIL, or WHILE statement, or BASIC–PLUS–2 signals an error.

Example

```
10  PROGRAM calculating_pay
    DECLARE INTEGER no_hours, &
            SINGLE weekly_pay, minimum_wage
    minimum_wage = 3.65
    no_hours = 40
    WHILE no_hours > 0
      INPUT "Enter the number of hours you intend to work this week";no_hours
      weekly_pay = no_hours * minimum_wage
      PRINT "If you worked";no_hours;"hours, your pay would be";weekly_pay
    NEXT
    END PROGRAM
```

The output is:

```
Enter the number of hours you intend to work this week? 35
If you worked 35 hours, your pay would be 127.75
Enter the number of hours you intend to work this week? 23
If you worked 23 hours, your pay would be 83.95
Enter the number of hours you intend to work this week? 0
If you worked 0 hours your pay would be 0
```

NOECHO

The NOECHO function disables echoing of input on a terminal.

Format

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

Syntax Rules

Chnl-exp is a numeric expression that specifies a channel number associated with an open terminal. It cannot be preceded by a number sign (#).

Remarks

1. If you specify NOECHO, BASIC–PLUS–2 accepts characters entered on the terminal as input, but the characters do not echo on the terminal.
2. The NOECHO function is the complement of the ECHO function; NOECHO disables the effect of ECHO and vice versa.
3. NOECHO always returns a value of zero.

Example

```
10  DECLARE INTEGER Y,      &
            STRING pass_word
    Y = NOECHO(0)
    INPUT "Enter your password";pass_word
    IF pass_word = "DARLENE" THEN PRINT "Confirmed"
    Y = ECHO(0)
```

The output is:

```
Enter your password?
Confirmed
```

NUM

The NUM function returns the row number of the last data element transferred into an array by a MAT I/O statement.

Format

int-var = **NUM**

Syntax Rules

None.

Remarks

1. NUM returns a value of zero if it is invoked before BASIC–PLUS–2 has executed any MAT I/O statements.
2. For a two-dimensional array, NUM returns an integer specifying the row number of the last data element transferred into the array. For a one-dimensional array, NUM returns the number of elements entered.
3. The value returned by the NUM function is an integer of the default size.

Example

```
10  OPEN "ACCT" FOR INPUT AS #2
    DIM stu_rec$(3,3)
    MAT INPUT #2, stu_rec$
    PRINT "Row count =";NUM
    PRINT "Column number =";NUM2
```

The output is:

```
Row count = 1
Column number = 1
```

NUM2

The NUM2 function returns the column number of the last data element transferred into an array by a MAT I/O statement.

Format

int-var = **NUM2**

Syntax Rules

None.

Remarks

1. NUM2 returns a value of zero if it is invoked before BASIC–PLUS–2 has executed any MAT I/O statements or if the last array element transferred was in a one-dimensional list.
2. The NUM2 function returns an integer specifying the column number of the last data element transferred into an array.
3. The value returned by the NUM2 function is an integer of the default size.

Example

```
10  OPEN "ACCT" FOR INPUT AS #2
    DIM stu_rec$(3,3)
    MAT INPUT #2, stu_rec$
    PRINT "Row count =";NUM
    PRINT "Column number =";NUM2
```

The output is:

```
Row count = 1
Column number = 1
```

NUM$

The NUM$ function evaluates a numeric expression and returns a string of characters in PRINT statement format, with leading and trailing spaces.

Format

str-var = **NUM$**(num-exp)

Syntax Rules

None.

Remarks

1. If *num-exp* is positive, the first character in the string expression is a space. If *num-exp* is negative, the first character is a minus sign (–).
2. The NUM$ function does not include trailing zeros in the returned string. If all digits to the right of the decimal point are zeros, NUM$ omits the decimal point as well.
3. When *num-exp* is a floating-point variable and has an integer portion of six decimal digits or less (for example, 1234.567), BASIC–PLUS–2 rounds the number to six digits (1234.57). If *num-exp* has seven decimal digits or more, BASIC–PLUS–2 rounds the number to six digits and prints it in E-format.
4. When *num-exp* is between 0.1 and 1 and contains more than 6 digits, BASIC–PLUS–2 rounds it to six digits. When *num-exp* is smaller than 0.1, BASIC–PLUS–2 rounds it to 6 digits and prints it in E-format.
5. The NUM$ function returns a maximum number of digits as follows:
   - Three digits for BYTE integers
   - Five digits for SINGLE floating-point numbers and WORD integers
   - Ten digits for LONG integers
   - Sixteen digits for DOUBLE floating-point numbers
6. The last character in the returned string is a space.

Example

```
10  DECLARE STRING number
    number = NUM$(34.5500/31.8)
    PRINT number
```

The output is:

```
1.08648
```

NUM1$

The NUM1$ function changes a numeric expression to a numeric character string without leading and trailing spaces and without rounding.

Format

str-var = **NUM1$**(num-exp)

Syntax Rules

None.

Remarks

1. The NUM1$ function returns a string consisting of numeric characters and a decimal point that corresponds to the value of *num-exp*. Leading and trailing spaces or zeroes are not included in the returned string. If all digits to the right of the decimal point are zeroes, the decimal point is also omitted.
2. If *num-exp* is negative, the first character returned by NUM1$ is a minus (-) sign.
3. Except for the special cases cited in remarks 1 and 2, the NUM1$ function returns a maximum number of digits as follows:
   - Three digits for BYTE integers
   - Six digits for SINGLE floating-point numbers and WORD integers
   - Ten digits for LONG integers
   - Sixteen digits for DOUBLE floating-point numbers
4. Disregarding leading and trailing zeroes, when SINGLE floating-point numbers contain more than six digits in the integer portion (such as 1234567.8), BASIC–PLUS–2 rounds the number to six digits, placing zeroes in the remaining integer portion (such as 1234570). When SINGLE floating-point numbers contain more than six digits, with six digits or less in the integer portion (such as 123456.7), BASIC–PLUS–2 rounds the number to six digits (such as 123457).

5. Rules for rounding DOUBLE floating-point numbers are the same as for SINGLE floating-point numbers, except the number of significant digits returned is 16, not six.
6. The NUM1$ function does not produce E notation.

Example

```
10  DECLARE STRING number
    number = NUM1$(PI/2)
    PRINT number
```

The output is:

```
1.5708
```

ON ERROR GO BACK

If the ON ERROR GO BACK statement is located in a subprogram or DEF function, it transfers control to the calling program when an error occurs. If the ON ERROR GO BACK statement is located in a main program module, it transfers control to the BASIC–PLUS–2 default error handler when an error occurs.

Format

{ **ONERROR** } **GO BACK**
{ **ON ERROR** }

Syntax Rules

None.

Remarks

1. If there is no error outstanding, execution of an ON ERROR GO BACK statement causes subsequent errors to return control to the calling program's error handler.
2. If there is an error outstanding, execution of an ON ERROR GO BACK statement immediately transfers control to the calling program's error handler.
3. By default, DEF functions and subprograms re-signal errors to the calling program.
4. The ON ERROR GO BACK statement remains in effect until the program unit completes execution or until BASIC–PLUS–2 executes another ON ERROR statement.
5. An ON ERROR GO BACK statement executed in the main program is equivalent to an ON ERROR GOTO 0 statement.
6. If a main program calls a subprogram named SUB1, and SUB1 calls the subprogram named SUB2, an ON ERROR GO BACK statement executed in SUB2 transfers control to SUB1's error handler when an error occurs in SUB2. If SUB1 also has executed an ON ERROR GO BACK statement, BASIC–PLUS–2 transfers control to the main program's error handling routine.

Example

```
10  IF ERR = 11
       THEN
            RESUME err_hand
       ELSE
            ON ERROR GO BACK
    END IF
```

ON ERROR GOTO

The ON ERROR GOTO statement transfers program control to a specified line or label in the current program unit when a trappable error occurs.

Format

{ **ONERROR** } { **GO TO** } target
{ **ON ERROR** } { **GOTO** }

Syntax Rules

1. *Target* must be a valid BASIC–PLUS–2 line number or label and must exist in the same program unit as the ON ERROR GOTO statement.
2. If an ON ERROR GOTO statement is in a DEF function, *target* must also be in that function definition.

Remarks

1. Execution of an ON ERROR GOTO statement causes subsequent errors to transfer control to the specified target.
2. The ON ERROR GOTO statement remains in effect until the program unit completes execution or until BASIC–PLUS–2 executes another ON ERROR statement.
3. BASIC–PLUS–2 does not allow recursive error handling. If a second error occurs during execution of an error-handling routine, control passes to the BASIC–PLUS–2 error handler and the program stops executing.

Example

```
50  SUB LIST (STRING A)
    DECLARE STRING B
    ON ERROR GOTO err_block
    OPEN A FOR INPUT AS FILE #1

    Input_loop:
        LINPUT #1, B
        PRINT B
    .
    .
    .
    GOTO Input_loop
```

```
err_block:
     IF (ERR=11%)
       THEN
          CLOSE #1%
          RESUME done
       ELSE
          ON ERROR GOTO 0
     END IF

done:
END SUB
```

ON ERROR GOTO 0

The ON ERROR GOTO 0 statement disables ON ERROR error handling and passes control to the BASIC–PLUS–2 error handler when an error occurs.

Format

{ **ON ERROR** } { **GO TO** } 0
{ **ONERROR** } { **GOTO** }

Syntax Rules

None.

Remarks

1. If an error is outstanding, execution of an ON ERROR GOTO 0 statement immediately transfers control to the BASIC–PLUS–2 error handler. The BASIC–PLUS–2 error handler will report the error and exit the program.
2. If there is no error outstanding, execution of an ON ERROR GOTO 0 statement causes subsequent errors to transfer control to the BASIC–PLUS–2 error handler.

Example

```
10  ON ERROR GOTO err_routine
    FOR I = 1% TO 10%
        PRINT "Please type a number"
        INPUT A
    NEXT I
    err_routine:
    IF ERR = 50
       THEN
          RESUME
       ELSE
          ON ERROR GOTO 0
    END IF
```

The output is:

```
Please type a number
? Ctrl/Z
?End of file on device at line   10 in "MYPROG  "
```

ON . . . GOSUB

The ON . . . GOSUB statement transfers program control to one of several subroutines, depending on the value of a control expression.

Format

ON int-exp **GOSUB** target , . . . [**OTHERWISE** target]

Syntax Rules

1. *Int-exp* determines which target BASIC–PLUS–2 selects as the GOSUB argument. If *int-exp* equals 1, BASIC–PLUS–2 selects the first target. If *int-exp* equals 2, BASIC–PLUS–2 selects the second target, and so on.
2. *Target* must be a valid BASIC–PLUS–2 line number or label and must exist in the current program unit.

Remarks

1. Control cannot be transferred into a statement block (such as FOR . . . NEXT, UNTIL . . . NEXT, WHILE . . . NEXT, DEF . . . END DEF, or SELECT . . . END SELECT).
2. If there is an OTHERWISE clause, and if *int-exp* is less than 1 or greater than the number of targets in the list, BASIC–PLUS–2 selects the target of the OTHERWISE clause.
3. If there is no OTHERWISE clause, and if *int-exp* is less than 1 or greater than the number of targets in the list, BASIC–PLUS–2 signals the error "ON statement out of range" (ERR=58).
4. If a target specifies a nonexecutable statement, BASIC–PLUS–2 transfers control to the first executable statement that lexically follows the target.

Example

```
100     INPUT "Please enter 1, 2 or 3"; A%
        ON A% GOSUB 1000, 2000, 3000 OTHERWISE err_routine
        GOTO done

1000    PRINT "That was a 1"
        RETURN
2000    PRINT "That was a 2"
        RETURN
3000    PRINT "That was a 3"
        RETURN

    err_routine:
        PRINT "Out of range:
        RETURN
    done:
        END
```

ON . . . GOTO

The ON . . . GOTO statement transfers program control to one of several lines or targets, depending on the value of a control expression.

Format

ON int-exp { **GO TO** / **GOTO** } target , . . . [**OTHERWISE** target]

Syntax Rules

1. *Int-exp* determines which target BASIC–PLUS–2 selects as the GOTO argument. If *int-exp* equals 1, BASIC–PLUS–2 selects the first target. If *int-exp* equals 2, BASIC–PLUS–2 selects the second target, and so on.
2. *Target* must be a valid BASIC–PLUS–2 line number or a label and must exist in the current program unit.

Remarks

1. Control cannot be transferred into a statement block (such as FOR . . . NEXT, UNTIL . . . NEXT, WHILE . . . NEXT, DEF . . . END DEF, or SELECT . . . END SELECT).
2. If there is an OTHERWISE clause, and if *int-exp* is less than 1 or greater than the number of targets in the list, BASIC–PLUS–2 transfers control to the target of the OTHERWISE clause.
3. If there is no OTHERWISE clause, and if *int-exp* is less than 1 or greater than the number of line numbers in the list, BASIC–PLUS–2 signals "ON statement out of range" (ERR=58).
4. If a target specifies a nonexecutable statement, BASIC–PLUS–2 transfers control to the first executable statement that lexically follows the target.

Example

```
10  ON INDEX% GOTO 700,800,900 OTHERWISE 1000
```

ONECHR

The ONECHR function allows single-character input (ODT submode) to a terminal opened on a specified channel. This function must be used in conjunction with the GET statement.

Format

int-vbl = **ONECHR**(chnl-exp)

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with an open terminal. It cannot be preceded by a number sign (#).
2. The ONECHR function must be used immediately before a GET statement.

Remarks

1. BASIC disables the ONECHR function immediately after a GET statement executes. Therefore, your program must invoke the ONECHR function for each single character input you want to perform.
2. Control passes to the program as soon as you enter a character. You do not have to enter a line terminator.
3. To obtain optimal performance (especially when using scrolling regions and other compilicated operations), it is necessary that you attach to the terminal before using the ONECHR function. To attach to the terminal use an IO.ATT QIO. See your operating system documentation for additional information about QIOs.

Example

```
100     OPEN "TI:" FOR INPUT AS FILE #1%
        Y% = ONECHR(1%)
        GET #1%
        MOVE FROM #1%, A$ = 1%
        PRINT A$
```

OPEN

The OPEN statement opens a file for processing. It transfers user-specified file characteristics to Record Management Services (RMS), or to the operating system on RSTS/E systems, and verifies the results.

Format

OPEN file-spec [**FOR INPUT** | **FOR OUTPUT**] **AS [FILE]** [#] chnl-exp
[, open-clause] . . .

open-clause:

[**ACCESS** { **APPEND** | **READ** | **WRITE** | **MODIFY** | **SCRATCH** }]

[**ALLOW** { **NONE** | **READ** | **WRITE** | **MODIFY** }]

[**BUFFER** *int-exp*]

[**CLUSTERSIZE** *int-exp*]

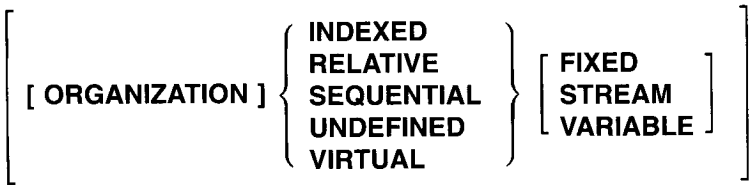
[**CONTIGUOUS**]

[**DEFAULTNAME** *file-spec*]

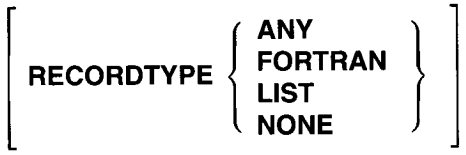
[**EXTENDSIZE** *int-exp*]

[**FILESIZE** *int-exp*]

[**MAP** *map-name*]



[**RECORDSIZE** *int-exp*]



[**TEMPORARY**]

[**USEROPEN** *func-name*]

[**WINDOWSIZE** *int-exp*]

Indexed files only:

[**ALTERNATE [KEY]** *key-clause* **[DUPLICATES] [CHANGES]**]

[**PRIMARY [KEY]** *key-clause* **[DUPLICATES]**]

Relative and indexed files only:

[**BUCKETSIZE** *int-exp*]

[**CONNECT** *chnl-exp2*]

Sequential files only:

[**BLOCKSIZE** *int-exp*]

[**NOREWIND**]

[**[NO]SPAN**]

Virtual files only:

[**MODE** *int-exp*]

key-clause: { *int-unsubs-var* | *decimal-unsubs-var* | *str-unsubs-var* | *(str-unsubs-var1 , . . . str-unsubs-var8)* }

Syntax Rules

1. *File-spec* specifies the file to be opened and associated with *chnl-exp*. It can be any valid string expression and must be a valid file specification. BASIC–PLUS–2 passes these values to RMS without editing, alteration, or validity checks.
2. BASIC–PLUS–2 supplies SY: as the default device if you do not specify a device in your file specification. No default file type is supplied unless you include the DEFAULTNAME clause in the OPEN statement.
3. The FOR clause determines how BASIC–PLUS–2 opens a file.
   - If you open a file with FOR INPUT, the file must exist or BASIC–PLUS–2 signals an error.
   - If you open a file with FOR OUTPUT, BASIC–PLUS–2 creates the file if it does not exist. On RSX systems, if the file exists, BASIC–PLUS–2 creates a new version of the file; On RSTS/E systems, BASIC–PLUS–2 overwrites the file.
   - If you do not specify either FOR INPUT or FOR OUTPUT, BASIC–PLUS–2 tries to open an existing file. If there is no such file, BASIC–PLUS–2 creates one.
4. *Chnl-exp* is a numeric expression that specifies a channel number to be associated with the file being opened. It can be preceded by an optional number sign (#) and must be in the range of 1 through 12.
5. A statement that accesses a file cannot execute until you open that file and associate it with a channel.

Remarks

1. The OPEN statement does not retrieve records.
2. Channel #0, the terminal, is always open. If you try to open channel zero, BASIC–PLUS–2 signals the error "I/O channel already open" (ERR=7).
3. If a program opens a file on a channel already associated with an open file, BASIC–PLUS–2 closes the previously opened file and opens the new one.

4. The implicit or explicit closing of a file will render any channels connected to that file invalid.
5. The ACCESS clause determines how the program can use the file.
   - ACCESS READ allows only FIND, GET, or other input statements on the file. The OPEN statement cannot create a file if the ACCESS READ clause is specified.
   - ACCESS WRITE allows only PUT, UPDATE, or other output statements on the file.
   - ACCESS MODIFY allows any I/O statement except SCRATCH on the file. ACCESS MODIFY is the default.
   - ACCESS SCRATCH allows any I/O statement valid for a sequential or terminal-format file.
   - ACCESS APPEND is the same as ACCESS WRITE for sequential files, except that BASIC–PLUS–2 positions the file pointer after the last record when it opens the file. You cannot use ACCESS APPEND on relative or indexed files.
6. The ALLOW clause can be used in the OPEN statement to specify file sharing of relative, indexed, sequential, and virtual files.
   - ALLOW NONE lets no other users access the file. This is the default when any access other than READ is specified.
   - ALLOW READ lets other users have read access to the file.
   - ALLOW WRITE lets other users have write access to the file.
   - ALLOW MODIFY lets other users have unlimited access to the file.
7. The BLOCKSIZE clause specifies the physical block size of magnetic tape files. The BLOCKSIZE clause can be used for magnetic tape files only.
   - The value of *int-exp* is the number of records in a block. Therefore, the block size in bytes is the product of the RECORDSIZE and the BLOCKSIZE value.
   - The default blocksize is one record.
8. The BUCKETSIZE clause applies only to relative and indexed files. It specifies the size of an RMS bucket in terms of the number of records one bucket should hold.
   - The value of *int-exp* determines the number of records in a bucket. Therefore, the BUCKETSIZE in bytes is the product of the record size and bucket size.

- The default is one record.

9. The BUFFER clause can be used with all file organizations except UNDEFINED.
   - For RELATIVE and INDEXED files, *int-exp* specifies the number of device or file buffers RMS uses for file processing.
   - For SEQUENTIAL files, *int-exp* specifies the size of the buffer; for example, BUFFER 8 for a SEQUENTIAL file sets the buffer size to eight 512-byte blocks.
   - For VIRTUAL files, the BUFFER clause has no effect.
10. The CLUSTERSIZE clause allows you to specify the smallest amount of contiguous disk space to be allocated when an RMS or RSTS/E file's present allocation is exhausted.
   - The CLUSTERSIZE clause is valid on RSTS/E systems only. On RSX systems, the EXTENDSIZE clause serves a function similar to that of the CLUSTERSIZE clause.
   - *Int-exp* must be a power of 2. For example, a CLUSTERSIZE of 8 means that each time the file requires more disk space, the RSTS/E operating system must have at least eight contiguous blocks to allocate. If the disk is fragmented, therefore, there can be no 8-block clusters and BASIC–PLUS–2 signals the error "No room for user on device" (ERR=4).
   - The default CLUSTERSIZE is determined when the disk pack is initialized or mounted. You can use the DCL command SHOW DISK to determine the default clustersize.
11. The CONTIGUOUS clause causes RMS to try to create the file as a contiguous sequence of disk blocks.
   - The CONTIGUOUS clause does not affect existing files or nondisk files.
   - If you specify a CONTIGUOUS clause and there is not enough contiguous disk space available, RMS signals an error.
   - On RSTS/E systems, you cannot extend a contiguous file beyond its initially allocated size. If you attempt to do so, BASIC–PLUS–2 signals the error "Protection violation" (ERR=10). Use the FILESIZE clause with the CONTIGUOUS clause to allocate enough contiguous disk space when you open the file.

12. The CONNECT clause establishes additional record access streams for RMS files that allow your program to process more than one record of a file at the same time. Each stream represents an independent and concurrently active sequence of record operations.

    - The CONNECT clause must specify a RELATIVE or INDEXED file already opened on *chnl-exp2* with the primary OPEN statement.
    - Each connection established in a secondary OPEN statement uses another I/O channel. Because there are 12 available I/O channels, you can have a maximum of 11 connections to a file.
    - All clauses in the secondary OPEN statements must be identical except the MAP, CONNECT, and USEROPEN clauses.
    - BASIC–PLUS–2 signals the error "Invalid file options" (ERR=139) if your program attempts to connect to a record stream that is already connected to another stream.
    - When a file is closed, all files that are connected to that file must also be closed.

13. The DEFAULTNAME clause lets you supply a default file specification.

    - The DEFAULTNAME clause is valid for RMS files only.
    - If *file-spec* is not a complete file specification, *file-spec2* specified in the DEFAULTNAME clause supplies the missing parts. For example:

```
10      INPUT 'FILE NAME';fnam$
20      OPEN fnam$ FOR INPUT AS FILE #1%,     &
                   ORGANIZATION SEQUENTIAL &
                   DEFAULTNAME "DU1:.DAT"
```

    Here, if you enter "ABC" for the file name, BASIC–PLUS–2 tries to open DU1:[123,2]ABC.DAT.

14. The EXTENDSIZE clause lets you specify the increment by which RMS extends a file after its initial allocation is filled.

    - The EXTENDSIZE clause is valid on RSX systems only. On RSTS/E systems, the CLUSTERSIZE clause serves a similar function to the EXTENDSIZE clause.
    - The value of *int-exp* is in 512-byte disk blocks.
    - The EXTENDSIZE clause has no effect on an existing file.

15. The FILESIZE clause lets you pre-extend a new file to a specified size.

    - The value of *int-exp* is the initial allocation of disk blocks.

- The FILESIZE clause has no effect on an existing file.

16. The MAP clause specifies that a previously declared map is associated with the file's record buffer. The MAP clause determines the record buffer's address and length unless overridden by the RECORDSIZE clause.
    - The size of the specified map must be as large or larger than the longest record length or maximum record size. For files with a fixed record size, the specified map must match exactly.
    - The size of the largest MAP with the same map name in the current program unit becomes the file's record size if the OPEN statement does not include a RECORDSIZE clause.
    - It is recommended that you do not use both the MAP and RECORDSIZE clauses in an OPEN statement. If you do use both the MAP and RECORDSIZE clauses in an OPEN statement, the following rules apply:
        - The RECORDSIZE clause overrides the record size set by the MAP clause.
        - The map must be as large or larger than the specified RECORDSIZE.
    - If there is no MAP clause, the record buffer space that BASIC–PLUS–2 allocates is not directly accessible. Therefore, MOVE statements are needed to access data in the record buffer.
    - You must have a MAP clause when creating an indexed file; you cannot use KEY clauses without MAP statements because keys serve as offsets into the buffer.
17. The MODE clause is provided for non-RMS file operations.
    - *Int-exp* specifies a MODE value.
    - On RSX systems, MODE is ignored except when your program is doing device-specific I/O to a magnetic tape. In this case, you can use MODE to set the tape density. In all other cases, RSX ignores the MODE value. See the *BASIC–PLUS–2 User's Guide* for information on MODE values.
    - On RSTS/E systems, MODE values affect only native RSTS/E files, not RMS files. MODE values have different meanings depending on the context in which you use them. This is because other pieces of software can scan the MODE values to see which bits are set. For example, bit 14 may have one meaning to the RSTS/E terminal driver but another

meaning to the file processor. See the *BASIC–PLUS–2 User's Guide* for information on MODE values.

18. The NOREWIND clause controls tape positioning on magnetic tape files so that the operating system does not position the tape at its beginning. Your program can then search for records from the current position.

    - When you specify NOREWIND with an OPEN FOR INPUT statement, BASIC–PLUS–2 instructs the system to search for the specified file without rewinding. If the file is not found, BASIC–PLUS–2 instructs the system to rewind the tape and search for the file from the start of the tape. If the file is still not found, BASIC–PLUS–2 signals the error "File not found."
    - If you specify NOREWIND with an OPEN FOR OUTPUT statement, BASIC–PLUS–2 instructs the system to position the tape at its logical end. The program can then write records.
    - By default, if you do not specify NOREWIND, the tape is positioned at its beginning.
    - The NOREWIND clause can be used for magnetic tape files only.

19. The NOSPAN clause specifies that records cannot cross block boundaries. The NOSPAN clause does not affect nondisk files. The SPAN clause specifies that sequential records can cross block boundaries. SPAN is the default.

20. The ORGANIZATION clause specifies the organization of a file. When present, it must precede all other clauses. When you specify an ORGANIZATION clause, you must also specify one of the following organization options: UNDEFINED, INDEXED, SEQUENTIAL, RELATIVE, or VIRTUAL.

    - Specify ORGANIZATION UNDEFINED if you do not know the actual organization of the file. When you specify the ORGANIZATION UNDEFINED clause, a block I/O file is opened. You can then use the FSP$ function or a USEROPEN routine to determine the attributes of the file. You will usually want to specify the RECORDTYPE ANY clause with the ORGANIZATION UNDEFINED clause. The combination of these two clauses should allow you to access any file sequentially. You must have exclusive access to a file in order to specify the ORGANIZATION UNDEFINED clause in an OPEN statement. Therefore, you must also specify the ALLOW NONE clause when you specify ORGANIZATION UNDEFINED.

- When you specify ORGANIZATION INDEXED, you create an indexed file that contains data records that are sorted in ascending or descending order according to a *primary index key value*. You can access an existing file with descending keys; however, you cannot create a file with descending keys.
  - The index keys you specify determine the order in which records are stored.
  - Index keys must be variables declared in a MAP statement associated with the OPEN statement for the file.
  - BASIC–PLUS–2 allows you to specify an indexed file as either variable or fixed length.
- When you specify ORGANIZATION SEQUENTIAL, you create a file that stores records in the order in which they are written.
  - Sequential files can contain records of any valid BASIC–PLUS–2 record format: fixed-length, variable-length, or stream. If you do not specify a record format with the ORGANIZATION SEQUENTIAL clause, the default is variable-length records.
  - If you open an existing file using stream as a record format option, the STREAM records can be delimited by any special character.
- When you specify ORGANIZATION RELATIVE, you create a file that contains a series of records that are numbered consecutively. BASIC–PLUS–2 allows you to specify either fixed-length or variable-length records.
- When you specify ORGANIZATION VIRTUAL, you create a sequentially fixed file with a record size of 512 (or a multiple of 512). You can then access the file with the FIND, GET, PUT, or UPDATE statements or through one or more virtual arrays. BASIC–PLUS–2 allows you to overwrite existing records in a file not containing virtual arrays and opened as ORGANIZATION VIRTUAL by using the PUT statement with a RECORD clause. All other organizations require the UPDATE statement to change an existing record. It is recommended that you also use the UPDATE statement to change existing records in VIRTUAL files that do not contain virtual arrays.
- If you omit the ORGANIZATION clause, BASIC–PLUS–2 opens a terminal-format file.
  - Terminal-format files are implemented as RMS sequential variable files on RSX systems and store ASCII characters in variable-length records.

- Carriage control is performed by the operating system; and on RSX systems, the record does not contain carriage returns or line feeds. RSTS/E terminal-format files, however, do contain carriage return and line feed characters.
- You use essentially the same syntax to access terminal-format files as when reading from or writing to the terminal (INPUT and PRINT).

21. The PRIMARY KEY clause lets you specify an indexed file's key. You must specify a primary key when opening an indexed file. The ALTERNATE KEY clause lets you specify up to 254 alternate keys. The ALTERNATE KEY clause is optional.

    - RMS creates one index list for each primary and alternate key you specify. These indexes are part of the file and contain pointers to the records. Each key you specify corresponds to a sorted list of record pointers.
    - The keys you specify determine the order in which records in the file are stored. All keys must be variables declared in the file's corresponding MAP statement. The position of the key in the MAP statement determines its position in the record. The data type and size of the key are as declared in the MAP statement.
    - A key can be an unsubscripted string or WORD variable.
    - You can also create a segmented index key for string keys by separating the string variable names with commas and enclosing them in parentheses. You can then reference a segment of the specified key by referencing one of the string variables instead of the entire key. A string key can have up to eight segments.
    - The order of appearance of keys determines key numbers. The primary key, which must appear first, is key #0. The first alternate key is #1, and so on.
    - DUPLICATES in the PRIMARY and ALTERNATE key clauses specifies that two or more records can have the same key value. If you do not specify DUPLICATES, the key value must be unique in all records.
    - CHANGES in the ALTERNATE KEY clause specifies that you can change the value of an alternate key when updating records. If you do not specify CHANGES when creating the file, you cannot change the value of a key. If you specify the CHANGES clause, you must also specify the duplicates clause. You cannot specify CHANGES with the PRIMARY KEY clause.

- ALTERNATE KEY clauses are optional for existing files. If you do specify a key, it must match a key in the file.

22. The RECORDTYPE clause specifies the file's record attributes. The RECORDTYPE clause can only be used with RMS files.
    - ANY specifies a match with any file attributes when opening an existing file. If you create a new file, ANY is treated as LIST for all organizations except VIRTUAL. For VIRTUAL, it is treated as NONE.
    - FORTRAN specifies a control character in the record's first byte.
    - LIST specifies implied carriage control, CR LF. This is the default for all file organizations except VIRTUAL.
    - NONE specifies no attributes. This is the default for VIRTUAL files.
23. The RECORDSIZE clause specifies the file's record size. Note that there are restrictions on the maximum record size allowed for various file and record formats. See the RMS–11 documentation for more information.
    - For fixed-length records, *int-exp* specifies the size of all records.
    - For variable-length records, *int-exp* specifies the size of the largest record.
    - It is recommended that you do not use both the MAP and RECORDSIZE clauses in an OPEN statement. If you do use both the MAP and RECORDSIZE clauses in an OPEN statement, the following rules apply:
      - The RECORDSIZE clause overrides the record size set by the MAP clause.
      - The map must be as large or larger than the specified RECORDSIZE.
    - If you specify a MAP clause but no RECORDSIZE clause, the record size is equal to the map size.
    - If there is no MAP clause, the RECORDSIZE clause determines the record size.
    - When creating a relative or indexed file, you must specify either a MAP or RECORDSIZE clause; otherwise, BASIC–PLUS–2 signals an error.
    - For fixed files, the record size must match exactly.
    - If you do not specify a RECORDSIZE clause when opening an existing file, BASIC–PLUS–2 retrieves the record size value from the file.

- When you create SEQUENTIAL files, BASIC–PLUS–2 supplies a default record size of 132.
- The record size is always 512 for VIRTUAL files unless you specify a RECORDSIZE.

24. The TEMPORARY clause causes BASIC–PLUS–2 to delete the output file as soon as the program closes it.
25. The USEROPEN clause lets you open a file with your own FUNCTION subprogram.
    - *Func-name* is the name of a MACRO program; it cannot be the name of a BASIC–PLUS–2 program.
    - You do not need to declare the useropen routine as an external function.
    - BASIC–PLUS–2 calls the user program after it fills the File Access Block (FAB), the Record Access Block (RAB), and the Extended Attribute Blocks (XAB) . The subprogram must issue the appropriate RMS calls, including $OPEN and $CONNECT, and return the RMS status as the value of the function. See the *BASIC–PLUS–2 User's Guide* for more information on the USEROPEN routine.
26. The WINDOWSIZE clause followed by *int-exp* lets you specify the number of block retrieval pointers you want to maintain in memory for the file. Retrieval pointers are associated with the file header and point to contiguous blocks on disk. By keeping retrieval pointers in memory, you can reduce the I/O associated with locating a record, as the operating system does not have to access the file header for pointers as frequently.
    - The number of retrieval pointers in memory at any one time is determined by the system default or by the WINDOWSIZE clause.
    - On RSX systems, you can specify up to 127 retrieval pointers. The default number of retrieval pointers is seven.
    - On RSTS/E systems, the number of pointers in a window block is fixed at seven. Thus, you cannot use the WINDOWSIZE clause. You can, however, use the CLUSTERSIZE clause to increase the number of contiguous blocks mapped by one retrieval pointer.

Examples

1.
```
10  OPEN "FILE.DAT" AS FILE #4
```

2.
```
10  OPEN "INPUT.DAT" FOR INPUT AS FILE #4,                &
             ORGANIZATION SEQUENTIAL FIXED,               &
             RECORDSIZE 200,                              &
             MAP ABC,                                     &
             ALLOW MODIFY, ACCESS MODIFY

    OPEN Newfile$ FOR OUTPUT AS FILE #3,                  &
             INDEXED VARIABLE,                            &
             MAP Emp_name,                                &
             DEFAULTNAME "SY:.DAT",                       &
             PRIMARY KEY Last$ DUPLICATES,                &
             ALTERNATE KEY First$ DUPLICATES, CHANGES

    MAP (SEGKEY) STRING last_name = 15,                   &
                 MI = 1, first_name = 15

    OPEN "NAMES.IND" FOR OUTPUT AS FILE #1,               &
             ORGANIZATION INDEXED,                        &
             PRIMARY KEY (last_name, first_name, MI),     &
             MAP SEGKEY
```

OPTION

The OPTION statement allows you to set compilation qualifiers such as default data type, size, and scale factor. You can also set compilation conditions such as constant type checking. The options you set affect only the program module in which the OPTION statement occurs.

Format

OPTION option-clause, . . .

option-clause: { **CONSTANT TYPE** =const-type-clause | **TYPE**=type-clause | **SIZE**=size-clause | **SCALE**=int-const }

const-type-clause: { **REAL** | **INTEGER** }

type-clause: { **INTEGER** | **REAL** | **EXPLICIT** }

size-clause: { size-item | (size-item, . . .) }

size-item: { **INTEGER** int-clause | **REAL** real-clause }

int-clause: { **BYTE** | **WORD** | **LONG** }

real-clause: { **SINGLE** | **DOUBLE** }

Syntax Rules

None.

Remarks

1. *Option-clause* specifies the compilation qualifiers to be in effect for the program module.
2. *Const-type-clause* specifies the data type for all constants that do not end in a data type suffix or are not in explicit literal notation with a data type supplied.
3. *Type-clause* sets the default data type for variables that have not been explicitly declared and for constants if no constant type clause is specified. You can specify only one *type-clause* in a program module.
4. *Size-clause* sets the default data subtypes for floating-point and integer data. *Size-item* specifies the data subtype you want to set. You can specify an INTEGER or REAL *size-item*, or any combination. Multiple *size-items* in an OPTION statement must be enclosed in parentheses and separated by commas.
5. The SCALE option controls the scaling of double-precision-floating-point variables. *Int-const* specifies the power of 10 you want as the scaling factor. It must be an integer between 0 and 6 or BASIC–PLUS–2 signals an error. See the description of the SCALE command in Chapter 2 of this manual for more information on scaling.
6. You can have more than one option in an OPTION statement, or you can use multiple OPTION statements in a program module; however, each OPTION statement must lexically precede all other source code in the program module, with the exception of comment fields, REM, PROGRAM, SUB, FUNCTION, and OPTION statements.
7. OPTION statement specifications apply only to the program module in which the statement appears and affect all variables in the module, including SUB and FUNCTION parameters.
8. BASIC–PLUS–2 signals an error in the case of conflicting options. For example, you cannot specify more than one *type-clause* or SCALE factor in the same program unit.
9. If you do not specify a *type-clause* or a *subtype-clause*, BASIC–PLUS–2 uses the current environment default data types.

10. If you do not specify a scale factor, BASIC–PLUS–2 uses the current environment default scale factor.

Example

```
FUNCTION REAL DOUBLE monthly_payment,                    &
        (DOUBLE interest_rate,                           &
          LONG   no_of_payments,                         &
          DOUBLE principle)
OPTION TYPE = REAL,                                      &
       SIZE = (REAL DOUBLE, INTEGER LONG),               &
       SCALE = 4
```

PLACE$

The PLACE$ function explicitly changes the precision of a numeric string. PLACE$ returns a numeric string, truncated or rounded, according to the value of an integer argument you supply.

Format

str-var = **PLACE$**(str-exp, int-exp)

Syntax Rules

1. *Str-exp* specifies the numeric string you want to process. It can have ASCII digits, an optional minus sign (–), and an optional decimal point (.).
2. *Int-exp* specifies the numeric precision of *str-exp*. Table 4–6 shows examples of rounding and truncation and the values of *int-exp* that produce them.

Remarks

1. If *str-exp* has more than 60 characters, BASIC–PLUS–2 signals the error "Illegal number" (ERR=52).
2. *Str-exp* is rounded or truncated, or both, according to the value of *int-exp*.
3. If *int-exp* is between –60 and 60, rounding and truncation occur as follows:
   - For positive integer expressions, rounding occurs to the right of the decimal place. For example, if *int-exp* is 1, rounding occurs one digit to the right of the decimal place (the number is rounded to the nearest tenth). If *int-exp* is 2, rounding occurs two digits to the right of the decimal place (the number is rounded to the nearest hundredth), and so on.
   - If *int-exp* is zero, BASIC–PLUS–2 rounds to the nearest unit.
   - For negative integer expressions, rounding occurs to the left of the decimal point. If *int-exp* is –1, for example, BASIC–PLUS–2 moves the decimal point one place to the left, then rounds to units. If *int-exp* is –2, rounding occurs two places to the left of the decimal point; BASIC–PLUS–2 moves the decimal point two places to the left, then rounds to tens.

4. If *int-exp* is between 9940 and 10,060, truncation occurs as follows:
   - If *int-exp* is 10,000, BASIC–PLUS–2 truncates the number at the decimal point.
   - If *int-exp* is greater than 10,000 (10,000 plus *n*) BASIC–PLUS–2 truncates the numeric string *n* places to the right of the decimal point. For example, if *int-exp* is 10,001 (10,000 plus 1), BASIC–PLUS–2 truncates the number starting one place to the right of the decimal point. If *int-exp* is 10,002 (10,000 plus 2), BASIC–PLUS–2 truncates the number starting two places to the right of the decimal point, and so on.
   - If *int-exp* is less than 10,000 (10,000 minus *n*), BASIC–PLUS–2 truncates the numeric string *n* places to the left of the decimal point. For example, if *int-exp* is 9999 (10,000 minus 1), BASIC–PLUS–2 truncates the number starting one place to the left of the decimal point. If 9998 (10,000 minus 2), BASIC–PLUS–2 truncates starting two places to the left of the decimal point, and so on.
5. If *int-exp* is not between –60 and 60 or 9940 and 10,060, BASIC–PLUS–2 returns a value of zero.
6. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to an integer of the default size.
7. Table 4–6 shows examples of rounding and truncation and the values of *int-exp* that produce them. The number used is 123456.654321.

Table 4–6 Rounding and Truncation of 123456.654321

Int-exp	Effect	Value Returned
–5	Rounded to 100,000s and truncated	1
–4	Rounded to 10,000s and truncated	12
–3	Rounded to 1000s and truncated	123
–2	Rounded to 100s and truncated	1235
–1	Rounded to 10s and truncated	12346
0	Rounded to units and truncated	123457
1	Rounded to tenths and truncated	123456.7
2	Rounded to hundredths and truncated	123456.65

(continued on next page)

Table 4–6 (Cont.) Rounding and Truncation of 123456.654321

Int-exp	Effect	Value Returned
3	Rounded to thousandths and truncated	123456.654
4	Rounded to ten-thousandths and truncated	123456.6543
5	Rounded to hundred-thousandths and truncated	123456.65432
9,995	Truncated to 100,000s	1
9,996	Truncated to 10,000s	12
9,997	Truncated to 1000s	123
9,998	Truncated to 100s	1234
9,999	Truncated to 10s	12345
10,000	Truncated to units	123456
10,001	Truncated to tenths	12345.6
10,002	Truncated to hundredths	123456.65
10,003	Truncated to thousandths	123456.654
10,004	Truncated to ten-thousandths	123456.6543
10,005	Truncated to hundred-thousandths	123456.65432

Example

```
10  DECLARE STRING str_exp, str_var
    str_exp = "9999.9999"
    str_var = PLACE$(str_exp,3)
    PRINT str_var
```

The output is:

```
10000
```

POS

The POS function searches for a substring within a string and returns the substring's starting character position.

The POS function is the same as the INSTR function except for the order of the arguments.

Format

int-var = **POS**(str-exp1, str-exp2, int-exp)

Syntax Rules

1. *Str-exp1* specifies the main string.
2. *Str-exp2* specifies the substring.
3. *Int-exp* specifies the character position in the main string at which BASIC–PLUS–2 starts the search.

Remarks

1. The POS function searches *str-exp1*, the main string, for the first occurrence of *str-exp2*, the substring, and returns the position of the substring's first character.
2. The position returned by the POS function is the number of characters from the beginning of the string regardless of the value specified in *int-exp*.
3. If *int-exp* is greater than the length of the main string, POS returns a value of zero.
4. POS always returns the character position in the main string at which BASIC–PLUS–2 finds the substring, with the following exceptions:
   - If only the substring is null, and if *int-exp* is less than or equal to zero, POS returns a value of 1.
   - If only the substring is null, and if *int-exp* is equal to or greater than 1 and less than or equal to the length of the main string, POS returns the value of *int-exp*.
   - If only the substring is null and if *int-exp* is greater than the length of the main string, POS returns the main string's length plus 1.
   - If only the main string is null, POS returns a value of zero.

- If both the main string and the substring are null, POS returns 1.

5. If BASIC–PLUS–2 cannot find the substring, POS returns a value of zero.
6. If *int-exp* is less than 1, BASIC–PLUS–2 assumes a starting position of 1.
7. If you know that the substring is not near the beginning of the string, specifying a starting position greater than 1 speeds program execution by reducing the number of characters BASIC–PLUS–2 must search.
8. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to an integer of the default size.

Example

```
400  DECLARE STRING main_str,   &
                    sub_str
     DECLARE INTEGER  first_char
     main_str = "ABCDEFG"
     sub_str = "DEFG"
     first_char = POS(main_str, sub_str, 1)
     PRINT first_char
```

The output is:

```
4
```

PRINT

The PRINT statement transfers program data to a terminal or a terminal-format file.

Format

PRINT [#chnl-exp,] [output-list]

output-list: [exp] [{ , ; } exp] . . . [, ;]

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#). If you do not specify a channel, BASIC–PLUS–2 prints to the controlling terminal.
2. *Output-list* specifies the expressions to be printed and the print format to be used.
3. *Exp* can be any valid expression.
4. A separator character (comma or semicolon) must separate each *exp*. Separator characters control the print format as follows:
   - A comma (,) causes BASIC–PLUS–2 to skip to the next print zone before printing the expression.
   - A semicolon (;) causes BASIC–PLUS–2 to print the expression immediately after the previous expression.

Remarks

1. A terminal or terminal-format file must be open on the specified channel. Your current terminal is always open on channel #0.
2. A PRINT line has an integral number of print zones. Note, however, that the number of print zones in a line differs from terminal to terminal.
3. The PRINT statement prints string constants and variables exactly as they appear, with no leading or trailing spaces.

4. BASIC–PLUS–2 prints quoted string literals exactly as they appear. Therefore, you can print quotation marks, commas, and other characters by enclosing them in quotation marks.
5. A PRINT statement with no *output-list* prints a blank line.
6. An expression in the *output-list* can be followed by more than one separator character. That is, you can omit an expression and specify where the next expression is to be printed by the use of multiple separator characters. For example:

```
200 PRINT "Name",,"Address and ";"City"
```

   The output is:

```
Name                          Address and City
```

   In this example, the double commas after "Name" cause BASIC–PLUS–2 to skip two print zones before printing "Address and." The semicolon causes the next expression, "City," to be printed immediately after the preceding expression. Multiple semicolons have the same effect as a single semicolon.
7. When printing numeric fields, BASIC–PLUS–2 precedes each number with a space or minus sign (–) and follows it with a space.
8. BASIC–PLUS–2 does not print trailing zeros to the right of the decimal point. If all digits to the right of the decimal point are zeros, BASIC–PLUS–2 omits the decimal point as well.
9. For REAL numbers (SINGLE and DOUBLE), BASIC–PLUS–2 does not print more than six digits in explicit notation. If a number requires more than six digits, BASIC–PLUS–2 uses E-format and precedes positive exponents with a plus sign (+). BASIC–PLUS–2 rounds a floating-point number with a magnitude between 0.1 and 1.0 to 6 digits. For magnitudes smaller than 1, BASIC–PLUS–2 prints a maximum of 5 leading zeros and 6 digits in explicit point unscaled notation.
10. The PRINT statement can print a maximum of:
    - Three digits of precision for BYTE integers
    - Five digits of precision for WORD integers
    - Six digits of precision for SINGLE floating-point numbers
    - Ten digits of precision for LONG integers
    - Sixteen digits of precision for DOUBLE floating-point numbers
    - The string length for STRING values

11. If there is a comma or semicolon following the last item in *output-list*, BASIC–PLUS–2 does the following:
    - When printing to a terminal, BASIC–PLUS–2 does not generate a line terminator after printing the last item. The next item printed with a PRINT statement is printed at the position specified by the separator character following the last item in the first PRINT statement.
    - When printing to a terminal-format file, BASIC–PLUS–2 does not write out the record until a PRINT statement without trailing punctuation executes.
    - If a comma is positioned after the last print zone, the output is displayed on the next line.
12. If no punctuation follows the last item in the *output-list*, BASIC–PLUS–2 does the following:
    - When printing to a terminal, BASIC–PLUS–2 generates a line terminator after printing the last item.
    - When printing to a terminal-format file, BASIC–PLUS–2 writes out the record after printing the last item.
13. If a string field does not fit on the current line, BASIC–PLUS–2 does the following:
    - When printing string elements to a terminal, BASIC–PLUS–2 prints as much as will fit on the current line and prints the remainder on the next line.
    - When printing string elements to a terminal-format file, BASIC–PLUS–2 prints the entire element on the next line.
14. If a numeric field is the first field in a line, and the numeric field spans more than one line, BASIC–PLUS–2 prints part of the number on one line and the remainder on the next; otherwise, numeric fields are never split across lines. If the entire field cannot be printed at the end of one line, the number is printed on the next line.
15. When a number's trailing space does not fit in the last print zone, the number is printed without the trailing space.

Example

```
10  PRINT "name "; "age", "height "; "weight"
```

The output is:

```
name age       height weight
```

PRINT USING

The PRINT USING statement generates output of numeric or STRING expressions, formatted according to a format string, and directs it to a terminal or a terminal-format file.

Format

PRINT [#chnl-exp] **USING** str-exp { , ; } output-list

output-list: [exp] [{ , ; } exp] . . . [{ , ; }]

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#). If you do not specify a channel, BASIC–PLUS–2 prints to the controlling terminal.
2. *Str-exp* is the format string. It must contain at least one valid format field and must be followed by a separator (comma or semicolon) and at least one expression.
3. *Output-list* specifies the expressions to be printed.
   - *Exp* can be any valid expression except a FORMAT$ function.
   - A comma or semicolon must separate each expression.
   - A comma or semicolon is optional after the last expression in the list.

Remarks

1. The PRINT USING statement can print a maximum of:
   - Three digits of precision for BYTE integers
   - Five digits of precision for WORD integers
   - Six digits of precision for SINGLE floating-point numbers
   - Ten digits of precision for LONG integers
   - Sixteen digits of precision for DOUBLE floating-point numbers

- The string length for STRING values

2. A terminal or terminal-format file must be open on the specified channel or BASIC–PLUS–2 signals an error.
3. The separator characters (comma or semicolon) in the PRINT USING statement do not control the print format. The print format is controlled by the format string. Therefore, it does not matter whether you use a comma or semicolon.
4. Formatting Numeric Output
   - The number sign (#) reserves space for one sign or digit.
   - The comma (,) causes BASIC–PLUS–2 to insert commas before every third significant digit to the left of the decimal point. In the format field, the comma must be to the left of the decimal point and to the right of the rightmost dollar sign, asterisk, or number sign. A comma reserves space for a comma or digit.
   - The period (.) inserts a decimal point. The number of reserved places on either side of the period determines where the decimal point appears in the output.
   - The hyphen (–) reserves space for a sign and specifies trailing minus sign format. If present, it must be the last character in the format field. It causes BASIC–PLUS–2 to print negative numbers with a minus sign after the last digit, and positive numbers with a trailing space. The hyphen (–) can be used as part of a dollar sign ($$) format field.
   - The letters CD (Credit/Debit) enclosed in angle brackets (<CD>) print CR (Credit Record) after negative numbers, or zero and DR (Debit Record) after positive numbers. If present, they must be the last characters in the format field. The Credit/Debit format can be used as part of a dollar sign ($$) format field.
   - Four carets (^^^^) specify E notation for floating-point numbers. They reserve four places for SINGLE and DOUBLE values. If present, they must be the last characters in the format field.
   - Two dollar signs ($$) reserve space for a dollar sign and a digit and cause BASIC–PLUS–2 to print a dollar sign immediately to the left of the most significant digit.
   - Two asterisks (**) reserve space for two digits and cause BASIC–PLUS–2 to fill the left side of the numeric field with leading asterisks.

- A zero enclosed in angle brackets (<0>) prints leading zeros instead of leading spaces.
- A percent sign enclosed in angle brackets (<%>) prints all spaces in the field if the value of the print item is zero.

Note

You cannot specify the dollar sign ($$), asterisk-fill (**), and zero-fill (<0>) formats within the same print field. Similarly, BASIC–PLUS–2 does not allow you to specify the zero-fill (<0>) and the blank-if-zero (<%>) formats within the same print field.

- An underscore (_) forces the next formatting character in the format string to be interpreted as a literal. It affects only the next character. If the next character is not a valid formatting character, the underscore has no effect.

5. BASIC–PLUS–2 interprets any other characters in a numeric format string as string literals.
6. Depending on usage, the same format string characters can be combined to form one or more print fields within a format string. For example:
   - When a dollar sign ($$) or asterisk-fill (**) format precedes a number sign (#) , it modifies the number sign format. The dollar sign or asterisk-fill format reserves two places, and with the number signs forms one print field. For example:

$$###	Forms one field and reserves five spaces
**##	Forms one field and reserves four spaces

     When these formats are not followed by a number sign or a blank-if-zero (<%>) format, they reserve two places and form a separate print field.
   - When a zero-fill (<0>) or blank-if-zero format precedes a number sign, it modifies the number sign format. The <0> or <%> reserves one place, and with the number signs forms one print field. For example:

<0>####	Forms one field and reserves five spaces
<%>###	Forms one field and reserves four spaces

     When these formats are not followed by a number sign, they reserve one space and form a separate print field.

- When a blank-if-zero (<%>) format follows a dollar sign or asterisk-fill format (**), it modifies the dollar sign ($$) or asterisk fill (**) format string. The blank-if-zero reserves one space, and with the dollar signs or asterisks forms one print field. For example:

  $$<%>### Forms one field and reserves six spaces

  **<%>## Forms one field and reserves five spaces

  When the blank-if-zero precedes the dollar signs or asterisks, it reserves one space and forms a separate print field.

7. For E notation, PRINT USING left-justifies the number in the format field and adjusts the exponent to compensate, except when printing zero. When printing zero in E notation, BASIC–PLUS–2 prints leading spaces, leading zeros, a decimal point, and zeros in the fractional portion if the PRINT USING string contains these formatting characters, and then the string "E+00."
8. Zero cannot be negative. If a small negative number rounds to zero, it is represented as a positive zero.
9. If there are reserved positions to the left of the decimal point, and the printed number is less than 1, BASIC–PLUS–2 prints one zero to the left of the decimal point and pads with spaces to the left of the zero.
10. If there are more reserved positions to the right of the decimal point than fractional digits, BASIC–PLUS–2 prints trailing zeros in those positions.
11. If there are fewer reserved positions to the right of the decimal point than fractional digits, BASIC–PLUS–2 rounds the number to fit the reserved positions.
12. If a number does not fit in the specified format field, BASIC–PLUS–2 prints a percent sign warning symbol (%), followed by the number in PRINT format.
13. Formatting String Output

    - Format string characters control string output and can be entered only as uppercase characters. All format characters except the backslash and exclamation point must start with a single quotation mark ('). A single quote by itself reserves one character position. A single quote followed by any format characters marks the beginning of a character format field and reserves one character position.

- L reserves one character position. The number of Ls plus the leading single quote determines the field's size. BASIC–PLUS–2 left-justifies the print expression and pads with spaces if the print expression is less than or equal to the field's width. If the print expression is larger than the field, BASIC–PLUS–2 left-justifies the expression and truncates its right side to fit the field.
- R reserves one character position. The number of Rs plus the leading single quote determines the field's size. BASIC–PLUS–2 right-justifies the print expression and pads with spaces if the print expression is less than or equal to the field's width. If the print expression is larger than the field, BASIC–PLUS–2 truncates the right side to fit the field.
- C reserves one character position. The number of Cs plus the leading single quote determines the field's size. If the string does not fit in the field, BASIC–PLUS–2 truncates its right side; otherwise, BASIC–PLUS–2 centers the print expression in this field. If the string cannot be centered exactly, it is offset one character to the left.
- E reserves one character position. The number of Es plus the leading single quote determines the field's size. BASIC–PLUS–2 left-justifies the print expression if it is less than or equal to the field's width and pads with spaces; otherwise, BASIC–PLUS–2 expands the field to hold the entire print expression.
- Two backslashes (\ \) when separated by *n* spaces reserve *n*+2 character positions. PRINT USING left-justifies the string in this field. BASIC–PLUS–2 does not allow a leading quotation mark with this format.
- An exclamation point (!) creates a 1-character field. The exclamation point both starts and ends the field. BASIC–PLUS–2 does not allow a leading quotation mark with this format.

14. BASIC–PLUS–2 interprets any other characters in the format string as string literals and prints them exactly as they appear.
15. If a comma or semicolon follows the last item, the following occurs: *output-list*
    - When printing to a terminal, BASIC–PLUS–2 does not generate a line terminator after printing the last item. The next item printed with a PRINT statement is printed at the position specified by the separator character following the last item in the first PRINT statement.

- When printing to a terminal-format file, BASIC–PLUS–2 does not write out the record until a PRINT statement without trailing punctuation executes.

16. If no punctuation follows the last item in *output-list*, the following occurs:
    - When printing to a terminal, BASIC–PLUS–2 generates a line terminator after printing the last item.
    - When printing to a terminal-format file, BASIC–PLUS–2 writes out the record after printing the last item.

Examples

1.
```
200  PRINT USING "###.###",-12.345
     PRINT USING "##.###",12.345
```

The output is:

```
-12.345
12.345
```

2.
```
30  INPUT "Your Name";Winner$
          Jackpot = 10000.0
    PRINT USING "CONGRATULATIONS, 'EEEEEEEEE, YOU WON $$######.##",
          Winner$, Jackpot
    END
```

The output is:

```
Your Name? Hortense Corabelle
CONGRATULATIONS, Hortense Corabelle, YOU WON $10000.00
```

PROD$

The PROD$ function returns a numeric string that is the product of two numeric strings. The precision of the returned numeric string depends on the value of an integer argument.

Format

str-var = **PROD$**(str-exp1, str-exp2, int-exp)

Syntax Rules

1. *Str-exp1* and *str-exp2* specify the numeric strings you want to multiply. A numeric string can have ASCII digits, an optional minus sign (–), and an optional decimal point (.).
2. If *str-exp* consists of more than 60 characters, BASIC–PLUS–2 signals the error "Illegal number" (ERR=52).
3. *Int-exp* specifies the numeric precision of *str-exp*. Table 4–6 shows examples of rounding and truncation and the values of *int-exp* that produce them.

Remarks

1. *Str-exp* is rounded or truncated, or both, according to the value of *int-exp*.
2. If *int-exp* is between –60 and 60, rounding and truncation occur as follows:
   - For positive integer expressions, rounding occurs to the right of the decimal place. For example, if *int-exp* is 1, rounding occurs one digit to the right of the decimal place (the number is rounded to the nearest tenth). If *int-exp* is 2, rounding occurs two digits to the right of the decimal place (the number is rounded to the nearest hundredth), and so on.
   - If *int-exp* is zero, BASIC–PLUS–2 rounds to the nearest unit.
   - For negative integer expressions, rounding occurs to the left of the decimal point. If *int-exp* is –1, for example, BASIC–PLUS–2 moves the decimal point one place to the left, then rounds to units. If *int-exp* is –2, rounding occurs two places to the left of the decimal point; BASIC–PLUS–2 moves the decimal point two places to the left, then rounds to tens.

3. If *int-exp* is between 9940 and 10,060, truncation occurs as follows:
   - If *int-exp* is 10,000, BASIC–PLUS–2 truncates the number at the decimal point.
   - If *int-exp* is greater than 10,000 (10000 plus *n*) BASIC–PLUS–2 truncates the numeric string *n* places to the right of the decimal point. For example, if *int-exp* is 10,001 (10,000 plus 1), BASIC–PLUS–2 truncates the number starting one place to the right of the decimal point. If *int-exp* is 10,002 (10,000 plus 2), BASIC–PLUS–2 truncates the number starting two places to the right of the decimal point, and so on.
   - If *int-exp* is less than 10,000 (10,000 minus *n*), BASIC–PLUS–2 truncates the numeric string *n* places to the left of the decimal point. For example, if *int-exp* is 9999 (10,000 minus 1), BASIC–PLUS–2 truncates the number starting one place to the left of the decimal point. If 9998 (10,000 minus 2), BASIC–PLUS–2 truncates starting two places to the left of the decimal point, and so on.
4. If *int-exp* is not between –60 and 60 or 9940 and 10,060, BASIC–PLUS–2 returns a value of zero.
5. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to an integer of the default size.

Example

```
10  DECLARE STRING num_exp1, &
                   num_exp2, &
                   product
    num_exp1 = "34.555"
    num_exp2 = "297.676"
    product = PROD$(num_exp1, num_exp2, 1)
    PRINT product
```

The output is:

```
10286.2
```

PROGRAM

The PROGRAM statement allows you to identify a main program with a name other than the file name.

Format

PROGRAM prog-name

Syntax Rules

Prog-name specifies the module name of the compiled source and cannot be the same as any SUB or FUNCTION name.

Remarks

1. The PROGRAM statement must be the first statement in a main program and can be preceded only by comment fields and lexical directives.
2. If you examine the program using the BASIC–PLUS–2 Debugger, the program name you specify will be the module name used.
3. The PROGRAM statement is optional; BASIC–PLUS–2 allows you to specify an END PROGRAM statement and an EXIT PROGRAM statement without a matching PROGRAM statement.

Example

```
20  PROGRAM first_test
    .
    .
    .
    END PROGRAM
```

PUT

The PUT statement transfers data from the record buffer to a file. PUT statements are valid on RMS sequential, relative, indexed, and block I/O files. You cannot use PUT statements on terminal-format files, virtual array files, or files opened with the ORGANIZATION UNDEFINED clause.

Format

PUT #chnl-exp [, **RECORD** num-exp [, **COUNT** int-exp]]

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. The RECORD clause allows you to randomly write records to a relative file by specifying the record number. *Num-exp* must be between 1 and the maximum record number allowed for the file. BASIC–PLUS–2 does not allow you to use the RECORD clause on sequential variable, sequential fixed, sequential stream, or indexed files.
3. *Int-exp* in the COUNT clause specifies the record's size. If there is no COUNT clause, the record's size is that defined by the MAP or RECORDSIZE clause in the OPEN statement. The RECORDSIZE clause overrides the MAP clause.
   - If you write a record to a file with variable-length records, *int-exp* must be an integer from zero through the maximum record size specified in the OPEN statement.
   - If *int-exp* equals zero, the entire record is written to the file.
   - If you write a record to a file with fixed-length records, the COUNT clause serves no purpose. If used, *int-exp* must equal the record size specified in the OPEN statement.

Remarks

1. For sequential access, the file associated with *chnl-exp* must be open with ACCESS WRITE, MODIFY, SCRATCH, or APPEND.
2. For random access, the relative or sequential fixed file associated with *chnl-exp* must be open with ACCESS WRITE or MODIFY.
3. To add records to an existing sequential file, open it with ACCESS APPEND. If you are not at the end of the file when attempting a PUT to a sequential file, BASIC–PLUS–2 signals "Not at end of file" (ERR=149).
4. After a PUT statement executes, there is no current record pointer. The next record pointer is set as follows:
   - For sequential files, variable and stream PUT operations set the next record pointer to the end of the file.
   - For relative and sequentially fixed files, a sequential PUT operation sets the next record pointer to the next record plus 1.
   - For relative and sequential fixed files, a random PUT operation leaves the next record pointer unchanged.
   - For indexed files, a PUT operation leaves the next record pointer unchanged.
5. When you specify a RECORD clause, BASIC–PLUS–2 evaluates *num-exp* and uses this value as the relative record number of the target cell.
   - If the target cell is empty or occupied by a deleted record, BASIC–PLUS–2 places the record in that cell.
   - If there is a record in the target cell and the file has not been opened as a VIRTUAL file, the PUT statement fails, and BASIC–PLUS–2 signals the error "Record already exists" (ERR=153).
6. A PUT statement with no RECORD clause writes records to the file as follows:
   - For sequential variable and stream files, a PUT operation adds a record at the end of the file.
   - For relative and sequential fixed files, a PUT operation places the record in the empty cell pointed to by the next record pointer. If the file is empty, the first PUT operation places a record in cell number 1, the second in cell number 2, and so on.

- For indexed files, RMS stores records in order of ascending primary key value and updates all indexes so that they point to the record.

7. When you open a file as ORGANIZATION VIRTUAL, the file you open is a sequential fixed file with a record size that is a multiple of 512 bytes. You can then access the file with the FIND, GET, PUT, or UPDATE statements or through one or more virtual arrays. BASIC–PLUS–2 allows you to overwrite existing records in a file not containing virtual arrays and opened as ORGANIZATION VIRTUAL by using the PUT statement with a RECORD clause. All other organizations require the UPDATE statement to change an existing record. It is recommended that you also use the UPDATE statement to change existing records in VIRTUAL files that do not contain virtual arrays.
8. If an existing record in an indexed file has a record with the same key value as the one you want to put in the file, BASIC–PLUS–2 signals the error "Duplicate key detected" (ERR=134) if you did not specify DUPLICATES for the key in the OPEN statement. If you specified DUPLICATES, RMS stores the duplicate records in a first-in/first-out sequence.
9. The number specified in the COUNT clause determines how many bytes are transferred from the buffer to a file:
   - If you have not completely filled the record buffer before executing a PUT statement, BASIC–PLUS–2 pads the record with nulls to equal the specified value.
   - If the specified COUNT value is less than the buffer size, the record is truncated to equal the specified value.
   - The number in the COUNT clause must not exceed the size specified in the MAP or RECORDSIZE clause in the OPEN statement or BASIC–PLUS–2 signals "Size of record invalid" (ERR=156).
   - For files with fixed-length records, the number in the COUNT clause must match the record size.
10. Although block I/O files are implemented through RMS–11 on RSX systems when you write a record to a block I/O file, RMS–11 does not perform the same error checking as with relative files. A PUT will write a record to a disk block specified in the RECORD clause, regardless of whether the block already contains a record. See the *BASIC–PLUS–2 User's Guide* for more information on RMS–11 block I/O files.
11. See the *BASIC–PLUS–2 User's Guide* for more information on RSTS/E native mode files.

Examples

1. 10 !Sequential, Relative, Indexed, and Virtual Files
   PUT #3, COUNT 55%

2. 10 !Relative and Virtual Files Only
   PUT #5, RECORD 133, COUNT 16%

QUO$

The QUO$ function returns a numeric string that is the quotient of two numeric strings. The precision of the returned numeric string depends on the value of an integer argument.

Format

str-var = **QUO$**(str-exp1, str-exp2, int-exp)

Syntax Rules

1. *Str-exp1* and *str-exp2* specify the numeric strings you want to divide. A numeric string can have ASCII digits, an optional minus sign (–), and an optional decimal point (.)
2. *Int-exp* specifies the numeric precision of *str-exp*. Table 4–6 shows examples of rounding and truncation and the values of *int-exp* that produce them.

Remarks

1. If *str-exp* consists of more than 60 characters, BASIC–PLUS–2 signals the error "Illegal number" (ERR=52).
2. *Str-exp* is rounded or truncated, or both, according to the value of *int-exp*.
3. If *int-exp* is between –60 and 60, rounding and truncation occur as follows:
   - For positive integer expressions, rounding occurs to the right of the decimal place. For example, if *int-exp* is 1, rounding occurs one digit to the right of the decimal place (the number is rounded to the nearest tenth). If *int-exp* is 2, rounding occurs two digits to the right of the decimal place (the number is rounded to the nearest hundredth), and so on.
   - If *int-exp* is zero, BASIC–PLUS–2 rounds to the nearest unit.
   - For negative integer expressions, rounding occurs to the left of the decimal point. If *int-exp* is –1, for example, BASIC–PLUS–2 moves the decimal point one place to the left, then rounds to units. If *int-exp* is –2, rounding occurs two places to the left of the decimal point; BASIC–PLUS–2 moves the decimal point two places to the left, then rounds to tens.

4. If *int-exp* is between 9940 and 10,060, truncation occurs as follows:
   - If *int-exp* is 10000, BASIC–PLUS–2 truncates the number at the decimal point.
   - If *int-exp* is greater than 10,000 (10,000 plus *n*) BASIC–PLUS–2 truncates the numeric string *n* places to the right of the decimal point. For example, if *int-exp* is 10,001 (10,000 plus 1), BASIC–PLUS–2 truncates the number starting one place to the right of the decimal point. If *int-exp* is 10,002 (10,000 plus 2), BASIC–PLUS–2 truncates the number starting two places to the right of the decimal point, and so on.
   - If *int-exp* is less than 10,000 (10,000 minus *n*), BASIC–PLUS–2 truncates the numeric string *n* places to the left of the decimal point. For example, if *int-exp* is 9999 (10,000 minus 1), BASIC–PLUS–2 truncates the number starting one place to the left of the decimal point. If 9998 (10,000 minus 2), BASIC–PLUS–2 truncates starting two places to the left of the decimal point, and so on.
5. If *int-exp* is not between –60 and 60 or 9940 and 10,060, BASIC–PLUS–2 returns a value of zero.
6. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to an integer of the default size.

Example

```
100  DECLARE STRING num_str1,  &
                    num_str2,  &
                    quotient
     num_str1 = "458996.43"
     num_str2 = "123222.444"
     quotient = QUO$(num_str1, num_str2, 2)
     PRINT quotient
```

The output is:

```
3.72
```

RAD$

The RAD$ function converts a specified integer in Radix–50 format to a 3-character string.

Format

str-var = **RAD$**(int-var)

Syntax Rules

None.

Remarks

1. The RAD$ function converts *int-var* to a 3-character string in Radix–50 format and stores it in *str-var*. Radix–50 format allows you to store three characters of data as a 2-byte integer.
2. If you specify a floating-point variable for *int-var*, BASIC–PLUS–2 truncates it to an integer of the default size.
3. See the *BASIC–PLUS–2 User's Guide* for information on the Radix–50 character set and ASCII/Radix–50 equivalents.

Example

```
100  DECLARE STRING radix
     radix = RAD$(1683)
```

RANDOMIZE

The RANDOMIZE statement gives the random number function RND a new starting value.

Format

```
{ RANDOMIZE }
{ RANDOM    }
```

Syntax Rules

None.

Remarks

1. Without the RANDOMIZE statement, successive runs of the same program generate the same random number sequence.
2. If you use the RANDOMIZE statement before invoking the RND function, the starting point changes for each run. Therefore, a different random number sequence appears each time.

Example

```
20  DECLARE REAL random_num
    RANDOMIZE
        FOR I = 1 TO 2
           random_num = RND
           PRINT random_num
        NEXT I
```

The output is:

```
.379784
.311572
```

RCTRLC

The RCTRLC function disables Ctrl/C trapping.

Format

int-var = **RCTRLC**

Syntax Rules

None.

Remarks

1. After BASIC–PLUS–2 executes the RCTRLC function, a Ctrl/C entered at the terminal returns you to DCL command level or to the BASIC environment.
2. RCTRLC always returns a value of zero.

Example

```
100  Y = RCTRLC
```

RCTRLO

The RCTRLO function cancels the effect of a C/o entered on a terminal opened on a specified channel.

Format

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

Syntax Rules

Chnl-exp is a numeric expression that specifies a channel number associated with an open terminal. It cannot be preceded by a number sign (#).

Remarks

1. If you enter a Ctrl/O to cancel terminal output, nothing is printed on the specified terminal until your program executes the RCTRLO or until you enter another Ctrl/O, at which time normal terminal output resumes.
2. The RCTRLO function always returns a value of zero.
3. RCTRLO has no effect if the specified channel is open to a device that does not use the Ctrl/O convention.

Example

```
10  PRINT "A" FOR I% = 1% TO 10%
    Y% =  RCTRLO(0%)
    PRINT "Normal output is resumed"
```

The output is:

```
A
A
A
A
```

READ

The READ statement assigns values from a DATA statement to variables.

Format

READ var, . . .

Syntax Rules

None.

Remarks

1. If your program has a READ statement without DATA statements, BASIC–PLUS–2 signals a compile-time error.
2. When BASIC–PLUS–2 initializes a program unit, it forms a data sequence of all values in all DATA statements. An internal pointer points to the first value in the sequence.
3. When BASIC–PLUS–2 executes a READ statement, it sequentially assigns values from the data sequence to variables in the READ statement variable list. As BASIC–PLUS–2 assigns each value, it advances the internal pointer to the next value.
4. BASIC–PLUS–2 signals the error "Out of data" (ERR=57) if there are fewer data elements than READ statements. Extra data elements are ignored.
5. The data type of the value must agree with the data type of the variable to which it is assigned or BASIC–PLUS–2 signals "Data format error" (ERR=50).
6. If you read a string variable and the DATA element is an unquoted string, BASIC–PLUS–2 ignores leading and trailing spaces. If the DATA element contains any commas, they must be inside quotation marks.
7. BASIC–PLUS–2 evaluates subscript expressions in the variable list after it assigns a value to the preceding variable, and before it assigns a value to the subscripted variable. For instance, in the following example, BASIC–PLUS–2 assigns the value of 10 to variable A, then assigns the string LESTER to array element A$(A).

```
100  READ A, A$(10)
   .
   .
   .
900  DATA 10, LESTER
```

The string, LESTER, will be assigned to A$(10).

Example

```
10  DECLARE STRING A,B,C
    READ A,B,C
    DATA "X", "Y", "Z"
    PRINT A + B + C
```

The output is:

```
XYZ
```

REAL

The REAL function converts a numeric expression or numeric string to a specified or default floating-point data type.

Format

real-var = **REAL**(exp [**,SINGLE** / **,DOUBLE**])

Syntax Rules

Exp can be either numeric or string. If a string, it can contain the ASCII digits 0 through 9, uppercase E, a plus sign (+), a minus sign (–), and a period (.).

Remarks

1. BASIC–PLUS–2 evaluates *exp*, then converts it to the specified REAL size. If you do not specify a size, BASIC–PLUS–2 uses the default REAL size.
2. BASIC–PLUS–2 ignores leading and trailing spaces and tabs if *exp* is a string.
3. The REAL function returns a value of zero when a string argument contains only spaces and tabs, or when the argument is null.

Example

```
100  DECLARE STRING any_num
     INPUT "Enter a number";any_num
     PRINT REAL(any_num, DOUBLE)
```

The output is:

```
Enter a number? 123095959
 .123096E+09
```

RECOUNT

The RECOUNT function returns the number of characters transferred by the last input operation.

Format

int-var = **RECOUNT**

Syntax Rules

None.

Remarks

1. The RECOUNT value is reset by every input operation on any channel, including channel #0.
   - After an input operation from your terminal, RECOUNT contains the number of characters (bytes) transferred (including line terminators).
   - After accessing a file record, RECOUNT contains the number of characters in the record.
2. Because RECOUNT is reset by every input operation on any channel, you should copy the RECOUNT value to a different storage location before executing another input operation.
3. If an error occurs during an input operation, the value of RECOUNT is undefined.
4. RECOUNT is unreliable after a Ctrl/C interrupt because the Ctrl/C trap may have occurred before BASIC–PLUS–2 set the value for RECOUNT.
5. The RECOUNT function returns a WORD value.

Example

```
200  DECLARE INTEGER character_count
     INPUT "Enter a sequence of numeric characters";character_count
     character_count = RECOUNT
     PRINT character_count;"characters received (including CR and LF)"
```

The output is:

```
Enter a sequence of numeric characters? 12345678
 10 characters received (including CR and LF)
```

REM

The REM statement allows you to document your program.

Format

REM [comment]

Syntax Rules

1. REM must be the only statement on the line or the last statement on a multi-statement line.
2. BASIC–PLUS–2 interprets every character between the keyword REM and the next line number as part of the comment.
3. Because the REM statement is not executable, you can place it anywhere in a program, except where other statements, such as SUB and END SUB, must be the first or last statement in a program unit.

Remarks

1. When the REM statement is the first statement on a line-numbered line, BASIC–PLUS–2 treats any reference to that line number as a reference to the next higher-numbered executable statement.
2. The REM statement is similar to the comment field that begins with an exclamation point, with one exception: the REM statement must be the last statement on a BASIC line. The exclamation point comment field can be ended with another exclamation point or a line terminator and followed by a BASIC–PLUS–2 statement. See Chapter 1 of this manual for more information on the comment field.

Example

```
10 REM This is a multi-line comment
   All text up to BASIC line 20 is a part of this REM statement.
   Any BASIC statements on line 10 are ignored.
 PRINT "This does not execute".

20 PRINT "This will execute"
```

The output is:

```
This will execute
```

REMAP

The REMAP statement defines or redefines the position in the storage area of variables named in the MAP DYNAMIC statement.

Format

REMAP (map-dyn-name) remap-item, . . .

map-dyn-name: { map-name | static-str-var }

remap-item: { num-var | num-array-name ([int-exp, . . .]) | str-var [= int-exp] | str-array-name ([int-exp, . . .])[= int-exp] | [data-type] **FILL** [(int-exp)] [= int-exp] | **FILL%** [(int-exp)] | **FILL$** [(int-exp)][= int-exp] }

Syntax Rules

1. *Map-dyn-name* can be either a map name or a static string variable.
   - *Map-name* is the storage area named in a MAP statement.
   - If you specify a map name, then a MAP statement with the same name must precede both the MAP DYNAMIC statement and the REMAP statement.
   - When you specify a static string variable, the string must be declared before you can specify a MAP DYNAMIC statement or a REMAP statement.
   - If you specify a *static-str-var*, the following restrictions apply:
     - *Static-str-var* cannot be a string constant.
     - *Static-str-var* cannot be the same as any previously declared *map-item* in a MAP DYNAMIC statement.
     - *Static-str-var* cannot be a subscripted variable.

2. *Remap-item* names a variable, array, or array element declared in a preceding MAP DYNAMIC statement:
   - *Num-var* specifies a numeric variable or array element. *Num-array-name* followed by a set of empty parentheses specifies an entire numeric array.
   - *Str-var* specifies a string variable or array element. *Str-array-name* followed by a set of empty parentheses specifies an entire fixed-length string array. You can specify the number of bytes to be reserved for string variables and array elements with the *=int-exp* clause. The default string length is 16.
3. *Remap-item* can also be a FILL item. The FILL, FILL%, and FILL$ keywords let you reserve parts of the record buffer. *Int-exp* specifies the number of FILL items to be reserved. The *=int-exp* clause allows you to specify the number of bytes to be reserved for string FILL items. Table 4–2 describes FILL item format and storage allocation.

   Note

   In the FILL clause, *(int-exp)* represents a repeat count, not an array subscript. FILL (*n*), for example, represents *n* elements, not *n* + 1.

4. All *remap-items*, except FILL items, must have been named in a previous MAP DYNAMIC statement, or BASIC–PLUS–2 signals an error.
5. *Data-type* can be any BASIC–PLUS–2 data type keyword. Data type keywords and their size, range, and precision are listed in Table 1-2 in this manual. You can specify a data type only for FILL items. If you do not specify a data type, FILL items take the current default data type and size.
6. *Remap-items* must be separated with commas.

Remarks

1. The REMAP statement does not affect the amount of storage allocated to the map area.
2. Each time a REMAP statement executes, BASIC–PLUS–2 sets record pointers to the named map area for the specified variables from left to right.

3. The REMAP statement must be preceded by a MAP DYNAMIC statement or BASIC–PLUS–2 signals the error "No such MAP area <name>." The MAP statement or static string variable creates a named area of static storage, the MAP DYNAMIC statement specifies the variables whose positions can change at run time, and the REMAP statement specifies the new positions for the variables names in the MAP DYNAMIC statement.

4. Before you can specify a map name in a REMAP statement, there must be a MAP statement in the program unit with the same map name; otherwise, BASIC–PLUS–2 signals the error "<Name> is not a DYNAMIC MAP variable of MAP <name>." Similarly, before you can specify a static string variable in a REMAP statement, the string variable must be declared or BASIC–PLUS–2 signals the same error message.

5. If a static string variable is the same as a map name, BASIC–PLUS–2 overrides the static string name and uses the map name.

6. Until the REMAP statement executes, all variables named in the MAP DYNAMIC statement point to the first byte of the MAP area and all string variables have a length of zero. When the REMAP statement executes, BASIC–PLUS–2 sets the internal pointers as specified in the REMAP statement. For example:

```
100     MAP (DUMMY) STRING map_buffer = 50
        MAP DYNAMIC (DUMMY) LONG A, STRING B, SINGLE C(7)
        REMAP (DUMMY) B=14, A, C()
```

   The REMAP statement sets a pointer to byte 1 of *DUMMY* for string variable *B*, a pointer to byte 15 for LONG variable *A*, and pointers to bytes 19, 23, 27, 31, 35, 39, 43, and 47 for the elements in SINGLE array *C*.

7. You can use the REMAP statement to redefine the pointer for an array element or variable more than once in a single REMAP statement. For example:

```
100     MAP (DUMMY) STRING FILL = 48
        MAP DYNAMIC (DUMMY) LONG A, B(10)
        REMAP (DUMMY) B(), B(0)
```

   This REMAP statement sets a pointer to byte 1 in *DUMMY* for array *B*. Because array *B* uses a total of 44 bytes, the pointer for the first element of array *B*, *B*(0) points to byte 45. References to array element *B*(0) will be to bytes 45 through 48. Pointers for array elements 1 through 10 are set to bytes 5, 9, 13, 17, and so forth.

8. Because the REMAP statement is local to a program module, it affects pointers only in the program module in which it executes.

Examples

1.
```
20  DECLARE LONG CONSTANT emp_fixed_info = 4 + 9 + 2
    MAP (employ) LONG badge,                          &
                 STRING social_sec_num = 9,           &
                 BYTE name_length,                    &
                      address_length,                 &
                      FILL (60)
    MAP DYNAMIC (employ) STRING emp_name,             &
                                emp_address
    WHILE 1%
          GET #1
          REMAP (employ) STRING FILL = emp_fixed_info,          &
                                emp_name = name_length,         &
                                emp_address = address_length

          PRINT emp_name
          PRINT emp_address
          PRINT

    NEXT

    END
```

2.
```
900  SUB deblock (STRING input_rec, STRING item())
     MAP DYNAMIC (input_rec) STRING A(3)
     REMAP (input_rec) &
         A(1) = 5, &
         A(2) = 3, &
         A(3) = 4
     FOR I = 1 TO 3
       item(I) = A(I)
     NEXT I
     END SUB
```

RESET

The RESET statement is a synonym for the RESTORE statement. See the RESTORE statement for more information.

Format

RESET [#chnl-exp [, **KEY** #int-exp]]

RESTORE

The RESTORE statement resets the DATA pointer to the beginning of the DATA sequence, or sets the record pointer to the first record in a file.

Format

RESTORE [#chnl-exp [, **KEY** #int-exp]]

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Int-exp* must be an integer from zero through the number of keys in the file minus 1. It must be immediately preceded by a number sign (#).

Remarks

1. If you do not specify a channel, RESTORE resets the DATA pointer to the beginning of the DATA sequence.
2. RESTORE affects only the current program unit. Thus, executing a RESTORE statement in a subprogram does not affect the DATA pointer in the main program.
3. If there is no channel specified, and the program has no DATA statements, RESTORE has no effect.
4. The file specified by *chnl-exp* must be open.
5. If *chnl-exp* specifies a magnetic tape file, BASIC–PLUS–2 rewinds the tape to the first record in the file.
6. The KEY clause applies to indexed files only. It sets a new key of reference equal to *int-exp* and sets the next record pointer to the first logical record in that key.
7. For indexed files, the RESTORE statement without a KEY clause sets the next record pointer to the first logical record specified by the current key of reference. If there is no current key of reference, the RESTORE statement sets the next record pointer to the first logical record of the primary key.
8. If you use the RESTORE statement on any file type other than indexed, BASIC–PLUS–2 sets the next record pointer to the first record in the file.

RESUME

The RESUME statement marks an exit point from an ON ERROR error-handling routine. BASIC–PLUS–2 clears the error condition and returns program control to a specified line number or label, or to the program block in which the error occurred.

Format

RESUME [target]

Syntax Rules

Target must be a valid BASIC–PLUS–2 line number or label and must exist in the same program unit.

Remarks

1. The following restrictions apply:
   - The RESUME statement cannot be used in a multi-line DEF unless the target is also in the DEF function definition.
   - The execution of a RESUME with no target is illegal if there is no error active.
   - A RESUME statement cannot transfer control out of the current program unit. Therefore, a RESUME statement with no target cannot terminate an error handler if the error handler is handling an error that occurred in a subprogram or an external function, and the error was passed to the calling program's error handler by an ON ERROR GO BACK statement or by default.
2. When no target is specified in a RESUME statement, BASIC–PLUS–2 transfers control based on where the error occurs. If the error occurs on a numbered line containing a single statement, BASIC–PLUS–2 always transfers control to that statement. When the error occurs within a multi-statement line under the following conditions, BASIC–PLUS–2 acts as follows:
   - Within a FOR, WHILE, or UNTIL loop, BASIC–PLUS–2 transfers control to the first statement that follows the FOR, WHILE, or UNTIL statement.

9. The RESTORE statement is not allowed on virtual array files or on files opened on unit record devices.

Example

```
200  RESTORE #7%, KEY #4%
```

RETURN

The RETURN statement transfers control to the statement immediately following the most recently executed GOSUB or ON . . . GOSUB statement in the current program unit.

Format

RETURN

Syntax Rules

None.

Remarks

1. Once the RETURN is executed in a subroutine, no other statements in the subroutine are executed, even if they appear after the RETURN statement.
2. Execution of a RETURN statement before the execution of a GOSUB or ON . . . GOSUB causes BASIC–PLUS–2 to signal "RETURN without GOSUB" (ERR=72).

Example

```
100  GOSUB subroutine_1
   .
   .
   .
     subroutine_1:
   .
   .
   .
     RETURN
```

- Within a SELECT block, BASIC–PLUS–2 transfers control to the start of the CASE block in which the error occurs.
- After a loop or SELECT block, BASIC–PLUS–2 transfers control to the statement that follows the NEXT or END SELECT statement.
- If none of the above conditions occurs, BASIC–PLUS–2 transfers control back to the statement that follows the most recent line number.

3. A RESUME statement with a specified line number transfers control to the first statement of a multi-statement line, regardless of which statement caused the error.
4. A RESUME statement with a specified label transfers control to the block of code indicated by that label.
5. When BASIC–PLUS–2 executes a RESUME statement, it clears the error condition.
6. After a Ctrl/C trap, a RESUME statement with no line number returns control to the statement immediately following the previous line number.

Example

```
10  Error_routine:
    IF ERR = 11
       THEN
          CLOSE #1
          RESUME end_of_prog
    ELSE
          RESUME
    END IF
    end_of_prog: END
```

RND

The RND function returns a random number greater than or equal to zero and less than 1.

Format

real-var = **RND**

Syntax Rules

None.

Remarks

1. If the RND function is preceded by a RANDOMIZE statement, BASIC–PLUS–2 generates a different random number or series of numbers each time a program executes.
2. The RND function returns a pseudorandom number if not preceded by a RANDOMIZE statement; that is, each time a program runs, BASIC–PLUS–2 generates the same random number or series of random numbers.
3. The RND function returns a floating-point value of the default size.

Example

```
40  DECLARE REAL random_num
    RANDOMIZE
    FOR I = 1 TO 3   !FOR loop causes BASIC to print three random numbers
            random_num = RND
            PRINT random_num
    NEXT I
```

The output is:

```
.865243
.477417
.734673
```

RIGHT$

The RIGHT$ function extracts a substring from a string's right side, leaving the string unchanged.

Format

str-var = **RIGHT[$]** *(str-exp, int-exp)*

Syntax Rules

None.

Remarks

1. The RIGHT$ function extracts a substring from *str-exp* and stores the substring in *str-var*. The substring begins with the character in the position specified by *int-exp* and ends with the rightmost character in the string.
2. The length of the resulting string is the number of characters in *str-exp* minus *int-exp*.
3. If *int-exp* is less than or equal to zero, RIGHT$ returns the entire string.
4. If *int-exp* is greater than the length of *str-exp*, RIGHT$ returns a null string.
5. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to a WORD integer.

Example

```
50  DECLARE STRING main_str,   &
                   end_result
    main_str = "1234567"
    end_result = RIGHT$(main_str, 3)
    PRINT end_result
```

The output is:

```
34567
```

SCRATCH

The SCRATCH statement deletes the current record and all following records in a sequential file.

Format

SCRATCH #chnl-exp

Syntax Rules

Chnl-exp is a numeric expression that specifies a channel associated with a file. It must be immediately preceded by a number sign (#).

Remarks

1. The SCRATCH statement applies to ORGANIZATION SEQUENTIAL files only.
2. Before you execute the SCRATCH statement, the file must be opened with ACCESS SCRATCH.
3. The SCRATCH statement has no effect on terminals or unit record devices.
4. For disk files, the SCRATCH statement discards the current record and all that follows it in the file. The physical length of the file does not change.
5. For magnetic tape files, the SCRATCH statement overwrites the current record with two end-of-file marks.

Example

```
600  SCRATCH #4%
```

RSET

The RSET statement assigns right-justified data to a string variable. RSET does not change a string variable's length.

Format

RSET str-var, . . . = str-exp

Syntax Rules

None.

Remarks

1. The RSET statement treats strings as fixed-length. It does not change the length of *str-var*, nor does it create new storage locations.
2. If *str-var* is longer than *str-exp*, RSET right-justifies the data and pads it with spaces on the left.
3. If *str-var* is shorter than *str-exp*, RSET truncates *str-exp* on the left.
4. With string virtual arrays, RSET changes the length of *str-exp* to the declared length by padding it with spaces on the right. Note that the LET statement uses null characters for padding.

Example

```
20  DECLARE STRING test
    test = "ABCDE"
    RSET test = "123"
    PRINT "X" + test
```

The output is:

```
X  123
```

Example

```
10  DECLARE STRING alpha, center
    alpha = "ABCDEFGHIJK"
    center = SEG$(alpha, 4, 8)
    PRINT center
```

The output is:

```
DEFGH
```

SEG$

The SEG$ function extracts a substring from a main string, leaving the original string unchanged.

Format

str-var = **SEG$**(str-exp, int-exp1, int-exp2)

Syntax Rules

None.

Remarks

1. BASIC–PLUS–2 extracts the substring from *str-exp*, the main string, and stores the substring in *str-var*. The substring begins with the character in the position specified by *int-exp1* and ends with the character in the position specified by *int-exp2*.
2. If *int-exp1* is less than 1, BASIC–PLUS–2 assumes a value of 1.
3. If *int-exp1* is greater than *int-exp2* or the length of *str-exp*, the SEG$ function returns a null string.
4. If *int-exp1* equals *int-exp2*, the SEG$ function returns the character at the position specified by *int-exp1*.
5. Unless *int-exp2* is greater than the length of *str-exp*, the length of the returned substring equals *int-exp2* minus *int-exp1* plus 1. If *int-exp2* is greater than the length of *str-exp*, the SEG$ function returns all characters from the position specified by *int-exp1* to the end of *str-exp*.
6. If you specify a floating-point expression for *int-exp1* or *int-exp2*, BASIC–PLUS–2 truncates it to WORD integer.

- *Exp3* and *exp4* specify a range of numeric or string values separated by the keyword TO. Multiple ranges must be separated with commas. BASIC–PLUS–2 executes the statements in the CASE block when *exp1* falls within any of the specified ranges.

5. A SELECT statement can have only one *else-clause*. The *else-clause* is optional and, when present, must be the last CASE block in the SELECT block.
6. The SELECT statement begins the SELECT BLOCK and the END SELECT keywords terminate it. BASIC–PLUS–2 signals an error if you do not include the END SELECT keywords.
7. Each CASE keyword establishes a CASE block. The next CASE or END SELECT keyword ends the CASE block.
8. You can nest SELECT blocks within a CASE or CASE ELSE block.

Remarks

1. Each statement in a SELECT block can have its own line number.
2. BASIC–PLUS–2 evaluates *exp1* when the SELECT statement is first encountered; BASIC–PLUS–2 then compares *exp1* with each *case-clause* in order of occurrence until a match is found or until a CASE ELSE block or END SELECT is encountered.
3. The following conditions constitute a match:
   - *Exp1* satisfies the relationship to *exp2* specified by *rel-op*.
   - *Exp1* is greater than or equal to *exp3* but less than or equal to *exp4*, greater than or equal to *exp5* but less than or equal to *exp6*, and so on.
4. When a match is found between *exp1* and a *case-item*, BASIC–PLUS–2 executes the statements in the CASE block where the match occurred. If ranges overlap, the first match causes BASIC–PLUS–2 to execute the statements in the CASE block. After executing CASE block statements, control passes to the statement immediately following the END SELECT keywords.
5. If no CASE match occurs, BASIC–PLUS–2 executes the statements in the *else-clause*, if present, and then passes control to the statement immediately following the END SELECT keywords.
6. If no CASE match occurs and you do not supply a *case-else* clause, control passes to the statement following the END SELECT keywords.

Example

```
100     SELECT A% + B% + C%
                CASE = 100
                        PRINT 'THE VALUE IS EXACTLY 100'
                CASE 1 TO 99
                        PRINT 'THE VALUE IS BETWEEN 1 AND 99'
                CASE > 100
                        PRINT 'THE VALUE IS GREATER THAN 100'
                CASE ELSE
                        PRINT 'THE VALUE IS LESS THAN 1'
        END SELECT
```

SET [NO] PROMPT

The SET PROMPT statement enables a question mark prompt to appear after BASIC–PLUS–2 executes either an INPUT, LINPUT, INPUT LINE, MAT INPUT, or MAT LINPUT statement on channel #0. The SET NO PROMPT statement disables the question mark prompt.

Format

SET [NO] PROMPT

Syntax Rules

None.

Remarks

1. If you do not specify a SET PROMPT statement, the default is SET PROMPT.
2. SET NO PROMPT disables BASIC–PLUS–2 from issuing a question mark prompt for the INPUT, LINPUT, INPUT LINE, MAT INPUT, and MAT LINPUT statements on channel #0.
3. Prompting is reenabled when either a SET PROMPT statement or a CHAIN statement is executed, or when a NEW, OLD, RUN or SCRATCH command is executed in the BASIC environment.
4. The SET NO PROMPT statement does not affect the string constant you specify as the input prompt with the INPUT statement.

Example

```
20  DECLARE STRING your_name, your_age, your_grade
    INPUT "Enter your name";your_name
    SET NO PROMPT
    INPUT "Enter your age";your_age
    SET PROMPT
    INPUT "Enter the last school grade you completed";your_grade
```

The output is:

```
Enter your name? Katherine Kelly
Enter your age 15
Enter the last school grade you completed? 9
```

SGN

The SGN function determines whether a numeric expression is positive, negative, or zero. It returns a 1 if the expression is positive, a –1 if the expression is negative, and zero if the expression is zero.

Format

int-var = **SGN**(real-exp)

Syntax Rules

None.

Remarks

1. If *real-exp* does not equal zero, SGN returns MAG(*real-exp*)/*real-exp*.
2. If *real-exp* equals zero, SGN returns a value of zero.
3. SGN returns a WORD integer.

Example

```
10  DECLARE INTEGER sign
    sign = SGN(46/23)
    PRINT sign
```

The output is:

```
1
```

SIN

The SIN function returns the sine of an angle in radians.

Format

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

Syntax Rules

Real-exp is an angle specified in radians.

Remarks

1. The returned value is between –1 and 1.
2. BASIC–PLUS–2 expects the argument of the SIN function to be a real expression. When the argument is a real expression, BASIC–PLUS–2 returns a value of the same floating-point size. When the argument is not a real expression, BASIC–PLUS–2 converts the argument to the default floating-point size and returns a value of the default floating-point size.

Example

```
100  s1_angle = SIN(PI/2)
     PRINT s1_angle
```

The output is:

```
1
```

SLEEP

The SLEEP statement suspends program execution for a specified number of seconds or until a carriage return is entered from the controlling terminal.

Format

SLEEP int-exp

Syntax Rules

1. *Int-exp* is the number of seconds BASIC–PLUS–2 waits before resuming program execution.
2. *Int-exp* must be an integer from 0 through 32767; if it is greater than 32767, BASIC–PLUS–2 signals the error "Integer error" (ERR=51).

Remarks

Pressing the Return key on the controlling terminal cancels the effect of the SLEEP statement.

Example

```
60  SLEEP 120%
```

SPACE$

The SPACE$ function creates a string containing a specified number of spaces.

Format

str-var = **SPACE$**(int-exp)

Syntax Rules

Int-exp specifies the number of spaces in the returned string.

Remarks

1. BASIC–PLUS–2 treats an *int-exp* less than zero as zero.
2. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to a WORD integer.

Example

```
10  DECLARE STRING A, B
    A = "1234"
    B = "5678"
    PRINT A + SPACE$(5%) + B
```

The output is:

```
1234     5678
```

SQR

The SQR function returns the square root of a positive number.

Format

real-var = { **SQRT** / **SQR** } (real-exp)

Syntax Rules

None.

Remarks

1. BASIC–PLUS–2 signals the error "Imaginary square roots" (ERR=54), when *real-exp* is negative and returns the square root of the absolute value of the expression.
2. BASIC–PLUS–2 assumes that the argument of the SQR function is a real expression. When the argument is a real expression, BASIC–PLUS–2 returns a value of the same floating-point size. When the argument is not a real expression, BASIC–PLUS–2 returns a value of the default floating-point size.

Example

```
100  DECLARE REAL root
     root = SQR(20*5)
     PRINT root
```

The output is:

```
10
```

STATUS

The STATUS function returns a word-length integer value containing information about the last opened channel. Your program can test each bit to determine the status of the channel.

Format

int-var = **STATUS**

Syntax Rules

None.

Remarks

1. The STATUS function returns a WORD integer.
2. The value returned by the STATUS function is undefined until BASIC–PLUS–2 executes an OPEN statement.
3. The STATUS value is reset by every input operation on any channel. Therefore, you should copy the STATUS value to a different storage location before your program executes another input operation.
4. On RSTS/E systems, depending on the error, the STATUS function displays a value representing one of the following:
   - The RMS–11 primary status field (STS) or the RMS–11 secondary status field (STV). See the *RSTS/E RMS–11 MACRO Programmer's Guide* for more information.
   - The device characteristics after an RMS–11 OPEN file operation (set by the DEV field of the FAB). See the *RSTS/E RMS–11 MACRO Programmer's Guide* system for more information.
   - For OPEN operations where no error occurs, the status word describes the device characteristics of the FIRQB and FQFLAG field. The first 7 bits describe the device, and bits 7 through 15 describe characteristics of the OPEN statement. See the *BASIC–PLUS Language Manual* or the *RSTS/E System Directives Manual* for more information on STATUS values.

5. On RSX systems, depending on the error, the STATUS function displays a value representing one of the following:

   - The RMS–11 primary status field (STS) or the RMS–11 secondary status field (STV). See the *RSX–11M/M–PLUS RMS–11 Macro Programmer's Guide* for more information.
   - The device characteristics after an RMS–11 OPEN file operation (set by the DEV field of the FAB). See the *RSX–11M/M–PLUS RMS–11 Macro Programmer's Guide* for more information.
   - In the event of a directive error, the Directive Status Word ($DSW) and its corresponding error code. See the *RSX–11M/M–PLUS Mini Reference* for the error codes.
   - The STATUS field of a QIO. See the *RSX–11M/M–PLUS I/O Drivers Reference Manual* for more information.
   - The first word of a GETLUN or GLUN$ directive describing device characteristics. See the *RSX–11M/M–PLUS and Micro/RSX Executive Reference Manual* for more information.

   See Table 4–7 for a list of the values of the STATUS word for OPEN operations where no errors occur.

Table 4–7 RSX STATUS Values

Value	Bit Set	Meaning
1	0	Record-oriented device
2	1	Carriage-control device
4	2	Terminal device
8	3	Directory device
16	4	Single-directory device
32	5	Sequential device
64	6	Mass storage device
128	7	User-mode diagnostics supported
256	8	Massbus device
512	9	Unit software write-locked
1024	10	Input spooled device

(continued on next page)

Table 4–7 (Cont.) RSX STATUS Values

Value	Bit Set	Meaning
2048	11	Output spooled device
4096	12	Pseudo device
8192	13	Device mountable as a communication
16384	14	Device mountable as a Files–11 device
-32768	15	Device mountable

Example

```
150       Y% = STATUS
```

STOP

The STOP statement halts program execution allowing you to optionally continue execution.

Format

STOP

Syntax Rules

None.

Remarks

1. The STOP statement cannot appear before a PROGRAM, SUB, or FUNCTION statement.
2. The STOP statement does not close files.
3. When a STOP statement executes in a program executed with the environment RUN/DEBUG command or compiled with the /DEBUG qualifier, control passes to the BASIC–PLUS–2 debugger. The debugger prints the line number and module name associated with the STOP statement, then displays the number sign (#) prompt. You can then use the BASIC–PLUS–2 debugger commands to analyze and debug your program. See B for a description of the BASIC–PLUS–2 debugger commands. Once you are finished debugging your program, use the EXIT command to exit from the debugger and end the program.
4. When a STOP statement executes in a program executed with the RUN command or compiled without the /DEBUG qualifier, the line number of the STOP statement and a number sign (#) prompt are printed. In response to the prompt, you can either enter the debugger command CONTINUE to continue program execution, or the EXIT command to end the program. The EXIT command closes all files before leaving the program.

Example

```
40  PRINT "Type CONTINUE when the program stops"
    INPUT "Do you want to stop now"; Quit$

    IF Quit$ = "Y"
       THEN
           STOP
       ELSE
           PRINT "So what are you waiting for?"
           STOP
    END IF
```

The output is:

```
Type CONTINUE when the program stops
Do you want to stop now? Y
Stop at line 40

#CONTINUE

BASIC2
```

STR$

The STR$ function changes a numeric expression to a numeric character string without leading and trailing spaces.

Format

str-var = **STR$**(num-exp)

Syntax Rules

None.

Remarks

1. If *num-exp* is negative, the first character in the returned string is a minus sign (–).
2. The STR$ function produces E notation.
3. When you print a floating-point number that has 6 decimal digits or more but the integer portion has 6 digits or less (for example, 1234.567), BASIC–PLUS–2 rounds the number to 6 digits (1234.57). If a floating-point number's integer part is 7 decimal digits or more, BASIC–PLUS–2 rounds the number to 6 digits and prints it in E format.
4. When you print a floating-point number with magnitude between 0.1 and 1, BASIC–PLUS–2 rounds it to 6 digits. When you print a number with magnitude smaller than 0.1, BASIC–PLUS–2 rounds it to 6 digits and prints it in E format.

Example

```
100  DECLARE STRING new_num
     new_num = STR$(1543.659)
     PRINT new_num
```

The output is:

```
1543.66
```

STRING$

The STRING$ function creates a string containing a specified number of identical characters.

Format

str-var = **STRING$**(int-exp1, int-exp2)

Syntax Rules

1. *Int-exp1* specifies the character string's length.
2. *Int-exp2* is the decimal ASCII value of the character that makes up the string. This value is treated modulo 256.

Remarks

1. BASIC–PLUS–2 signals the error "Integer error" (ERR=51), if *int-exp1* is greater than 32767.
2. If *int-exp1* is less than or equal to zero, BASIC–PLUS–2 treats it as zero.
3. BASIC–PLUS–2 treats *int-exp2* as an unsigned 8-bit integer. For example, –1 is treated as 255.
4. If either *int-exp1* or *int-exp2* is a floating-point expression, BASIC–PLUS–2 truncates it to a WORD integer.

Example

```
40  DECLARE STRING output_str
    output_str = STRING$(10%, 50%)  !50 is the ASCII value of the
    PRINT output_str                !character "2"
```

The output is:

```
2222222222
```

SUB

The SUB statement marks the beginning of a BASIC–PLUS–2 subprogram and specifies the number and data type of its parameters.

Format

SUB sub-name [(formal-param, . . .)] [statement] . . .
[statement]...

{ **END SUB** }
{ **SUBEND** }

formal-param: [data-type] { unsubs-var
array-name ([int-const] , ...) }

Syntax Rules

1. The SUB statement must be the first statement in the SUB subprogram.
2. *Sub-name* is the name of the separately compiled subprogram.
3. *Formal-param* specifies the number and type of parameters for the arguments the SUB subprogram expects to receive when invoked.
   - Empty parentheses indicate that the SUB subprogram has no parameters.
   - *Data-type* specifies the data type of a parameter. If you do not specify a data type, parameters are of the default data type and size. When you do specify a data type, all following parameters are of that data type until you specify a new data type. Data type keywords and their size, range, and precision are listed in Table 1-2 in this manual.
4. *Sub-name* can have from one through six characters and must conform to the following rules:
   - The first character of an unquoted name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), or periods (.).

- A quoted name can consist of any combination of alphabetic characters, digits, dollar signs ($), periods (.), or spaces.

5. Parameters defined in *formal-param* must agree in number, type, and ordinality with the arguments specified in the CALL statement of the calling program.
6. You can specify up to 32 formal parameters. MACRO–11 subprograms accept a maximum of 255 parameters.
7. Each SUB statement must have a corresponding END SUB statement or SUBEND statement.

Remarks

1. Compiler directives and comment fields created with an exclamation point (!) can precede the SUB statement because they are not BASIC–PLUS–2 statements. Note that REM is a BASIC–PLUS–2 statement; therefore, it cannot precede the SUB statement.
2. Any BASIC–PLUS–2 statement except the FUNCTION statement can appear in a SUB subprogram.
3. All variables, except those named in MAP and COMMON statements, are local to that subprogram.
4. BASIC–PLUS–2 initializes local variables to zero or the null string.
5. In BASIC–PLUS–2 you cannot specify how subprograms receive parameters. BASIC–PLUS–2 subprograms receive numeric unsubscripted variables by reference and string unsubscripted variables by descriptor. Table 4–1 lists and describes BASIC–PLUS–2 parameter-passing mechanisms.
   - BY REF specifies that the subprogram receives the argument's address.
   - BY DESC specifies that the subprogram receives the address of a BASIC–PLUS–2 descriptor. For information about the format of a BASIC–PLUS–2 descriptor for strings and arrays, see the *BASIC–PLUS–2 User's Guide*.
6. You cannot call subprograms recursively.
7. The default error handling for SUB subprogams is ON ERROR GO BACK.

Example

```
100  SUB SUBPRO (BYTE AGE, DOUBLE WAGE(20,20), STRING EMP_NAME)
   .
   .
   .
900  END SUB
```

SUBEND

The SUBEND statement is a synonym for END SUB. See the END statement for more information.

Format

SUBEND

SUBEXIT

The SUBEXIT statement is a synonym for the EXIT SUB statement. See the EXIT statement for more information.

Format

SUBEXIT

SUM$

The SUM$ function returns a string whose value is the sum of two numeric strings.

Format

str-var = **SUM$**(str-exp1, str-exp2)

Syntax Rules

None.

Remarks

1. Each string expression can contain up to 54 ASCII digits and an optional decimal point and sign.
2. BASIC–PLUS–2 adds *str-exp2* to *str-exp1* and stores the result in *str-var*.
3. If *str-exp1* and *str-exp2* are integers, *str-var* takes the precision of the larger string unless trailing zeros generate that precision.
4. If *str-exp1* and *str-exp2* are decimal fractions, *str-var* takes the precision of the more precise fraction unless trailing zeros generate that precision.
5. SUM$ omits trailing zeros to the right of the decimal point.
6. The sum of two fractions takes precision as follows:
   - The sum of the integer parts takes the precision of the larger part.
   - The sum of the decimal fraction part takes the precision of the more precise part.
7. SUM$ truncates leading and trailing zeros.

Example

```
40  DECLARE STRING A, B, total
    A = "46"
    B = "87"
    total = SUM$(A,B)
    PRINT total
```

The output is:

```
133
```

SWAP%

The SWAP% function transposes a WORD integer's bytes.

Format

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

Syntax Rules

None.

Remarks

1. SWAP% is a WORD function. BASIC–PLUS–2 evaluates *int-exp* and converts it to the WORD data type, if necessary.
2. BASIC–PLUS–2 transposes the bytes of *int-exp* and returns a WORD integer.

Example

```
30  DECLARE INTEGER word_int
    word_int = SWAP%(23)
    PRINT word_int
```

The output is:

```
5888
```

SYS

The SYS function lets you perform special I/O functions, establish special characteristics for a job, set terminal characteristics, and cause the monitor to execute special operations. The SYS function can be used on RSTS/E systems only.

Format

str-vbl = **SYS**(str-exp)

Syntax Rules

Str-exp is a RSTS/E SYS call code. See the *RSTS/E Programming Manual* for a complete list of SYS call codes and their meanings.

Remarks

None.

Example

```
100     OPEN User_keyboard$ AS FILE #1
        Tmp$ = SYS(CHR$(118)+CHR$(18))    ! Cancel any typeahead from user
        LINUT 'Enter the first line of text';User_input$
```

TAB

When used with the PRINT statement, the TAB function moves the cursor or print mechanism to a specified column.

Format

str-var = **TAB**(int-exp)

Syntax Rules

Int-exp specifies the column number of the cursor or print mechanism.

Remarks

1. The leftmost column position is zero.
2. If *int-exp* is less than the current cursor position, the TAB function has no effect.
3. The TAB function can move the cursor or print mechanism only from the left to the right.
4. You can use more than one TAB function in the same PRINT statement.
5. Use semicolons to separate multiple TAB functions in a single statement. If you use commas, BASIC–PLUS–2 moves to the next print zone before executing the TAB function.
6. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to WORD integer.

Example

```
200  PRINT "Number 1"; TAB(15); "Number 2"; TAB(30); "Number 3"
```

The output is:

```
Number 1      Number 2       Number 3
```

TAN

The TAN function returns the tangent of an angle in radians.

Format

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

Syntax Rules

Real-exp is an angle specified in radians.

Remarks

BASIC–PLUS–2 expects the argument of the TAN function to be a real expression. When the argument is a real expression, BASIC–PLUS–2 returns a value of the same floating-point size. When the argument is not a real expression, BASIC–PLUS–2 converts the argument to the default floating-point size and returns a value of the default floating-point size.

Example

```
10  tangent = TAN(PI/4)
    PRINT tangent
```

The output is:

```
1
```

TIME

The TIME function returns the time of day (in seconds) as a floating-point number. On RSTS/E systems, the TIME function can also return process CPU time and connect time.

Format

real-var = **TIME**(int-exp)

Syntax Rules

None.

Remarks

1. The value returned by the TIME function depends on the value of *int-exp*.
2. If *int-exp* equals zero, TIME returns the number of seconds since midnight.
3. BASIC–PLUS–2 on RSX systems accepts only an argument of zero. All other arguments to the TIME function are undefined and cause BASIC–PLUS–2 to signal "Not implemented" (ERR=250).
4. BASIC–PLUS–2 on RSTS/E systems accepts values 0 through 4 and returns values as shown in Table 4–8. All other arguments to the TIME function are undefined and cause BASIC–PLUS–2 to signal "Not implemented" (ERR=250).
5. The TIME function returns a floating-point value of the default size.
6. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to a WORD integer.

Table 4–8 TIME Function Values

Argument Value	BASIC–PLUS–2 Returns
0	The amount of time elapsed since midnight in seconds
1	The CPU time of the current process in tenths of a second

(continued on next page)

Table 4–8 (Cont.) TIME Function Values

Argument Value	BASIC–PLUS–2 Returns
2	The connect time of the current process in minutes
3	Kilo-core ticks
4	Device time in minutes

Example

```
10  PRINT TIME(0)
```

The output is:

```
49671
```

TIME$

The TIME$ function returns a string displaying the time of day in the form *hh:mm* AM, *hh:mm* PM, or 24-hour clock.

Format

str-var = **TIME$**(int-exp)

Syntax Rules

1. *Str-var* is the time of day.
2. *Int-exp* specifies the number of minutes before midnight.

Remarks

1. If *int-exp* equals zero, TIME$ returns the current time of day.
2. *Int-exp* must be a value between 0 and 1440 or BASIC–PLUS–2 signals an error.
3. The TIME$ function uses either an AM/PM or 24-hour clock. The type of clock is an installation option.
4. On RSTS/E systems, the clock type can also be set by the system manager at system start-up time.
5. If you specify a floating-point expression for *int-exp*, BASIC–PLUS–2 truncates it to a WORD integer.

Example

```
20  DECLARE STRING current_time
    current_time = TIME$(0)
    PRINT current_time
```

The output is:

```
01:51 PM
```

TRM$

The TRM$ function removes all trailing blanks and tabs from a specified string.

Format

str-var = **TRM$**(str-exp)

Syntax Rules

None.

Remarks

The returned *str-var* is identical to *str-exp*, except that it has all the trailing blanks and tabs removed.

Example

```
20  DECLARE STRING old_string, new_string
    old_string = "ABCDEFG        "
    new_string = TRM$(old_string)
    PRINT old_string;"XYZ"
    PRINT new_string;"XYZ"
```

The output is:

```
ABCDEFG        XYZ
ABCDEFGXYZ
```

UNLESS

The UNLESS qualifier modifies a statement. BASIC–PLUS–2 executes the modified statement only if a conditional expression is false.

Format

statement **UNLESS** (cond-exp)

Syntax Rules

The UNLESS statement cannot be used on nonexecutable statements or on statements such as SELECT, IF, and DEF that establish a statement block.

Remarks

BASIC–PLUS–2 executes the statement only if *cond-exp* is false (value zero).

Example

```
10  PRINT "A DOES NOT EQUAL 3" UNLESS A% = 3%
```

UNLOCK

The UNLOCK statement unlocks the current record or bucket locked by the last FIND or GET statement.

Format

UNLOCK #chnl-exp

Syntax Rules

Chnl-exp is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).

Remarks

1. A file must be opened on the specified channel before UNLOCK can execute.
2. The UNLOCK statement applies only to files on disk.
3. If the current record is not locked by a previous GET or FIND statement, the UNLOCK statement has no effect and BASIC–PLUS–2 does not signal an error.
4. The UNLOCK statement does not affect record buffers.
5. After BASIC–PLUS–2 executes the UNLOCK statement, you cannot update or delete the current record.
6. Once the UNLOCK statement executes, the position of the current record pointer is undefined.

Example

```
20  UNLOCK #10%
```

UNTIL

The UNTIL statement marks the beginning of an UNTIL loop or modifies the execution of another statement.

The first format shows the loop definition form; the second format shows the statement modification form.

Format

1. Conditional

UNTIL cond-exp
[statement]...
.
.
.
NEXT

2. Statement Modifier

statement **UNTIL** cond-exp

Syntax Rules

The UNTIL statement cannot be used on nonexecutable statements or on statements such as SELECT, IF, and DEF that establish a statement block.

Remarks

1. The following remarks apply to loop definition:
   - A NEXT statement must end the UNTIL loop.
   - BASIC–PLUS–2 evaluates *cond-exp* before each loop iteration. If the expression is false (value zero), BASIC–PLUS–2 executes the loop. If the expression is true (value nonzero), control passes to the first executable statement after the NEXT statement.
2. In the statement modifier form, BASIC–PLUS–2 executes the statement repeatedly until *cond-exp* is true.

Examples

1.
```
10  !Conditional
    UNTIL A >= 5
          A  = A + .01
          TOTAL = TOTAL + 1
    NEXT
```

2.
```
10  !Statement Modifier
    A = A + 1 UNTIL A >= 200
```

UPDATE

The UPDATE statement replaces a record in a file with a record in the record buffer. The UPDATE statement is valid on RMS sequential, relative, and indexed files.

Format

UPDATE #chnl-exp [, **COUNT** int-exp]

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a number sign (#).
2. *Int-exp* specifies the size of the new record.

Remarks

1. If *int-exp* equals zero, the entire record is written to the file.
2. The file associated with *chnl-exp* must be a disk file opened with ACCESS MODIFY.
3. Each UPDATE statement must be preceded by a successful GET or FIND operation or BASIC–PLUS–2 signals "No current record" (ERR=131). Because FIND locates but does not retrieve records, you must specify a COUNT clause in the UPDATE statement when the preceding operation was a FIND. *Int-exp* must exactly match the size of the old record.
4. After an UPDATE statement executes, there is no current record pointer. The next record pointer is unchanged.
5. The length of the new record must be the same as that of the existing record for all files with fixed-length records and for all sequential files. If the new record is larger than the existing record, BASIC–PLUS–2 truncates the right side of the new record to fit the existing record. If the new record is smaller than the existing record, the file gets corrupted. If you specify a COUNT clause, the *int-exp* must match the size of the existing record.

6. For relative files with variable-length records, the new record can be larger or smaller than the record it replaces.
   - The new record must be smaller than or equal to the maximum record size set with the MAP or RECORDSIZE clause when the file was opened.
   - You must use the COUNT clause to specify the size of the new record if it is different from that of the record last accessed by a GET operation on that channel.
7. For indexed files with variable-length records, the new record can be larger or smaller than the record it replaces.
   - When an indexed file permits duplicate primary keys, an updated record must be the same length as the old one.
   - When the program does not permit duplicate primary keys, the new record can be no longer than the maximum record size specified in the MAP or RECORDSIZE clause when the file was opened and must include at least the primary key field.
8. An indexed file alternate key for the new record can differ from that of the existing record only if the OPEN statement for that file specified CHANGES for the alternate key.
9. On RSTS/E systems, you can use the UPDATE statement on native-mode files opened with mode 1 bit set (UPDATE mode).

Example

```
100 UPDATE #4%, COUNT 32
```

VAL

The VAL function converts a numeric string to a floating-point value.

Format

real-var = **VAL**(str-exp)

Syntax Rules

Str-exp can contain the ASCII digits 0 through 9, uppercase E, a plus sign (+), a minus sign (–), and a period (.).

Remarks

1. The VAL function ignores spaces and tabs.
2. If *str-exp* is null or contains only spaces and tabs, VAL returns a value of zero.
3. The value returned by the VAL function is of the default floating-point size.

Example

```
10  DECLARE REAL real_num
    real_num = VAL("990.32")
    PRINT real_num
```

The output is:

```
990.32
```

VAL%

The VAL% function converts a numeric string to an integer.

Format

int-var = **VAL%**(str-exp)

Syntax Rules

Str-exp can contain the ASCII digits 0 through 9, a plus sign (+), or a minus sign (–).

Remarks

1. The VAL% function ignores spaces and tabs.
2. If *str-exp* is null or contains only spaces and tabs, VAL% returns a value of zero.
3. The value returned by the VAL% function is an integer of the default size.

Example

```
10  DECLARE INTEGER ret_int
    ret_int = VAL%("789")
    PRINT ret_int
```

The output is:

```
789
```

WAIT

The WAIT statement specifies the number of seconds the program waits for terminal input before signaling an error.

Format

WAIT int-exp

Syntax Rules

Int-exp must be a number from 0 through 32767; if it is greater than 32767, BASIC–PLUS–2 assumes a value of 32767.

Remarks

1. The WAIT statement must precede a GET operation to a terminal or an INPUT, INPUT LINE, LINPUT, MAT INPUT, or MAT LINPUT statement; otherwise, it has no effect.
2. *Int-exp* is the number of seconds BASIC–PLUS–2 waits for input before signaling the error "Keyboard wait exhausted" (ERR=15).
3. After BASIC–PLUS–2 executes a WAIT statement, all input statements wait the specified amount of time before BASIC–PLUS–2 signals an error.
4. WAIT 0 disables the WAIT statement.

Example

```
10  DECLARE STRING your_name
    WAIT 60
    INPUT "You have sixty seconds to type your name";your_name
    WAIT 0
```

The output is:

```
You have sixty seconds to type your name?
%Keyboard wait exhausted at line 10 in "WAIT "
```

WHILE

The WHILE statement marks the beginning of a WHILE loop or modifies the execution of another statement.

The first format shows the loop definition form; the second format shows the statement modification form.

Format

1. Conditional

WHILE cond-exp
[statement]...
.
.
.
NEXT

2. Statement Modifier

statement **WHILE** cond-exp

Syntax Rules

1. *Cond-exp* can be any valid relational or logical expression.
2. A NEXT statement must end the WHILE loop.
3. The WHILE statement cannot be used on nonexecutable statements or on statements such as SELECT, IF, and DEF that establish a statement block.

Remarks

1. BASIC–PLUS–2 evaluates *cond-exp* before each WHILE loop iteration. If the expression is true (value nonzero), BASIC–PLUS–2 executes the loop. If the expression is false (value zero), control passes to the first executable statement after the NEXT statement.
2. In a statement modifier WHILE, BASIC–PLUS–2 executes *statement* repeatedly as long as *cond-exp* is true.

Examples

1.
```
10  !Conditional
    WHILE X < 100
        X = X + SQR(X)
    NEXT
```

2.
```
10  !Statement Modifier
    X% = X% + 1% WHILE X% < 100%
```

XLATE$

The XLATE$ function translates one string to another by referencing a table string you supply.

Format

str-var = **XLATE[$]** *(str-exp1, str-exp2)*

Syntax Rules

1. *Str-exp1* is the input string.
2. *Str-exp2* is the table string.

Remarks

1. *Str-exp2* can contain up to 256 ASCII characters, numbered from 0 to 255; the position of each character in the string corresponds to an ASCII value. Because 0 is a valid ASCII value (null), the first position in the table string is position zero.
2. XLATE$ scans *str-exp1* character by character, from left to right. It finds the ASCII value *n* of the first character in *str-exp1* and extracts the character it finds at position *n* in *str-exp2*. XLATE$ then appends the character from *str-exp2* to *str-var*. XLATE$ continues this process, character by character, until the end of *str-exp1* is reached.
3. The output string may be smaller than the input string for the following reasons:
   - XLATE$ does not translate nulls. If the character at position *n* in *str-exp2* is a null, XLATE$ does not append that character to *str-var*.
   - If the ASCII value of the input character is outside the range of positions in *str-exp2*, XLATE$ does not append any character to *str-var*.

Example

```
10  DECLARE STRING A, table, source
    A = "abcdefghijklmnopqrstuvwxyz"
    table = STRING$(65, 0) + A
    LINPUT "Type a string of uppercase letters"; source
    PRINT XLATE$(source, table)
```

The output is:

```
Type a string of uppercase letters? ABCDEFG
abcdefg
```

A

BASIC–PLUS–2 Keywords

This appendix contains a list of the BASIC–PLUS–2 reserved and unreserved keywords as well as a list of keywords that are reserved in VAX BASIC.

A.1 BASIC–PLUS–2 Reserved and Unreserved Keywords

The following is a list of the BASIC–PLUS–2 keywords. Most of the keywords are reserved. The unreserved keywords are marked with a dagger. If you use a reserved keyword as a program variable, you receive an error message. You can use unreserved keywords as program variables.

%ABORT
%CDD
%CROSS
%ELSE
%END
%FROM
%IDENT
%IF
%INCLUDE
%LET
%LIBRARY
%LIST
%NOCROSS
%NOLIST
%PAGE
%PRINT
%SBTTL
%THEN
%TITLE
%VARIANT
ABORT
ABS
ABS%
ACCESS

ACCESS%
ACTIVE
ALIGNED
ALLOW
ALTERNATE
AND
ANY
APPEND
AS
ASC
ASCII
ATN
ATN2
BACK
BASE
BEL
BINARY
BIT
BLOCK
BLOCKSIZE
BS
BUCKETSIZE
BUFFER
BUFSIZ
BY
BYTE
CALL
CASE
CAUSE
CCPOS
CHAIN
CHANGE
CHANGES
CHECKING
CHR$
CLK$
CLOSE
CLUSTERSIZE
COM
COMMON
COMP%
CON
CONNECT

CONSTANT
CONTIGUOUS
CONTINUE
COS
COT
COUNT
CR
CTRLC
CVTF$
CVT$F
CVT$$
CVT$%
CVT%$
DAT
DAT$
DATA
DATE$
DECIMAL
DECLARE
DEF
DEFAULTNAME
DEL
DELETE
DESC
DET
DIF$
DIM
DIMENSION
DOUBLE
DOUBLEBUF
DUPLICATES
DYNAMIC
ECHO
EDIT$
ELSE
END
EQ
EQV
ERL
ERN$
ERR
ERROR
ERT$

ESC
EXIT
EXP
EXPLICIT
EXTEND
EXTENDSIZE
EXTERNAL
EXTRACT
FF
FIELD
FILE
FILESIZE
FILL
FILL$
FILL%
FIND
FIX
FIXED
FLUSH
FNAME$
FNEND
FNEXIT
FOR
FORMAT$
FORTRAN
FREE
FROM
FSP$
FSS$
FUNCTION
FUNCTIONEND
FUNCTIONEXIT
GE
GET
GETRFA
GFLOAT
GO
GOBACK
GOSUB
GOTO
GROUP
GT
HFLOAT

HT
IDN
IF
IFEND
IFMORE
IMAGE
IMP
INACTIVE
INDEXED
INPUT
INSTR
INT
INTEGER
INV
INVALID
ITERATE
KEY
KILL
LEFT
LEFT$
LEN
LET
LF
LINE
LINO
LINPUT
LIST
LOC
LOCKED
LOG
LOG10
LONG
LSET
MAG
MAGTAPE
MAP
MAR
MAR%
MARGIN
MAT
MAX
MID
MID$

MIN
MOD
MOD%
MODE
MODIFY
MOVE
NAME
NEXT
NO †
NOCHANGES
NODATA
NODUPLICATES
NOECHO
NOEXTEND
NOMARGIN
NONE
NOPAGE
NOREWIND
NOSPAN
NOT
NUL$
NUM
NUM$
NUM1$
NUM2
ON
ONECHR
ONERROR
OPEN
OPTION
OR
ORGANIZATION
OTHERWISE
OUTPUT
OVERFLOW
PAGE
PEEK
PI
PLACE$
POS
POS%

† Unreserved keyword.

PPS%
PRIMARY
PRINT
PROD$
PROGRAM
PROMPT †
PUT
QUO$
RAD$
RANDOM
RANDOMIZE
RCTRLC
RCTRLO
READ
REAL
RECORD
RECORDSIZE
RECORDTYPE
RECOUNT
REF
REGARDLESS
RELATIVE
REM
REMAP
RESET
RESTORE
RESUME
RETURN
RFA
RIGHT
RIGHT$
RND
ROUNDING
RSET
SCALE
SCRATCH
SEG$
SELECT
SEQUENTIAL
SET
SETUP

† Unreserved keyword.

SGN
SI
SIN
SINGLE
SIZE
SLEEP
SO
SP
SPACE$
SPAN
SPEC%
SQR
SQRT
STATUS
STEP
STOP
STR$
STREAM
STRING
STRING$
SUB
SUBEND
SUBEXIT
SUBSCRIPT
SUM$
SWAP%
SYS
TAB
TAN
TEMPORARY
TERMINAL
THEN
TIM
TIME
TIME$
TO
TRM$
TRN
TYP
TYPE
TYPE$
UNALIGNED
UNDEFINED

UNLESS
UNLOCK
UNTIL
UPDATE
USAGE$
USEROPEN
USING
USR$
VAL
VAL%
VALUE
VARIABLE
VARIANT
VFC
VIRTUAL
VPS%
VT
WAIT
WHILE
WINDOWSIZE
WORD
WRITE
XLATE
XLATE$
XOR
ZER

A.2 Reserved Keywords In VAX BASIC

The following keywords are reserved keywords in VAX BASIC. If you use these keywords in a program, BASIC–PLUS–2 signals the warning message "Keyword <keyword> is reserved in VAX BASIC." It is recommended that you do not use these keywords if you want your program to be transportable to VAX BASIC.

ACTIVATE
ASK
BASIC
CAUSE
CLEAR
CONTINUE
DEACTIVATE
DEVICE

DRAW
GRAPH
HANDLE
HANDLER
INFORMATIONAL
INITIAL
INKEY$
JSB
LBOUND
NX
NXEQ
OF
OPTIONAL
PICTURE
PLOT
PROGRAM
RETRY
RMSSTATUS
ROTATE
SET
SEVERE
SHEAR
SHIFT
TRANSFORM
UBOUND
WARNING
WHEN

B

Debugger Commands

This section describes the BASIC–PLUS–2 debugger commands. BASIC–PLUS–2 debugger commands help you locate run-time errors and debug program modules. You can use the debugger commands interactively in the BASIC–PLUS–2 environment or from DCL command level. To use debugger commands, you must compile the program using the /DEBUG qualifier. For more information on the BASIC–PLUS–2 debugger see the *BASIC–PLUS–2 User's Guide*.

BREAK

The BREAK command lets you stop program execution at a specified line number, program statement, or at the beginning of CALL statements, user-defined functions, and FOR, UNTIL, and WHILE loops. The program stops before executing the specified breakpoint.

Format

BREAK { **ON** block / [**ON**] stmnt-break, . . . }, . . .

block: { **CALL** / **DEF** / **LOOP** }

stmnt-break: lin-num[.stmnt-num] [;mod-nam]

Syntax Rules

1. The BREAK command with no parameters sets a breakpoint at each line number. The program stops at each line number before executing any statements on the line.
2. *Block* specifies a particular statement or function where execution is to stop. The *ON* keyword must precede the *block*. You can specify only one *block* in each BREAK command.
   - BREAK ON CALL stops execution each time BASIC–PLUS–2 executes a CALL statement to a subprogram. The program stops before any statements in the subprogram execute. If you are executing a task-built program, both the calling and the called program must be compiled with the /DEBUG qualifier; otherwise, the BREAK ON CALL command has no effect. If you are executing a program in the BASIC–PLUS–2 environment, the called program must be compiled with the /DEBUG qualifier.
   - BREAK ON DEF stops execution each time BASIC–PLUS–2 encounters a user-defined function in a module compiled with the /DEBUG qualifier. The statement stops before any statements in the function execute.

- BREAK ON LOOP stops execution each time BASIC–PLUS–2 encounters a FOR, WHILE, or UNTIL statement or modifier. The program breaks each time the program loops back to the loop statement. The program stops after the loop is initialized or incremented, but before any statements in the loop execute.

3. *Stmnt-break* specifies a particular line number or statement where execution is to stop. You can specify a maximum of 10 *stmnt-break* breakpoints. If you specify more than 10 breakpoints, BASIC–PLUS–2 signals the error message "No room."

   - *Lin-num* is a program line number.
   - *Stat-num* is a particular statement associated with the specified line number. The period (.) is required and must immediately follow the line number. No space is allowed between the *lin-num* and *stat-num*. If you include a space between the *lin-num* and *stat-num*, BASIC–PLUS–2 signals an error. The compiler listing lists statements on multi-statement lines by line and statement number.
   - *Mod-nam* specifies that the preceding breakpoint is a breakpoint only in the named program module. The semicolon (;) is required.

Remarks

1. If you specify a *stmnt-break* or *block* that does not exist, no break occurs, BASIC–PLUS–2 does not signal an error or warning, and the program executes normally.
2. To disable program breakpoints, use the UNBREAK command.

Example

```
# BREAK 30.2, 500;PROGB, 2000.3;PROGC
# BREAK ON CALL
# CON

BREAK at line  30 statement 2
#
```

CONTINUE

The CONTINUE command continues program execution.

Format

CONTINUE

Syntax Rules

None.

Remarks

When you have finished entering debugger commands, use the CONTINUE command to resume program execution.

Example

```
# BREAK ON LOOP
# CON
```

CORE

The CORE command returns the number of words currently allocated in memory for your entire task. Use the CORE command in conjunction with the debugger commands FREE, STRING, and I/O BUFFER to determine how memory is allocated for your task.

Format

CORE

Syntax Rules

None.

Remarks

1. The maximum program space allowed on RSX systems is 32K words minus the size of your resident library. The maximum program space allowed on RSTS/E systems is 31K words minus the size of your resident library. See the *BASIC–PLUS–2 User's Guide* for more information on program space and resident libraries.
2. You can use the CORE command only when at least one program module has been compiled with the /DEBUG qualifier. Note, however, that the number returned by the CORE command reflects the memory allocation for the entire task, not just the program module compiled with the /DEBUG qualifier.
3. Knowing the size of core memory can help you control the size of your program and allow you optimize to accordingly. See the *BASIC–PLUS–2 User's Guide* for more information on optimization.

Example

```
# CORE
CORE =  7647

#
```

ERL

The ERL command returns the line number of the line executing when the last error occurred.

Format

ERL

Syntax Rules

None.

Remarks

If no error has occurred, the result returned by the ERL command is undefined.

Example

```
# ERL
ERL =  1050
#
```

ERN

The ERN command returns the 1- to 6-character name of the program module that was executing when the last successfully handled error occurred. If a fatal error was not successfully trapped, control passes from the debugger to command level.

Format

ERN

Syntax Rules

None.

Remarks

1. The ERN command returns a module name only when an error is handled successfully.
2. If no error occurs, the result returned by the ERN command is undefined.

Example

```
# ERN
ERN$ =  CHECKS
#
```

ERR

The ERR command returns the error number of the last error that occurred. See the *BASIC–PLUS–2 User's Guide* for a list of errors and their numbers.

Format

ERR

Syntax Rules

None.

Remarks

If no error occurs, the result returned by the ERR command is undefined.

Example

```
# ERR
ERR =  55
#
```

EXIT

The EXIT command returns control to BASIC–PLUS–2 if you are executing a program in the BASIC–PLUS–2 environment and to command level if you are executing a task-built program.

Format

EXIT

Syntax Rules

None.

Remarks

The EXIT command does not close open channels.

Example

```
# EXIT
```

FREE

The FREE command returns the number of words currently available in memory for I/O and string operations. Use the FREE command in conjunction with the debugger commands CORE, STRING, and I/O BUFFER to determine how memory is allocated for your task.

Format

FREE

Syntax Rules

None.

Remarks

1. When string or I/O operations exceed the amount of available free memory, BASIC–PLUS–2 extends the amount of memory allocated for your task.
2. Knowing the amount of free memory space available can help you control the size of your program and optimize accordingly. See the *BASIC–PLUS–2 User's Guide* for information on optimization.

Example

```
# FREE
FREE =  184

#
```

I/O BUFFER

The I/O BUFFER command returns the number of words currently allocated for I/O buffer space. Use the I/O BUFFER command in conjunction with the debugger commands CORE, STRING, and FREE to determine how memory is allocated for your task.

Format

I/O BUFFER

Syntax Rules

None.

Remarks

Knowing the size of the I/O buffer can help you control the size of your program and optimize accordingly. See the *BASIC–PLUS–2 User's Guide* for information on optimization.

Example

```
# I/O BUFFER
I/O BUFFERS =  1765
#
```

LET

The LET command allows you to change the contents of program variables.

Format

LET vbl1 = { vbl2 / const }

Syntax Rules

1. *Vbl1* specifies a numeric or string variable that you want to change.
2. *Const* or *vbl2* specifies the new value for *vbl1*. You can specify only a constant or variable name. BASIC–PLUS–2 does not allow expressions.
3. You cannot set string variables to a null string with the LET command. If you try to do so, BASIC–PLUS–2 signals the error message "Illegal syntax in LET." However, you can set a variable to the null string in your source program and then assign that variable to another variable with the LET command. For example:

```
1000    NULL$= ""
1010    A$="HELLO"
1020    PRINT A$
```

Then, compile or run the program with the /DEBUG qualifier, establish a breakpoint at line 1020, and set A$ to the null string with the LET command. For example:

```
BREAK at line 1020

# LET A$ = NULL$
```

Remarks

1. If you attempt to create a new variable with the LET command, BASIC–PLUS–2 signals the error "Illegal syntax in LET."
2. Task-built programs must be compiled with the /DEBUG qualifier for the LET command to take effect. BASIC–PLUS–2 signals the error "Illegal syntax in LET" if the program module was not compiled with the /DEBUG qualifier.

3. BASIC–PLUS–2 signals the error "Illegal syntax in LET" when you try to access a variable across modules or in a module not compiled with the /DEBUG qualifier.

Example

```
# LET A%=15%
# LET NAME$="EILEEN"
#
```

PRINT

The PRINT command allows you to display the current contents of program variables.

Format

PRINT vbl

Syntax Rules

1. *Vbl* specifies the numeric or string variable whose contents you want to display.
2. *Vbl* cannot be a constant or expression.

Remarks

1. When executing a task-built program, you can access only those variables contained in program modules that have been compiled with the /DEBUG qualifier.
2. Task-built programs must be compiled with the /DEBUG qualifier for the PRINT command to take effect.
3. BASIC–PLUS–2 signals the error "Illegal syntax in PRINT" when you try to access a variable across modules or in a module not compiled with the /DEBUG qualifier.

Example

```
# PRINT C
# 23

#
```

RECOUNT

The RECOUNT command tells you how many characters, including blanks and terminators, were transferred by the last I/O operation.

Format

RECOUNT

Syntax Rules

None.

Remarks

If you exit from your program without closing open channels or executing the END statement, the Debugger signals the error "End-of-file on device." If you then try to continue program execution by typing the CONTINUE command, the debugger signals the error "Can't CONTINUE or STEP." When you exit from the debugger, files are not closed and data is not transferred. You can remedy this situation by including an error handler which passes control to an END statement. BASIC–PLUS–2 will then close files and transfer data.

Example

```
# RECOUNT
RECOUNT =  19
#
```

REDIRECT

The REDIRECT command allows you to direct all debugging I/O operations to a specified terminal.

Format

REDIRECT term-nam

Syntax Rules

Term-nam specifies the name of an unattached terminal. The terminal name must be an unquoted string that corresponds to an existing terminal, or BASIC–PLUS–2 signals the error "Cannot open device."

Remarks

1. The program executes on the terminal that issued the RUN command. Use another REDIRECT command to direct debugger I/O back to the terminal on which the program is executing.
2. You can use the REDIRECT command only when at least one program module has been compiled with the /DEBUG qualifier.
3. On RSTS/E systems, if the specified terminal is unavailable, the debugger signals the error "Cannot open device." On RSX systems, if the specified terminal is unavailable, the debugger stops executing until the specified terminal is available and does not signal an error.

Example

```
# REDIRECT KB2:
```

STATUS

The STATUS command returns a word-length integer that contains information about the last opened channel.

Format

STATUS

Syntax Rules

None.

Remarks

1. The STATUS function returns a WORD integer.
2. The debugger returns the last STATUS word.
3. On RSTS/E systems, depending on the error, the STATUS function displays a value representing one of the following:
   - The RMS–11 primary status field (STS) or the RMS–11 secondary status field (STV). See the *RSTS/E RMS–11 MACRO Programmer's Guide* for more information.
   - The device characteristics after an RMS–11 OPEN file operation (set by the DEV field of the FAB). See the *RSTS/E RMS–11 MACRO Programmer's Guide* system for more information.
   - For OPEN operations where no error occurs, the status word describes the device characteristics of the FIRQB and FQFLAG field. The first 7 bits describe the device, and bits 7 through 15 describe characteristics of the OPEN statement. See the *BASIC–PLUS Language Manual* or the *RSTS/E System Directives Manual* for more information on STATUS values.
4. On RSX systems, depending on the error, the STATUS function displays a value representing one of the following:
   - The RMS–11 primary status field (STS) or the RMS–11 secondary status field (STV). See the *RSX–11M/M–PLUS RMS–11 Macro Programmer's Guide* for more information.

- The device characteristics after an RMS–11 OPEN file operation (set by the DEV field of the FAB). See the *RSX–11M/M–PLUS RMS–11 Macro Programmer's Guide* for more information.
- The Directive Status Word ($DSW) and its corresponding error code. (This is displayed in the event of a directive error.) See the *RSX–11M/M–PLUS Mini Reference* for the error codes.
- The STATUS field of a QIO. See the *RSX–11M/M–PLUS I/O Drivers Reference Manual* for more information.
- The first word of a GETLUN or GLUN$ directive describing device characteristics. See the *RSX–11M/M–PLUS and Micro/RSX Executive Reference Manual* for more information.

  See Table 4-7 for a list of the values of the STATUS word for OPEN operations where no errors occur.

Example

```
# STATUS
STATUS =  31
#
```

STEP

The STEP command causes the program module to execute statement by statement, stopping after a specified number of statements have executed.

Format

STEP [int-const]

Syntax Rules

1. *Int-const* is the number of statements to be executed before the program stops. It must be a positive integer from 1 through 32767.
2. You must include a space between the STEP command and *int-const* or only one statement executes.
3. STEP with no *int-const* is the same as specifying STEP 1. Only one statement executes and the program then stops.

Remarks

1. When you execute a task-built program, the BASIC–PLUS–2 debugger only counts statements executing in program modules compiled with the /DEBUG qualifier.
2. If a module not compiled with the /DEBUG qualifier executes before a module compiled with the /DEBUG qualifier, the program does not stop until the specified number of statements in the module compiled with /DEBUG have executed.

Example

```
BREAK at line 1050 statement 1
# STEP 2
# CON

STEP at line  1050 statement 3
#
```

STRING

The STRING command tells you how many words are currently allocated for string storage for your entire task. Use the STRING command in conjunction with the debugger commands CORE, I/O BUFFER, and FREE to determine how memory is allocated for your task.

Format

STRING

Syntax Rules

None.

Remarks

Knowing how much memory is allocated to string operations can help you control the size of your program and optimize accordingly. See the *BASIC–PLUS–2 User's Guide* for information on optimization.

Example

```
# STRING
STRING =  2086

#
```

TRACE

The TRACE command displays line numbers and statement numbers as the program executes.

Format

TRACE

Syntax Rules

None.

Remarks

1. The TRACE command does not affect program execution or breakpoints.
2. A task-built program must be compiled with the /DEBUG qualifier for the TRACE command to take effect. When the BASIC–PLUS–2 encounters a program module not compiled with the /DEBUG qualifier, tracing stops. When BASIC–PLUS–2 returns to a module compiled with the /DEBUG qualifier, tracing resumes.
3. Specify the UNTRACE command to disable tracing. See the description of the UNTRACE command for more information.

Example

```
# TRACE
# BREAK 300
# CONT
at line 100 statement 1
at line 100 statement 2
at line 200 statement 1
BREAK at line 300 statement 1
# BREAK 500
# CONT
```

UNBREAK

The UNBREAK command disables previously set breakpoints in programs and subprograms.

Format

UNBREAK { **ON** block / [**ON**] stmnt-break, . . . }, . . .

block: { **CALL** / **DEF** / **LOOP** }

stmnt-break: lin-num[.stmnt-num] [;mod-nam]

Syntax Rules

1. *ON block* specifies a particular statement or function where execution is to stop. You can specify only one *ON block* in each UNBREAK command:
   - UNBREAK ON CALL disables breakpoints that occur when BASIC–PLUS–2 executes a CALL statement to a subprogram.
   - UNBREAK ON DEF disables breakpoints that occur when BASIC–PLUS–2 encounters a user-defined function in a module compiled with the /DEBUG qualifier.
   - UNBREAK ON LOOP disables breakpoints that occur when BASIC–PLUS–2 encounters a FOR, WHILE, or UNTIL statement or modifier.
2. *Stmnt-break* specifies a particular line number or statement where execution is to stop.
   - *Lin-num* specifies a program line number.
   - *Stat-num* specifies a particular statement associated with *lin-num*. The period (.) separator is required and must immediately follow the line number. BASIC–PLUS–2 signals an error if you include a space between *lin-num* and *stat-num*. The cross-reference section of the compiler listing lists statements on multi-statement lines by number.

- *Mod-nam* specifies a program module compiled with the /DEBUG qualifier. When you specify a module name, the specified line number is disabled as a breakpoint only in that program module. If a breakpoint has not been previously set, BASIC–PLUS–2 signals an error.
- If *lin-num* or *stat-num* do not exist, the debugger signals the error "Bad line spec in (UN)BREAK."

3. UNBREAK with no parameters disables all previously specified *stmnt-break* breakpoints. *ON block* breakpoints are not disabled.

Remarks

None.

Example

```
# UNBREAK ON LOOP
# UNBREAK 100;GAMES, 500. 600.2
# CON
```

UNTRACE

The UNTRACE command disables the TRACE command.

Format

UNTRACE

Syntax Rules

None.

Remarks

Enter the UNTRACE command when the program encounters a specified breakpoint and stops executing.

Example

```
# UNTRACE
# CON
```

C

Editing Mode Commands

This appendix describes the BASIC–PLUS–2 editing mode commands. To use the editing mode commands, enter the EDIT command while in the BASIC–PLUS–2 environment. Once you enter the EDIT command, BASIC–PLUS–2 places you in editing mode where you can use the editing mode commands to modify your program. See Chapter 2 for more information on the BASIC–PLUS–2 EDIT command.

DEFINE

The DEFINE editing mode command allows you to enter a macro definition. A macro definition consists of a series of editing mode commands in the order in which they are to execute.

Format

DEFINE

Syntax Rules

The DEFINE and EXECUTE editing mode commands are invalid in a macro definition. If you specify these commands in your macro definition, BASIC–PLUS–2 signals an error message.

Remarks

1. To create a macro definition, enter the DEFINE command. BASIC–PLUS–2 displays the DEFINE prompt (–>). Next, enter the editing mode commands in the sequence in which you want them to execute. When you are finished, enter EXIT or CTRL/Z to exit. You can then use the EXECUTE command to execute your macro definition. See the description of the EXECUTE editing mode command for more information.
2. BASIC–PLUS–2 writes the macro definition to a file, so the definition remains in effect until you enter another DEFINE command.

Example

```
* DEFINE

  Enter command sequence:
  ->FIND REM
  ->SUBSTITUTE /REM/!/
  ->EXIT

* EXECUTE
```

EXECUTE

The EXECUTE editing mode command executes the last macro defined by the DEFINE command. You specify the number of times the macro is to execute.

Format

EXECUTE [int-const]

Syntax Rules

Int-const specifies the number of times the macro executes. If you do not specify *int-const*, BASIC–PLUS–2 executes the macro once.

Remarks

An EXECUTE command always executes the last defined macro definition. If no macro definition exists, BASIC–PLUS–2 signals the error "Command sequence has not been defined."

Example

```
* EXECUTE 5
```

EXIT

The EXIT or Ctrl/Z editing mode command marks the end of a DEFINE or INSERT command or exits you from editing mode.

Format

EXIT

Syntax Rules

None.

Remarks

1. If you enter EXIT or Ctrl/Z in response to the editing mode prompt, BASIC–PLUS–2 exits from the editing mode.
2. If you enter EXIT or Ctrl/Z to end a DEFINE or INSERT command, BASIC–PLUS–2 displays the editing mode prompt and you can enter more editing mode commands.

Example

```
* DEFINE

 Enter command sequence
  ->FIND REM
  ->SUBS /REM/!
  ->EXIT

*
```

FIND

The FIND editing mode command searches the current program for a specified string starting at the last edited line and continuing to the end of the program.

Format

FIND [unq-str]

Syntax Rules

1. *Unq-str* specifies the string you want to find.
2. If you do not specify a *unq-str*, the FIND command matches the *unq-str* specified by the last FIND command. If there is no previous FIND command, BASIC–PLUS–2 matches the first character of the last edited line.

Remarks

1. When BASIC–PLUS–2 locates the string, BASIC–PLUS–2 displays the line containing the string and sets it as the last edited line. BASIC–PLUS–2 also displays an informational message.
2. If the string does not exist, the last edited line remains unchanged and BASIC–PLUS–2 displays a message which informs you that the string was not located.
3. The FIND command matches the string exactly as you enter it. If the string is uppercase, BASIC–PLUS–2 searches for matching uppercase characters.

Example

```
* FIND PRIMT
330  PRIMT 'How many receipts do you have';RECEIPTS
 "PRIMT" found on line 330
*
```

INSERT

The INSERT editing mode command allows you to add lines to a program.

Format

INSERT [lin-num]

Syntax Rules

1. *Lin-num* specifies the line number after which you want to insert new program lines.
2. If the line number does not exist, BASIC–PLUS–2 signals an error.
3. If you do not specify a line number, BASIC–PLUS–2 inserts program lines after the last edited line.
4. The first line of the inserted lines must begin with a line number.

Remarks

1. To insert program lines, enter the INSERT command and then enter in the program lines you want to insert. When you are finished inserting lines, enter the EXIT command to return to editing mode. BASIC–PLUS–2 displays the editing mode prompt and you can enter more editing subcommands.
2. If you insert a line number that already exists, BASIC–PLUS–2 replaces the existing line with the code you insert and does not signal a warning.
3. BASIC–PLUS–2 does not perform syntax checking on program lines inserted by the INSERT command.
4. The current edit line does not change. For example, if the current edit line is 10 and you insert lines 20 and 30, line 10 remains the current edit line.

Example

```
* INSERT 30
 Enter lines to be added after line 30
 ->40  INPUT 'More receipts';RECEIPTS$
 ->50  IF RECEIPTS$ = ""
 ->       THEN GOTO 32767
 ->    END IF
 ->    EXIT
*
```

SUBSTITUTE

The SUBSTITUTE editing mode command allows you to substitute one character string for another in the program currently in memory. SUBSTITUTE is the editing mode equivalent of the EDIT command with one exception: you cannot specify a range of lines. The SUBSTITUTE subcommand can replace only one occurrence of the specified search string, while the EDIT command can replace all occurrences in a range of lines, if you so specify.

Format

SUBSTITUTE search-clause [replace-clause]

search-clause: delim unq-str1 delim

replace-clause: [unq-str2] [delim] [int-const]

Syntax Rules

1. *Delim* is a unique delimiter character that marks the beginning and end of the search and replace strings.
   - *Delim* cannot be a character used in the search or replace strings.
   - The beginning and ending *delim* characters must match, or BASIC–PLUS–2 signals an error.
2. *Unq-str1* specifies the string you want to remove or replace. *Unq-str2* specifies the string to be substituted for *unq-str1*.
   - If *unq-str1* is found, BASIC–PLUS–2 replaces it with *unq-str2*.
   - If *unq-str1* is not found, BASIC–PLUS–2 signals an error.
   - If you do not specify *unq-str2*, BASIC–PLUS–2 deletes *unq-str1*.
   - If you do not specify *unq-str1*, BASIC–PLUS–2 replaces the first character of the last edited line with *unq-str2*.
   - The SUBSTITUTE command matches and replaces strings exactly as you enter them. If *unq-str1* is uppercase, BASIC–PLUS–2 searches for an uppercase string. If it is lowercase, BASIC–PLUS–2 searches for a lowercase string.

3. *Int-const* specifies the occurrence of *str-lit1* you want to replace. If you do not specify an *int-const*, BASIC–PLUS–2 replaces the first occurrence of *str-lit1*.
4. If you specify the SUBSTITUTE command without an argument, BASIC–PLUS–2 signals the error "Parameters required."

Remarks

BASIC–PLUS–2 displays the edited line with changes after the SUBSTITUTE command executes.

Example

```
* SUBSTITUTE /A%/ABSOLUTE%/3
```

D

Object Time System Routines

This appendix contains a list of the module names in the BASIC–PLUS–2 Object Time System (OTS) and a brief description of each of their functions.

Table D–1 Control, Matrix, and Miscellaneous Modules

Module Name	Description
$BTDID	Logical functions
$BTDLD	Logical functions—LONGWORD
$CALLR	CALL thread—BY REF
$CALLS	CALL thread—BP2
$CHANG	ASCII string conversions
$CNTRL	GOTO and branch statements
$ERTHR	Error threads
$FUNC1	Function call threads (obsolete)
$FUNC2	General function threads
$FUNC3	DEF* function code thread
$FUNC4	DEF function code thread
$LOADS	Loading of truth values
$MATIC	Matrix inversion—COMMON
$MATID	Matrix inversion—DOUBLE
$MATIF	Matrix inversion—FLOATING
$MTRET	Matrix inversion—RELOAD RETURN
$MATRT	Matrix inversion—root
$MATRX	Matrix threads

(continued on next page)

Table D–1 (Cont.) Control, Matrix, and Miscellaneous Modules

Module Name	Description
$NEGAT	Negate operations
$REDIM	General redimensioning routine
$TESTS	Logical test routines
$RUNDN	Dummy routine to fill $ICIO0

Table D–2 Array Threads

Module Name	Description
A1I$	1-dimensional INTEGER array address with bounds checking
A1D$	1-dimensional DOUBLE array address with bounds checking
A1F$	1-dimensional FLOATING array address with bounds checking
A2I$	2-dimensional INTEGER array address with bounds checking
A2D$	2-dimensional DOUBLE array address with bounds checking
A2F$	2-dimensional FLOATING array address with bounds checking
B1I$	1-dimensional INTEGER array address without bounds checking
B1D$	1-dimensional DOUBLE array address without bounds checking
B1F$	1-dimensional FLOATING array address without bounds checking
B2I$	2-dimensional INTEGER array address without bounds checking
B2D$	2-dimensional DOUBLE array address without bounds checking
B2F$	2-dimensional FLOATING array address without bounds checking
V1I$	1-dimensional INTEGER array value with bounds checking
V1D$	1-dimensional DOUBLE array value with bounds checking
V1F$	1-dimensional FLOATING array value with bounds checking
V2I$	2-dimensional INTEGER array value with bounds checking
V2D$	2-dimensional DOUBLE array value with bounds checking
V2F$	2-dimensional FLOATING array value with bounds checking
W1I$	1-dimensional INTEGER array value without bounds checking
W1D$	1-dimensional DOUBLE array value without bounds checking

(continued on next page)

Table D–2 (Cont.) Array Threads

Module Name	Description
W1F$	1-dimensional FLOATING array value without bounds checking
W2I$	2-dimensional INTEGER array value without bounds checking
W2D$	2-dimensional DOUBLE array value without bounds checking
W2F$	2-dimensional FLOATING array value without bounds checking

Table D–3 String Modules

Module Name	Description
$PROCT	CTRL/C protection routine
$SSCST	Subscript routines for common string arrays
$SSOFF	Find offset into array
$SSPTR	Subscript pointer mode threads
$SSRDM	Redimensioning thread
$SSSUB	Subscript routines for normal arrays
$SSVIR	Subscripting for virtual arrays
$STCFS	String/numeric conversion functions
$STCMP	String comparison routines
$STCOS	String concatenation routines
$STCVT	String conversion function
$STFNS	String functions
$STFN1	String functions (continued)
$STGTA	String get a mode source routines
$STLSS	String left set routines
$STMOS	String move routines
$STMSC	String routines (miscellaneous)
$STNMD	DOUBLE NUM(1)$ function
$STNMF	SINGLE NUM(1)$ function
$STNML	LONGWORD NUM(1)$ function

(continued on next page)

Table D–3 (Cont.) String Modules

Module Name	Description
$STRSS	String right set routines
$STVLD	VAL function—DOUBLE
$STVLF	VAL function
$STVLI	VAL function
$STVLL	VAL function—LONGWORD
$STXCM	XLT-CVT common routines
$STXLT	String XLATE function
$SWPST	Swap stack locations

Table D–4 Common Math Modules

Module Name	Description
$BMOVS	BYTE moves
$BNEXT	BYTE next threads
$DCMP	DOUBLE comparison routines
$DMOV	DOUBLE moves
$DRAND	RND DOUBLE
$DTAN	DOUBLE tangent
$DTIME	TIME(X%) function—DOUBLE
$DXDD	DOUBLE ** DOUBLE
$ECDF	Approximate FLOATING/DOUBLE comparison
$ECONV	FLOATING formatted conversion
$ECPY	DOUBLE/SINGLE copy threads
$EMISC	ABSOLUTE VALUE/SIGN FUNCTIONS
$FCMP	FLOATING comparisons
$FMOV	FLOATING moves
$FRAND	RND/RANDOMIZE
$FTAN	Tangent FLOATING

(continued on next page)

Table D–4 (Cont.) Common Math Modules

Module Name	Description
$FXFF	SINGLE ** SINGLE
$JADDS	INTEGER addition
$JCMPS	INTEGER comparison
$JCOMP	INTEGER complements
$JCONV	INTEGER conversion routines
$JDIVS	INTEGER division
$JMISC	ABS function
$JMOVS	Simple INTEGER moves
$JMUL	INTEGER multiplication
$JNCR	Increments and decrements
$JNEXT	INTEGER next threads
$JPADD	INTEGER addition with arguments
$JPCMP	INTEGER argument comparison
$JPMOV	INTEGER argument movement
$JPSUB	INTEGER subtraction with arguments
$JSUBS	INTEGER subtraction
$JXII	INTEGER ** INTEGER
$LADDS	LONGWORD addition
$LCMP	LONGWORD comparison
$LCOMP	LONGWORD complement
$LCON2	LONGWORD and INTEGER conversion
$LCONV	LONG INTEGER conversion routines
$LDIV	LONGWORD division
$LXLL	LONGWORD exponentiation routine
$LMISC	ABS—LONGWORD
$LMUL	LONGWORD multiplication

(continued on next page)

Table D–4 (Cont.) Common Math Modules

Module Name	Description
$LNEXT	LONGWORD next threads
$LSUBS	LONGWORD subtraction
$RCMP	RFA comparison
$RMOV	RFA movement

Table D–5 FPU Math Modules

Module Name	Description
$BFPER	Floating-point setup
$BFPEI	Floating-point error
$DATAN	Double-precision arctangent
$DCON1	DOUBLE-to-INTEGER conversion
$DDIV	Double-floating divide
$DEXP	Double-precision EXP
$DFIX	DOUBLE-to-DOUBLE truncation
$DINT	INTEGER part of DOUBLE
$DLOG	Double-precision LOG
$DMUL	Double-floating multiplication
$DSCAL	SCALE factor preprocessor
$DSIN	Double-precision SIN and COS
$DSQRT	Double-precision square root
$DXDI	DOUBLE ** INTEGER
$FADD	Floating addition
$FADDA	Real addition through address
$FADDM	Real addition to memory
$FADDP	Real addition to pointer
$FATAN	Arctangent
$FCON1	FLOAT-to-INTEGER conversion

(continued on next page)

Table D–5 (Cont.) FPU Math Modules

Module Name	Description
$FDIV	Floating division
$FEXP	Real exponent routine
$FFIX	Real-to-real truncation
$FINT	INTEGER part of real
$FLOG	Single-precision log
$FMUL	FLOATING multiplication
$FNEXT	FLOATING next threads
$FSIN	FLOATING SIN
$FSQRT	FLOAT square root
$FXFI	FLOATING ** INTEGER
$JCON1	INTEGER to FLOAT conversion
$LCON1	LONGWORD conversion
$MATII	Matrix inversion—INTEGER

Table D–6 Common I/O Modules

Module Name	Description
$ICCRL	Shared RMS OPEN code
$ICEOL	End I/O list
$ICEND	Common end
$ICFLD	Field threads
$ICINI	I/O initializations
$ICIO0	AST vector area
$ICMOV	Buffer move threads
$ICPRG	PRINT USING (all types)
$ICRCL	Common close thread
$ICRDD	Input DOUBLE
$ICRDL	DELETE threads

(continued on next page)

Table D–6 (Cont.) Common I/O Modules

Module Name	Description
$ICRED	Common inputs (all types)
$ICRE1	New READ routines
$ICRFG	FIND/GET threads
$ICRKY	FIND/GET key thread
$ICRMP	REMAP move threads
$ICRPT	PUT threads
$ICRSC	SCRATCH thread
$ICRSR	RESTORE thread
$ICRUL	UNLOCK thread
$ICRUN	UPDATE thread
$ICULT	Common utility routines
$ICUL1	Common utility routines
$ICWRT	PRINT (all variables)
$ICWR1	PRINT LONGWORD variables
$IVVIR	Virtual block
RAD50	ASCII string to RAD50 conversion
RQLCB	Core block request and release
SAVRG	Non-volatile registers save or restore
$OVSG1	Overlay SEG 1 routine
$OVSG2	Overlay SEG 2 routine
$OVSG3	Overlay SEG 3 routine
$OVSG4	Overlay SEG 4 routine

Table D–7 RMS I/O Modules

Module Name	Description
$ICFSP	Return file information
$IMALL	Common code for all RMS

(continued on next page)

Table D–7 (Cont.) RMS I/O Modules

Module Name	Description
$IMALQ	RMS allocation routines
$IMCLS	Common CLOSE—RMS
$IMDEL	Common DELETE
$IMFGC	Common for FIND/GET
$IMGSA	RMS allocation routines
$IMOPN	Shared RMS OPEN code
$IMPUT	Common PUT
$IMRES	Common RMS RESTORE
$IMUPD	Common UPDATE
$IROPN	Relative OPEN
$IRROT	Relative root resident module
$ISROT	Sequential root resident module
$ISSCR	Sequential SCRATCH
$IUROT	Undefined root resident module
$IXCLS	Indexed CLOSE
$IXOPN	Indexed OPEN
$IXROT	IDX root resident module
$RMSUP	AST vector area
$CMRMS	AST vector area

Table D–8 RSTS/E-Specific Modules

Module Name	Description
$BERFS	File names for error
$BINIT	Run-time initializations
$BXTRA	Extra routines
$CCTRP	BASIC–PLUS–2 CTRL/C trap
$DEBUG	BASIC–PLUS–2 Debugger

(continued on next page)

Table D–8 (Cont.) RSTS/E-Specific Modules

Module Name	Description
$ERROR	Error handler
$ERROT	Error handler
$FTIME	TIME(X%) function
$ICFNS	Non-FIP CALL functions
$ICFSS	File string scan
$ICIO2	I/O impure area (such as FAB)
$ICROP	Common OPEN thread
$ICTRM	System terminal functions
$IECHN	CHAIN thread
$IEMSC	I/O functions
$IESPX	SPEC function
$IESYS	BASIC–PLUS–2 SYS function
$IEULT	I/O utilities
$IMATR	Read file attributes
$IMERR	RMS error handler
$IMROT	RMS root resident code
$IMULK	RMS UNLOCK (free and release)
$ISCLS	Sequential CLOSE—RMS
$ISOPN	Sequential OPEN
$IVOPN	Virtual OPEN
PATCH	Patch space
$START	String arithmetic functions
$STPDB	Stop processor for debugger
$TIME	TIME function
$CMROT	Compiler root resident module
$RNROT	RUN controller

Table D–9 RSX-Specific Modules

Module Name	Description
$BERFS	File names for error
$BINIT	Run-time initializations
$CCTRP	BASIC–PLUS–2 CTRL/C trap
$DEBUG	Debugger for BASIC–PLUS–2
$ERROR	Error handler
$ERROT	Error handler
$FTIME	TIME(X%) function
$ICFNS	Non-FIP CALL functions
$ICFSS	File string scan
$ICIO2	I/O impure area (such as FAB)
$ICROP	Common OPEN thread
$ICTRM	System terminal functions
$IMATR	Read file attributes
$IMERR	RMS error handler
$IMROT	RMS root resident code
$IMULK	RMS unlock (free and release)
$IQCHN	CHAIN thread
$IQKIL	KILL function
$IQMGT	MAGTAPE functions
$IQMSC	I/O functions
$IQNMA	NAME...AS function
$IQROT	Terminal I/O root
$IQULT	I/O utilities
$ISCLS	SEQUENTIAL close - RMS
$ISOPN	SEQUENTIAL open
$IVOPN	VIRTUAL open
PATCH	Patch space
$START	STRING arithmetic functions

(continued on next page)

Table D–9 (Cont.) RSX-Specific Modules

Module Name	Description
$STPDB	Stop processor for debug
$TIME	TIME function
$IVROT	Virtual root resident module
$CMROT	Compiler root resident module
$RNROT	Run controller

Index

A

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	———	Local Digital subsidiary or approved distributor
Internal[1]	———	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

BASIC–PLUS–2
Reference Manual

AA–JP30B–TK

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page Description

_____ _____

_____ _____

_____ _____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
If Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Information Products
NUO1–1/G10
55 NORTHEASTERN BLVD
NASHUA, NH 03062–9934

Do Not Tear - Fold Here