MCDONNELL DOUGLAS

DATA/BASIC Programming Reference Manual

1360

REALITY®

DATA/BASIC™

Programming Reference Manual

Release

2.3 5.3 6.0

Manual #

87-1360

MCDONNELL DOUGLAS COMPUTER SYSTEMS COMPANY

Post Office Box 19501, Irvine, California 92713
Telephone: 714/250-1000 - TWX: 910-595-2441

Chapter 1
How To Use This Manual

Overview

This manual provides in-depth information on the DATA/BASIC programming language. It is intended for use by programmers familiar with the REALITY operating system.

Definition

This manual describes the DATA/BASIC source language, an extended version of Dartmouth BASIC.

BASIC (Beginners All-purpose Symbolic Instruction Code) is a simple yet versatile programming language suitable for expressing a wide range of problems. Developed at Dartmouth College in 1963, BASIC is a language especially easy for the beginning programmer to master.

Features

DATA/BASIC has the following features:

- Optional statement labels (or statement numbers).
- Statement labels of any length.
- Multiple statements on one line.
- Computed GOTO and GOSUB statements.
- Complex IF statements.
- Multiline IF statements.
- Priority case statement selection.
- String handling with variable length strings up to 31,743 characters in 1Kbyte frame systems (32,243 in other REALITY systems).
- External subroutine calls.
- Direct and indirect calls.
- Magnetic tape input and output.
- Fixed-point scaled arithmetic with up to six decimal digit precision.
- ENGLISH data conversion capabilities.
- REALITY file access and update capabilities.
- File level or item level lock capabilities.
- Pattern matching.
- Dynamic arrays.
- Location of elements in dynamic arrays.
- Math functions.
- DATA/BASIC Symbolic Debugger.
- Ability to PERFORM any TCL expression.

Related Documents

DATA/BASIC interfaces with both the ENGLISH language and the PROC programming language. Frequent references to these languages are made throughout this document. If you need additional information on either of these languages, please refer to the ENGLISH Programming Reference Manual and the PROC Programming Reference Manual.

Abbreviations

The following abbreviations may appear in the text:

VM	Value Mark
AM	Attribute Mark
SVM	Subvalue Mark
M/D	Master Dictionary
DICT	Dictionary File
EOF	End-of-File Mark
BOT	Beginning-of-Tape Mark

Conventions

The following conventions apply to this manual.

Convention	**Meaning**
TEXT	Boldface text represents your input.
UPPER CASE	Characters printed in upper case must be entered exactly as shown.
lower case	Characters or words printed in lower case are parameters that you supply (e.g., when you see file-name, type in the actual name of your file).
<RETURN>	Keys, such as Return and Control, appear in angle brackets, with the name of the appropriate key to press inside.
{ }	Any parameter enclosed in braces is optional.
[ ]	Brackets indicate a choice between two or more parameters. Only one of the parameters in brackets may be selected.
{ }...	If an ellipsis is used in the syntax of a command, it means that the parameter immediately preceding the ellipsis may be repeated any number of times.
...	If an ellipsis is used in an example, it means the line is shown as more than one line in this manual but must be entered as one line.

Conventions (cont'd)

Whenever a term is introduced or defined, it appears in **boldface** type.

Monetary Signs

Throughout this manual, reference is made to the dollar sign ($). In other countries, the dollar sign may be replaced by the monetary sign of the currency of that country and the screen will display the appropriate monetary sign. In such cases, please interpret all references to the dollar sign accordingly.

Debugger

The DATA/BASIC Symbolic Debugger, a unique feature of REALITY, facilitates debugging new DATA/BASIC programs and provides a means for maintaining existing programs.

The DATA/BASIC Symbolic Debugger incorporates the following features:

- Single statement execution.
- Multiple statements execution.
- DATA/BASIC Symbolic Debugger entered upon execution of the 'DEBUG' statement.
- Break execution on a selected statement number.
- Break execution when a variable contains a certain value.
- Display current line number.
- Display variables.
- Display executing program name.
- Display arrays.
- Automatically display variable or array values at execution breaks.
- Change any variable or array value.
- Continue execution after an execution break.
- Proceed through a selected number of execution breaks.
- Resume execution at a particular statement number.
- Inhibit program output.
- Spool program output to printer.
- Output to terminal only.
- Dump contents of all variables and arrays.
- Transfer control to System Debugger.
- End execution of program.
- Log off.

Chapter 2
DATA/BASIC Elements

Overview

This chapter explains some of the elements of the DATA/BASIC language.

Multiple Statements

A DATA/BASIC program consists of a sequence of DATA/BASIC statements terminated by an END statement. More than one statement may appear on the same program line. Multiple statements must be separated by semicolons. For example:

```
X=0; Y=1; GOTO 50
```

Note: It's easier to use the DATA/BASIC Symbolic Debugger if only one statement appears on each line.

Labels

Any DATA/BASIC statement may begin with an optional statement label. A statement label allows that statement to be referenced from other parts of the program.

You can use either numeric or alphanumeric labels for branching purposes. Alphanumeric labels must end with a colon (:). The colon is not used when referencing an alphanumeric label (e.g., GO MAINLOOP). For example:

```
   MAINLOOP:
     INPUT X
     IF NOT(NUM(X)) THEN GOTO MAINLOOP
     ON X GOTO 10,20,30
10 PRINT "FIRST"; GO MAINLOOP
20 PRINT "SECOND"; GO MAINLOOP
30 PRINT "THIRD"; GO MAINLOOP
FINISH: END
```

In this example, MAINLOOP and FINISH are alphanumeric labels, while 10, 20 and 30 are numeric labels.

Blank Spaces and Lines

Unless stated otherwise, blank spaces appearing in the program line (that are not a part of a data item) are ignored. Therefore, you may use blanks freely within a program for the sake of appearance.

Blank Spaces and Lines (Continued)

Blank lines may be inserted between lines of code or comment lines. For example:

```
   REM Program to print the
                             ← ← ← blank line
   REM numbers from 1 to 10.
   I=1
5  PRINT I
   IF I = 10 THEN STOP
                             ← ← ← blank line
   I=I+1
   GOTO 5
   END
```

Comments

You may place comments anywhere in a DATA/BASIC program without affecting program execution.

A comment is specified by the REM statement, an asterisk (*), or an exclamation point (!). (The exclamation point (!) is a special case.) For example, the statements below are all valid comment lines:

```
REM THESE STATEMENTS DO NOT
* AFFECT PROGRAM EXECUTION
X=Y+3; ! ASSIGN SUM OF Y+3 TO VARIABLE X
```

Note: If you use an exclamation point (!) at the beginning of a comment line (or all by itself), a line of asterisks will be printed when the program is BLISTed.

A comment may appear on a line all by itself, or it can appear at the end of a line of code. If this is the case, a semicolon (;) must separate the comment from the statement(s) preceding it. For example:

```
I=I+1; *Increments the counter
```

Note: Do not place an equal sign (=) after the letters REM unless the entire string that follows is enclosed in quotes.

For more information about comment lines, refer to the topic "REM" in Chapter 4, "DATA/BASIC Statements".

Line Continuation

Lines can be continued by inserting an ellipsis (...) at the end of a line break. You can even include comment lines between the parts of a continued line. For example:

```
PRINT 'THIS IS AN EXAMPLE OF ':...;* This is a
          * comment embedded between continued
          lines 'LINE CONTINUATION.'
```

Note: There is no limit to the number of times a line can be continued.

Program Storage

A DATA/BASIC program is stored as a file item and is referenced by its item-id. (The item-id is the name given to it when it is created with the Editor.) An individual line within a DATA/BASIC program is an attribute.

Reserved Words

DATA/BASIC contains several reserved words. These words cannot be used as variable names or as names of subroutines. They are:

ABS	DELETE	LE	RTNLIST
ALPHA	DO	LEN	SEQ
AND	DQUOTE	LN	SETTING
ASCII	EBCDIC	LOCKED	SIN
AT	ELSE	LT	SPACE
BITCHANGE	END	MATCH	SPOOLER
BITCHECK	EQ	MATCHES	SQUOTE
BITLOAD	EXP	MAXIMUM	SQRT
BITRESET	EXTRACT	MINIMUM	STEP
BITSET	FIELD	MNUM	STR
BY	FROM	MOD	SUMMATION
CAPTURING	GE	NE	SYSTEM
CASE	GETMSG	NEXT	TAN
CAT	GO	NOT	THEN
CHANGE	GOSUB	OCONV	TIME
CHECKSUM	GOTO	ON	TIMEDATE
CHAR	GROUP	OR	TO
COL1	GT	PASSLIST	TRIM
COL2	ICONV	PWR	UNASSIGNED
COS	IN	REM	UNTIL
COUNT	INDEX	REPEAT	USING
DATE	INSERT	REPLACE	WHILE
DCOUNT	INT	RND	

VARIABLES AND CONSTANTS

Definition

There are two kinds of data: numeric and string. Numeric data consists of a series of digits and represents an amount. String data consists of a set of characters, such as a name and address. These data types are represented in DATA/BASIC as either variables or constants.

Numeric Constants

A numeric constant can contain up to 15 digits, including fractional digits. With a default precision of 4, the effective range is -14,073,748,835.0000 to 14,073,748,835.0000.

The unary minus sign specifies a negative constant. For example, -3.4, -17000000 and -14.3375 are all negative constants.

String Constants

A string constant is represented by a set of characters enclosed in either single or double quotes. If a single quote mark is to be included as part of the string, then the string must be delimited by double quote marks, and vice versa. For example:

```
"Mr. Wilson's report"

'Test of the "GOTO" statement'
```

A string may contain from 0 to 31,743 characters (in systems with 1Kbyte frame size - 32,243 in other REALITY systems). If only a RETURN is entered in response to an INPUT statement, a null string is created.

Variables

Data can also be represented as variables. A variable may contain either a numeric or string value, which may change dynamically throughout execution of the program.

Variable names consist of an alphabetic letter followed by zero or more letters, numbers, punctuation marks or symbols. (Commas and hyphens are not allowed.) A variable name cannot contain spaces.

A variable name identifies that variable, and the name remains constant throughout program execution. The length of a variable name may be from 1 to 31,767 characters in systems with 1Kbyte frame size (32,267 in other REALITY systems).

Note: DATA/BASIC reserved words should not be used as variable names. (A list of reserved words can be found on page 2-5.)

VARIABLES AND CONSTANTS (Continued)

Examples Examples of valid string constants are:

```
"ABC%123#*4AB"
'1Q2Z....'
"A 'literal' string"
''
```

Examples of valid variable names are:

```
X
B$...$
Data.Length
Z...$
var
```

ASSIGNMENTS

Definition

The simple assignment statement is used to assign a value to a variable.

Syntax

variable = expression

Comments

The **expression** on the right side of the equal sign is evaluated and the result is stored in the **variable** on the left. Expression can be any valid DATA/BASIC expression. For example:

```
ABC = 500
X2 = (ABC+100)/2
```

The first statement assigns the value of 500 to the variable ABC. The second statement assigns the value of 300 to variable X2.

String value may also be assigned. For example:

```
VALUE = "THIS IS A STRING"
SUB = VALUE[6,2]
```

The first statement assigns the string "THIS IS A STRING" to the variable called VALUE. The second statement assigns the substring "IS" to variable SUB. (For more information on substring assignments, refer to the topic "Substring Extraction and Assignment" in this chapter.)

Examples

```
X=5
```

Assigns the value 5 to variable X.

***

```
ST="XXXYZ"
```

Assigns the string "XXXYZ" to variable ST.

***

```
ST1 = ST[4,1]
```

Assigns the substring "Y" to variable ST1.

ARITHMETIC EXPRESSIONS

Definition

Arithmetic expressions consist of numeric values and arithmetic operators. The numeric values may be constants, variables or intrinsic functions.

The simplest arithmetic expression is a single numeric constant, variable or intrinsic function. If the arithmetic operator is not expressed, it is implied to be a unary +. A simple arithmetic expression may combine two operands using an arithmetic operator. More complicated arithmetic expressions are formed by combining simple expressions using arithmetic operators.

Expressions are evaluated by performing the operations specified by each of the arithmetic operators on the adjacent constants, identifiers or intrinsic functions. (Intrinsic functions are discussed in a subsequent chapter.)

Precedence

When more than one operator appears in an expression, operations are performed in the order of their precedence. If two or more operators have the same precedence, the leftmost operation is performed first. (Refer to Figure 2-1.)

Operator	Operation	Precedence
( )	Expression within parameters	1
^	Exponents	2
+ -	Unary plus and minus	3
* /	Multiplication and division	4
+ -	Addition and subtraction	5
	Format Strings	6
:	Concatenation	7
LT,+,...	Relational operators	8
AND,OR	Logical AND or OR	9

Figure 2-1. Operator Precedence

Any subexpression may be enclosed in parenthesis. Within the parenthesis, the rules of precedence apply. However, the parenthesized expression as a whole has the highest precedence and is evaluated first. For example:

```
(10+2)*(3-1) = 12*2 = 24
```

ARITHMETIC EXPRESSIONS (Continued)

Parenthesis may be used anywhere in order to clarify the order of evaluation, even if they do not change the order.

Strings in Arithmetic Expressions

If a non-null string value containing only numeric characters is used in an arithmetic expression, it is treated as a decimal number. For example:

```
123 + "456" = 579
```

If you use a string containing nonnumeric characters in an arithmetic expression, when you run the program the following message will be displayed:

```
[B16] NON-NUMERIC DATA WHEN NUMERIC REQUIRED; ZERO USED!
```

Examples

Expression	Answer
2+6+8/2+6	Evaluates to 18
12/2*3	Evaluates to 18
12/(2*3)	Evaluates to 2
A+75/25	Evaluates to A + 3
-5+2	Evaluates to -3
-(5+2)	Evaluates to -7
8*(-2)	Evaluates to -16
5*"3"	Evaluates to 15
3:2*2	Evaluates to 34

STRING EXPRESSIONS AND CONCATENATION

Definition

A string expression may be any of the following: a string constant, a variable with a string value, a substring or a concatenation of string expressions. String expressions may be combined with arithmetic expressions.

Substrings

A substring is a set of characters which makes up part of a whole string. For example, "SO.", "123" and "ST." are all substrings of the string "1234 SO. ELM ST."

Substrings are specified by a starting character position and a substring length, separated by a comma and enclosed in square brackets. For example, if the current value of variable S is the string "ABCDEFG", then the current value of S[3,2] is the substring "CD", (i.e., the two-character substring starting at character position 3 of string S).

If a negative number is specified as the starting character position, then count backward from the end of the string to find the first character of the substring. For example, if string A = "9382", then A[-2,2] equals the substring "82".

For more information on substrings, refer to the next topic, "Substring Extraction and Assignment".

Concatenation

Strings may be concatenated using the colon (:) or CAT operator. Concatenation of two strings appends the characters of the second operand onto the end of the first. For example, both of the following strings evaluate to the string "An example of concatenation":

```
"An example of " CAT "concatenation."

"An example of ":"concatenation."
```

Multiple concatenation operations are performed from left to right. Subexpressions in parenthesis are evaluated first.

The concatenation operator treats both its operands as string values. For example:

```
56:"ABC"
```

evaluates to 56ABC.

STRING EXPRESSIONS AND CONCATENATION (Continued)

CAUTION

Segment marks (X'FF') are used by DATA/BASIC as string terminators. They cannot be manipulated or concatenated to strings.

Examples

In the following examples:

```
Z = "EXAMPLE"
A = "ABC123".
```

Expression	**Answer**
Z[1,4]	Evaluates to "EXAM"
A CAT Z	Evaluates to "ABC123EXAMPLE"
Z[-2,2]	Evaluates to "LE"
A : Z[1,1]	Evaluates to "ABC123E"
A[6,1]+5	Evaluates to 8
Z CAT " ONE"	Evaluates to "EXAMPLE ONE"

SUBSTRING EXTRACTION

Definition The assignment statement extracts substring values from strings.

Syntax

```
X = string[n,m]

X = string{<attr#{,value#{,subvalue#}}>}{[n,m]}

X = string{(array)}{<attr#{,value#{,subvalue#}}>}
{[n,m]}
```

Comments The value **n** is the starting character position and **m** is the length of the substring.

In the second form, the substring value is extracted from a dynamic array. In the third form, the substring value is extracted from a dimensioned array. In both cases, the value **n** is the starting position within the array, attribute, value or subvalue. For more information about array extraction and assignment, refer to Chapter 3, "DATA/BASIC Arrays".

If you specify a negative number as the starting character position (n), then you count backward from the end of the string to find the first character of the substring. For example, if string A = "3621", then A[-2,2] equals the substring "21".

If you specify -1 as the length of the substring (m), the substring starting at position n and continuing through the end of the string is replaced. This makes it easy to manipulate strings of unknown length.

In the statement:

```
X = VAR[n,m]
```

n and m have the following meanings:

n >= 0	The starting position of the substring, from the left. (0 and 1 both indicate the first character.)
n = 0	Same as the value n = 1.
n < 0	The starting position of the substring, from the right (e.g., -1 is the last character of the string, -2 is second to the last, etc.)

SUBSTRING EXTRACTION (Continued)

m > 0	The number of characters, starting from character n to extract. Characters are extracted from left to right.
m = 0	Returns a null string.
m < 0	The number of the character, counting from the ending position, at which to stop substring extraction. For example: If X = "ABCDEFGH", then Y=X[3,-2] yields the result "CDEFG". Starting with the value of the third character position (C), we extracted the string up to and including the second-to-the-last character (G).

Examples

```
VAR = "123456"
X = VAR[3,2]
```

Extracts the substring in VAR, starting in position 3 with a length of 2, and assigns the resulting value of 34 to X.

***

SUBSTRING ASSIGNMENT

Definition

The assignment statement assigns a value to a substring of a variable.

Syntax

```
X[n,m] = expression

X {<attr#{,value#{,subvalue#}}>}{[n,m]} =
expression

X{(array)}{<attr#{,value#{,subvalue#}}>}{[n,m]} =
expression
```

Comments

In the first form the value **n** is the starting character position and **m** is the length of the substring.

In the second form, **expression** is assigned to a subvalue of a dynamic array and **attr#**, **value#**, and **subvalue#** represent the attribute, value and subvalue, respectively. The value **n** is the starting position within the attribute, value or subvalue.

In the third form, **expression** is assigned to a subvalue of a dimensioned array. **Array** represents an array reference, and **attr#**, **value#**, and **subvalue#** represent the attribute, value and subvalue, respectively. The value **n** is the starting position within the array element, attribute, value or subvalue.

You can specify any of the parameters above or all of them. This discussion concentrates on simple substring extraction and assignment, therefore the emphasis is on the first form above, X[n,m] = expression. For more information about array extraction and assignment, refer to Chapter 3, "DATA/BASIC Arrays".

If you specify a negative number as the starting character position (n), then you count backward from the end of the string to find the first character of the substring. For example, if string A = "3621", then A[-2,2] equals the substring "21".

If you specify -1 as the length of the substring (m), the substring starting at position n and continuing through the end of the string is replaced. This makes it easy to manipulate strings of unknown length.

SUBSTRING ASSIGNMENT (Continued)

In the statement

```
X[n,m] = "ABC"
```

n and m have the following meanings:

n >= 0	The starting position of the substring counting from the left. (0 and 1 both indicate the first character.)
n < 0	The starting character of the substring in X, counting backward from the right.
m > 0	The number of characters to replace in the substring in X, starting with character n and continuing left to right.
m = 0	A substring of zero length is replaced in X, meaning that the new string ("ABC" in the example), is inserted, starting at character position n. If n >= 0, the new string is inserted to the right of n. If n < 0, the new string is inserted to the left of n. X[1,0] concatenates a string to the front of X and X[-1,0] concatenates a string to the end of X.
m < 0	The ending point of the substring, counting from the right side of the string. If m is -1, the substring is replaced, from n to the end of the string, with the expression on the right-hand-side.

Examples

```
X = "ABCDEFGH"
X[-5,-3] = "VWXYZ"
```

Replaces the substring starting 5 character positions from the right and ending 3 positions from the right in string X. The new value of string X is "ABCVWXYZGH".

***

```
A = "99999"
A[1,0] = "DDD"
```

Concatenates the value DDD to the front of string A. The new value of A is "DDD99999".

RELATIONAL EXPRESSIONS

Definition

A relational expression consists of a relational operator applied to a pair of arithmetic or string expressions.

Relational Operators

Symbol	Operation
< or LT	less than
> or GT	greater than
<= or LE	less than or equal to
>= or GE	greater than or equal to
= or EQ	equal to
#, <>, >< or NE	not equal to
MATCH or MATCHES	pattern matching

Comments

A relational operation evaluates to 1 if the relation is true, to 0 if the relation is false.

Relational operators have lower precedence and are therefore evaluated after all arithmetic and string operations have been performed.

Relational expressions can be divided into two types: arithmetic relations and string relations.

Arithmetic Relations

An arithmetic relation is a pair of arithmetic expressions separated by any one of the relational operators. For example:

```
3 > 4       (3 is greater than 4) = false = 0

6 >= 6      (6 is greater than or equal to 6) =
            true = 1

5+1 < 4*2   (5 plus 1 is less than 4 times 2) =
            true = 1.
```

String Relations

A string relation may contain a pair of string expressions, or a string and an arithmetic expression separated by any one of the relational operators. If a relational operator encounters one string operand and one arithmetic operand, it treats them both as strings.

In string relations, characters are compared one at a time from left to right. Characters are evaluated according to their numeric ASCII code equivalents. The string with the higher ASCII code equivalent is considered to be "greater than" the other string. For example:

```
AAB > AAA
```

RELATIONAL EXPRESSIONS (Continued)

This relation evaluates to 1 (true), because the ASCII equivalent of B (66) is greater than the ASCII equivalent of A(65).

If two strings are not the same length, but the shorter string is identical to the beginning of the longer string, the longer string is considered to be "greater" than the shorter string. For example:

```
"STRINGS" GT "STRING"
```

This relation evaluates to 1 (true).

Note: The null string ("") is less than zero.

Examples

Expression	**Answer**
4 < 5	Evaluates to 1 (true)
"D" EQ "A"	Evaluates to 0 (false)
"Q" < 5	Evaluates to 0, because the ASCII equivalent of Q (81) is greater than the ASCII equivalent of 5 (53).
Q EQ 5	Evaluates to 1 if current value of variable Q is 5.
"XXX" LE "XX"	Evaluates to 0.
0 > ""	Evaluates to 1.

PATTERN MATCHING

Definition

Pattern matching compares a string value to a predefined pattern or patterns. Pattern matching is specified by the MATCH or MATCHES relational operator.

Syntax

expression MATCH{ES} expression

Comments

MATCH(ES) compares the string value of the expression to the predefined pattern (which is also a string value). If it matches, it evaluates to 1 (true). If it does not match, it evaluates to 0 (false).

The pattern may consist of any of the following:

- An integer number followed by the letter N, which tests for that number of numeric characters.
- An integer number followed by the letter A, which tests for that number of alphabetic characters.
- An integer number followed by the letter X, which tests for that number of numeric or alphabetic characters.
- A literal string enclosed in quotes, which tests for that literal string of characters.

If the integer number used in the pattern is 0, the relation evaluates to true only if all the characters in the string match the specification letter (N, A or X). For example:

```
X MATCH "0A"
```

This relation evaluates to 1 if the current string value of X consists of alphabetic characters only.

Note: A null string matches 0A, 0N, 0X and "".

Multiple Match Strings

MATCH(ES) can compare a string value to more than one pattern at the same time. Simply separate each string with a value mark (XX'FD'). For example:

```
IF X MATCHES "1N3A]1N3A2N" THEN ...
```

PATTERN MATCHING (Continued)

If X contains either one numeric followed by three alphabetic characters or one numeric followed by three alphabetic characters followed by two more numerics, the statements following the THEN clause will be executed.

Examples

```
Z MATCHES '9N'
```

This example evaluates to 1 (true) if the current value of Z consists of 9 numeric characters; evaluates to 0 otherwise.

***

```
B MATCH '3N"-"2N"-"4N'
```

Evaluates to 1 if B contains three numerics, followed by a hyphen, two numerics, a hyphen, and four numerics; 0 otherwise.

***

```
A MATCHES "0N'.'0N"
```

Evaluates to 1 if the current value of A is any number containing a decimal point, or just a decimal point by itself; evaluates to 0 otherwise.

***

```
"ABC" MATCHES "3N"
```

Evaluates to 0.

***

```
B MATCHES "2A]3N"
```

Evaluates to 1 if the value of B consists of 2 alphabetic characters or 3 numerics; 0 otherwise.

***

```
X MATCH ''
```

Evaluates to 1 if X contains a null string.

LOGICAL EXPRESSIONS

Definition

Logical expressions (also called Boolean expressions) consist of logical operators applied to relational or arithmetic expressions.

Logical Operators

Symbol	**Operation**
AND or &	logical AND
OR or !	logical OR

Comments

Logical operators operate on the true or false results of relational and arithmetic expressions.

Note: Relational expressions are false if they evaluate to 0 and true if they evaluate to 1.

Arithmetic expressions are false if they evaluate to 0 and true if they evaluate to anything other than 0.

Precedence

Logical operators have the lowest precedence and are only evaluated after the other operations have been performed. If two or more operators appear in an expression, the leftmost is performed first.

Operation

The expression a OR b is true (1) if a and/or b is true. It is false (0) only if both a and b are false.

The expression a AND b is true (1) only if both a and b are true. It is false (0) if a and/or b is false.

You may use the NOT intrinsic function in logical expressions to negate (invert) the expression. (Refer to Chapter 5, "DATA/BASIC Intrinsic Functions".)

Examples

```
A=16
X = 1 AND A
```

Evaluates to 1, because the current value of A is nonzero.

***

LOGICAL EXPRESSIONS (Continued)

```
A=4
B=1
J=13
Y=A*2-5>B AND 7>J
```

Evaluates to 0 (false), because the value of 7>J is false. Both expressions would have to be true, in order for the result to be true.

***

```
1 AND (0 OR 1)
```

Evaluates to 1, because the expression within the parenthesis (which is evaluated first) is true and the first value is true.

***

```
X1 AND X2 AND X3
```

Evaluates to 1 if the current value of each variable is nonzero; evaluates to 0 otherwise.

***

```
"XYZ1" MATCHES "4X" AND X
```

Evaluates to 1 if X is nonzero, because the first expression is true.

THEN/ELSE CLAUSE

Definition Any DATA/BASIC statements that require or allow a THEN and/or an ELSE clause have the same syntax.

Syntax

```
THEN statement(s) {ELSE statement(s)}

ELSE statement(s)
```

Comment **Statements** may be any number of valid DATA/BASIC statements, either separated by semicolons or contained on separate lines and followed by an END statement.

Where Used The THEN/ELSE clause is used with the following DATA/BASIC statements:

FIND	LOCATE	READ	READV
FINDSTR	LOCK	READLIST	READVU
GETLIST	MATINPUT USING	READNEXT	REWIND
IF	MATREAD	READT	WEOF
INPUT	MATREADU	READU	WRITET
INPUT USING	OPEN		

The ELSE clause is used with the PROCREAD statement.

Examples

```
IF X=0 ELSE STOP
```

If the value of X is zero, control passes to the next statement; otherwise, the program terminates.

***

```
IF X=0 THEN Y=1;Z=2;ELSE Y=2;Z=2
```

If X equals 0, the values 1 and 2 are assigned to variables Y and Z respectively; otherwise, both Y and Z are given the value 2.

***

```
IF X=0 THEN Y=1;Z=2 ELSE
   IF Z=0 THEN STOP
   Y=10
END
```

If the value of X is 0, then Y and Z are assigned the values of 1 and 2 and control passes to the statement following END. Otherwise, if Z equals 0, the program terminates; if not, Y is assigned a value of 10, and control passes to the statement following END.

FORMAT STRINGS

Definition

Print-list values in PRINT and CRT statements can be formatted using format strings. (Refer to Chapter 4, "DATA/BASIC Statements" for more information on the PRINT and CRT statements.)

Syntax

"{j}{$}{,}{n}{field}"

Parameters

The value **j** is either the letter L or the letter R, specifying left or right justification. (L is the default.)

A dollar sign (**$**) concatenates a dollar sign onto the front of the value.

A comma (**,**) inserts a comma between every three digits to the left of the decimal point.

The value **n** specifies the number of fractional digits to print, in the range 0-4 (or 0-6 if extended precision is used).

Field specifies the width of the field within which the value is to be left- or right-justified. Field can be padded out with spaces, zeros or asterisks. The following formats are available:

Symbol	**Meaning**
###.. or #r	Fill with spaces
%r	Fill with zeros
*r	Fill with asterisks

#r

The number of #'s present or the value of integer r determines the number of character positions in the field. For example:

```
X=4; Y=1
PRINT Y:X "R##########"
```

This example prints the following values:

```
1        4
```

Note: The above example also could have been written as PRINT Y:X "R#10".

If the number of #'s or the value r is less than the number of characters to be printed, the leftmost characters in a right-justified field or the rightmost characters in a left-justified field will be truncated. No indication of truncation is provided.

FORMAT STRINGS (Continued)

%r

Zero fill is specified by %r. For example:

```
PRINT 12345.678 'R2%10'
```

This example prints the value:

```
0012345.68
```

*r

To fill the field with asterisks, specify *r as the field parameter. For example:

```
X=4 "L*5"
PRINT X
```

This example prints the following:

```
4****
```

Note: A format string can be assigned to a variable.

Multiple Format Strings

Multiple format strings may be specified contiguously and will operate from left to right. For example:

```
PRINT X*Y '2' 'L#20'
```

This example first rounds the result of X*Y to 2 decimal places and prints that result left-justified in a field of 20 blanks.

Multiple Formatted Values

You can format more than one value in a PRINT statement. Simply separate the values with a colon (:) and format them individually. For example:

```
X = 2; Y = 1
PRINT X "L$######" : Y "L$%5"
```

This example prints the following:

```
$2    $1000
```

FORMAT STRINGS (Continued)

Examples

In the following examples, b represents a blank space.

Print statement	Value of A	Actual Output
PRINT A "L#########"	2	2bbbbbbbb
PRINT A "R#########"	2	bbbbbbbb2
PRINT A "L******"	16	16****
PRINT A "R$,2#########"	10.2	bbb$10.20
PRINT A "R$,2#9"	1000	$1,000.00
PRINT A "R,2#########"	1000	b1,000.00
PRINT A "R2#9"	1000	bb1000.00
PRINT A "R%10"	4.3	00000004.3
PRINT A "R%%%%%"	56	00056
PRINT A*2 "L2#########"	1000	2000.00bb
PRINT A "L,0######"	2300.5	2,301
PRINT A+1 "R1#8"	7.9	bbbbb8.9
PRINT A*6 "2" "R%5"	1111	66.00
PRINT A "$2"	0.1234	$0.12
PRINT A "L$#######"	0.1234	$0.1234
PRINT A "L$#3"	0.1234	$0.
PRINT A "R$,#######"	0.1234	$0.1234
PRINT A "R$,#######"	1234	b$1,234
PRINT A "#####"	999	999bb
PRINT A "R4####"	10.3333	3333
PRINT A "R$,#10"	10000000	10,000,000
PRINT A-7 "L,#9"	1234567	1,234,560
PRINT A "L4####"	12.3456	12.3
PRINT A "2#5"	.1277	0.13bb
PRINT A "R2#10"	-.559	bbbbbb-.56

CONVERSIONS

Definition

The same input and output conversions performed in ENGLISH can be performed in DATA/BASIC. Input and output conversions are performed by the ICONV and OCONV intrinsic functions. The ICONV and OCONV functions are explained in Chapter 5.

Output Conversions

Output conversions, for the most part, convert from some internal format or internally stored data to an output format. Date and time are stored internally as integers and converted to normal date and time strings by an output conversion.

The format strings explained in the previous pages are, themselves, output conversions. Therefore, output conversions that are to be printed (with the PRINT or CRT statements) may be executed as a type of format string. The conversion expression is used as the parameter of the format string. For example:

```
PRINT OCONV(DATE(),'D2/')
```

The above example can be output as a format string:

```
PRINT DATE() 'D2/'
```

You may ask the question, "Why use the OCONV function at all when a format string is so much easier?" Use the OCONV function to perform an output conversion when the result may have to be further used in the program, stored, and/or output in several places. Use a format string when you simply want to print the conversion.

Additional examples of equivalent output conversions are shown below.

Examples

```
ASC = "ABC123"
PRINT OCONV(ASC,"MX")

ASC = "ABC1123"
PRINT ASC "MX"
```

Both of the above examples will convert the ASCII string "ABC123" to hexadecimal and print the value 414243313233.

CONVERSIONS (Continued)

External to Internal Date Conversion

Conversion from external date format to internal date format is an internal conversion normally performed by the ICONV function. However, dates may be stored in a file in external format and you may have the need to convert to internal format as an output function. For that, the **DI** conversion is provided.

DI is a special case of date conversion. It allows you to convert from external to internal format as an output conversion, the inverse of the normal D conversion.

Because DI is used as an output conversion, it may also be specified as a type of DATA/BASIC format string to convert a date to internal format.

All three of the following examples produce the same result:

```
INPUT TODAY
CRT ICONV(TODAY,"D")

INPUT TODAY
CRT OCONV(TODAY,"DI")

INPUT TODAY
CRT TODAY "DI"
```

Chapter 3
DATA/BASIC Arrays

Overview This chapter explains how dimensioned and dynamic arrays operate and provides a comparison of both kinds. It also explains how to extract, replace, insert and delete dynamic array references.

DIMENSIONED ARRAYS

Definition A variable with more than one value associated with it is called an array. Each value is referred to as an element of the array, and the elements are ordered.

Vector Array A is a one-dimensional array (also called a vector).

Table 3-1. One-dimensional Array.

Array A	
3	First element of A has value 3.
8	Second element of A has value 8.
-20.3	Third element of A has value -20.3.
ABC	Fourth element of A has string value ABC.

Matrix Array B is a two-dimensional array (also called a matrix), which has both rows and columns.

Table 3-2. Two-dimensional Array.

Array B				
	Col. 1	Col. 2	Col. 3	Col. 4
Row 1	3	XYZ	A	-8.2
Row 2	8	3.1	500	.333
Row 3	2	-5	Q123	84

DIMENSIONED ARRAYS (Continued)

Accessing Elements

Any array element can be accessed by specifying its position in the array. For example, in the first table, the first element of Array A has a value of 3. Therefore A(1) = 3. The second element of Array A (referred to as A(2)) has the value 8, and so on.

For a two-dimensional array (or matrix), you must specify both the row and column positions of the element. Specify the row first, then the column. For example, in Array B (shown above), element B(1,1) contains the value 3, while element B(2,3) contains the value 500.

DIMENSION Statement

Before an array can be used in a DATA/BASIC program, it must be dimensioned via the DIMENSION or COMMON statement. (For information on how to dimension an array, refer to the topic "DIMENSION" in Chapter 4, "DATA/BASIC Statements".)

DYNAMIC ARRAYS

Definition

A dynamic array consists of one or more attributes separated by attribute marks. An attribute mark (AM) is shown here as an up arrow (^).

Attributes can consist of a number of values, which are separated by value marks. A value mark (VM) is represented by a right bracket (]).

A value may consist of a number of subvalues, separated by subvalue marks. A subvalue mark (SVM) is represented by a backslash (\).

The elements of a dynamic array can be added to or deleted from the dynamic array without recompiling the program, as long as the string does not exceed 31,743 characters (in 1Kbyte/frame systems).

Simple Dynamic Array

The following dynamic array contains four attributes.

```
55^ABCD^732XYZ^100000.33
```

Complex Dynamic Array

The following dynamic array is more complex.

```
Q5^AA^952]ABC]12345^A^B^C]TEST\121\9\99.3]2^555
```

This array contains the following attributes:

```
Q5, AA, 952]ABC]12345, A, B,
C]TEST\121\9\99.3]2, and 555
```

the following values:

```
952, ABC, 12345, C, TEST\121\9\99.3, and 2
```

and the following subvalues:

```
TEST, 121, 9, and 99.3
```

Examples

Array X = 123^456^789]ABC]DEF

In Array X, 123, 456, and 789]ABC]DEF are all attributes. In addition, 789, ABC, and DEF are values.

***

Array B = 1234567890

Array B contains a single attribute.

***

DYNAMIC ARRAYS (Continued)

```
M = Q56^3.22]3.56\88\B]C^99
```

Dynamic array M contains three attributes (Q56, 3.22]3.56\88\B]C, and 99), three values (3.22, 3.56\88\B, and C), and three subvalues (3.56, 88, and B).

***

```
DYNARRAY = A]B]C]D^E]F]G]H^I]J
```

The dynamic array called DYNARRAY contains the attributes A]B]C]D, E]F]G]H, and I]J. It also contains the following values: A, B, C, D, E, F, G, H, I and J.

Note: In the above examples, the attribute mark is represented by an up arrow (^). However, if you were entering any of the examples in a program, they would not compile as shown. You would have to equate a symbol to the attribute mark and concatenate the symbol into the string. For example:

```
EQU AM TO CHAR(254)
DYNARRAY = "A]B]C]D":AM:"E]F]G]H":AM:"I]J"
```

DYNAMIC ARRAYS VS. DIMENSIONED ARRAYS

Introduction

DATA/BASIC lets you manipulate variables in the form of dynamic arrays or as individual array elements. This topic discusses the tradeoffs involved in each format.

Storage

Dynamic arrays are stored in the following way:

```
ABC^DEF]GHI]JKL^MNO\PQR\STU]VWX^YZ
```

Dimensioned arrays are stored as follows:

```
1  ABC
2  DEF]GHI]JKL
3  MNO\PQR\STU]VWX
4  YZ
```

Dynamic Arrays

Dynamic arrays are primarily useful in interfacing to REALITY file items. By specifying a single variable name, you can read or write an entire item, or access individual items easily.

The functions described in the following topic, "Dynamic Array Referencing" scan dynamic arrays, looking for individual elements. This can sometimes be inefficient, depending on the application.

For instance, to extract attribute 10 from an item, DATA/BASIC first scans over attributes 1 through 9. When replacing data in an item, DATA/BASIC passes by all the data preceding the replacement, copies in the new value and then appends all the data that came after the attribute. This continual scanning becomes inefficient when you are accessing a large number of elements or processing a large item.

Dimensioned Arrays

In such cases, it is better to read the item into a dimensioned array (using the MATREAD statement). MATREAD places each attribute of the item into separate variable locations that can be accessed individually, without scanning the item. This is particularly useful if several attributes need to be changed, because you can modify attributes without moving the rest of the item.

DYNAMIC ARRAYS VS. DIMENSIONED ARRAYS (Continued)

You can still use dynamic array references to extract, delete, insert and replace attributes, values and subvalues. You simply specify the array subscript.

For example, if the item is stored as a dynamic array, the expression ITEM<3,2,4> accesses the fourth subvalue in the second value in the third attribute. On the other hand, if the item is stored as a dimensioned array, the expression ITEM(3)<1,2,4> accesses the same element.

This method has its disadvantages too. Because there is a separate descriptor for each cell in the array, it uses more variable descriptor space. It also uses more freespace, because each attribute may have its own buffer in freespace.

In addition, items must be disassembled for a MATREAD and reassembled with a MATWRITE, which takes more time than a simple READ or WRITE.

Rules of Usage

Use the following rules as guidelines in determining whether to use dynamic or dimensioned arrays.

1. Use dynamic arrays only when dealing with data read from or written to REALITY file items. Use standard dimensioned arrays when the need for an internal table arises.

2. If a value is to be used more than once, assign it to a variable instead of performing multiple extractions. Also, if the application is to reference many values/sub-values in an attribute, assign the attribute to a variable and perform the extractions and replacements on this smaller string (using an attribute value of one) rather than continually scanning the entire item.

3. Use dynamic arrays to extract several attributes from the beginning of an item or to replace four or five values in a large item. However, use a dimensioned array to construct new items or to access several different attributes.

4. Use dimensioned arrays only where necessary, because the object code required to reference them is greater than for single variables. However, avoid doubling the source code simply to avoid using subscripted variables.

DYNAMIC ARRAY REFERENCING

Introduction

DATA/BASIC lets you reference any attribute, value or subvalue of a dynamic array directly.

Note: Direct referencing makes the EXTRACT, REPLACE, INSERT and DELETE functions obsolete. However, they are maintained in this document for compatibility.

Syntax

```
expression<attribute {,value {,subvalue}}>
```

Comments

Expression specifies the dynamic array (usually a variable). The following variables are called positional expressions and identify the particular element being referenced.

If both value and subvalue are omitted (or zero), then the entire **attribute** is referenced.

If no subvalue is referenced (or if it is zero), then the specified **value** is referenced.

If a **subvalue** is specified (and it's greater than zero), then it is referenced.

Note: A dynamic array may not be specified in an EQUATE statement.

Examples

```
ITEM<3>
```

References the third attribute in dynamic array ITEM.

***

```
X=2; Y=5
A<X,Y>
```

References the fifth value in the second attribute in dynamic array A.

***

```
DARRAY<I,J,K+2>
```

References the subvalue, specified by K+2, in the J'th value in the I'th attribute in dynamic array DARRAY.

DYNAMIC ARRAY REFERENCING: EXTRACTION

Introduction

An attribute, value or subvalue may be extracted from a dynamic array by specifying a dynamic array reference as an expression in any DATA/BASIC statement.

Rules of Usage

Values other than nonzero positive integers are not valid specifications for dynamic array elements. They are converted according to the rules below:

1. Nonnumeric expressions display a warning message and default to zero.
2. Noninteger expressions truncate the decimal value (e.g., 1.7 and 1.2 become 1).
3. All trailing zero-valued positional expressions (except attribute) are ignored.
4. Any remaining zero-valued positional expressions default to one.
5. Negative values return a null value.
6. If the positional expressions specify a nonexistent position, or if the dynamic array is initially null, a null value is returned.

Examples

```
QTY = ITEM<3>
```

Assigns the value of the third attribute in dynamic array ITEM to variable QTY.

***

```
PRINT A<X,Y>
```

Prints the Y'th value in the X'th attribute in dynamic array A.

***

```
PO.YEAR=ORDER<I,J,3> / 365
```

Assigns the value of the third subvalue in the J'th value in the I'th attribute of dynamic array ORDER, divided by 365, to variable PO.YEAR.

***

DYNAMIC ARRAY REFERENCING: EXTRACTION (Continued)

Note: In the following examples dynamic array S has the value:

```
S="1\2\3]11\22]333^A\B\C]AA"                    *
```

Original Expression	Intermediate Expression	Returns
S<0,0,0>	S<0> (R3)	
	S<1> (R4)	1\2\3]11\22]333
S<0,0,2>	S<1,1,2> (R4)	2
S<0,2,0>	S<0,2> (R3)	
	S<1,2> (R4)	11\22
S<2,0,0>	S<2> (R3)	A\B\C]AA
S<"ABC">	S<0> (R1)	Error Message
	S<1> (R4)	1\2\3]11\22]333
S<1.9>	S<1> (R2)	1\2\3]11\22]333
S<1,-1>	null (R5)	null
S<5>	null (R6)	null

* **NOTE:**

In the above example the attribute mark is represented by an up arrow (^). However, if you were entering the example in a program, it would not compile as shown. You would have to equate a symbol to the attribute mark and concatenate the symbol into the string. For example:

```
EQU AM TO CHAR(254)
S = "1\2\3]11\22]333":AM:"A\B\C]AA"
```

DYNAMIC ARRAY REFERENCING: REPLACEMENT

Introduction

An attribute, value or subvalue may be replaced in a dynamic array by specifying a dynamic array reference as the object (left-side) of an assignment statement.

Rules of Usage

Values other than nonzero positive integers are not valid specifications for dynamic array elements. They are converted according to the rules below:

1. Nonnumeric expressions display a warning message and default to zero.
2. Noninteger expressions truncate the decimal value (e.g., 1.7 and 1.2 become 1).
3. All trailing zero-valued positional expressions (except attribute) are ignored.
4. Any remaining zero-valued positional expressions default to one.
5. The first occurrence of a negative value remains negative and subsequent negative values become one. (See rule 6.)
6. The single remaining negative value creates a new position according to the following scheme:

   If the dynamic array is not null, then the replacement value (preceded by a delimiter) is added as a new attribute, value or subvalue (depending on which position was negative) at the end of the existing specified item, attribute, or value respectively. The remaining positional expressions (if any) are treated as usual. However, if the dynamic array is initially null, the negative value is converted to 1 and Rule 7 is applied.

7. If the positional expressions specify a nonexistent position, or if the dynamic array is initially null, then nulls are created where needed to put the replacement value in the indicated position. A trailing delimiter is not added.

DYNAMIC ARRAY REFERENCING: REPLACEMENT (Continued)

Examples

```
A<X,Y>=PRICE
```

Replaces the Y'th value in the X'th attribute in dynamic array A with the current value of PRICE.

***

```
ORDER<I,J,3> = YR*365
```

Replaces the third subvalue in the J'th value in the I'th attribute in dynamic array ORDER with the value of YR multiplied by 365.

***

```
IF PART<1,X+2> = 0 THEN PART<1,X+2> = 1
```

If the value represented by X+2 in attribute 1 of dynamic array PART is zero it is replaced with 1.

***

Note: In the following examples, N, S and X have the following values, and (Rn) represents the corresponding rule that applies.

```
N="" (null string)
S="1\2\3]11\22]333^A\B\C]AA"    *
X="XXX"
```

Original Expression	Intermediate Expression	Resulting String in First Variable
S<2,0>=X	S<2> (R3)	1\2\3]11\22]333^XXX
S<0,2,0>=X	S<0,2> (R3) S<1,2> (R4)	1\2\3]XXX]333^A\B\C]AA
S<1,1.6>=X	S<1,1> (R2)	XXX]11\22]333^A\B\C]AA
S<"ABC">=X	S<0> (R1) S<1> (R4)	Error Message XXX^A\B\C]AA
S<-5>=X	S<-5> (R6)	1\2\3]11\22]333^A\B\C]AA^XXX

S<-5,3,-2>=X	S<-5,3,1> (R5)
	S<-5,3,1> (R6)
	S<-5,3,1> (R7)
		1\2\3]11\22]333^A\B\C]AA^]]XXX

* See note on page 3-11.

DYNAMIC ARRAY REFERENCING: REPLACEMENT (Continued)

Note: In the following examples, N, S and X have the following values, and (Rn) represents the corresponding rule that applies.

```
N="" (null string)
S="1\2\3]11\22]333^A\B\C]AA"    *
X="XXX"
```

Original Expression	**Intermediate Expression**	**Resulting String in First Variable**
S<0,-5,0>=X	S<0,-5> (R3)	
	S<1,-5> (R4)	
	S<1,-5> (R6)	1\2\3]11\22]333]XXX^A\B\C]AA
S<-2,-2,3>=X	S<-2,1,3> (R5)	
	S<-2,1,3> (R6)	
	S<-2,1,3> (R7)	1\2\3]11\22]333^A\B\C]AA^\\XXX
S<-2,3,2>=X	S<-2,3,2> (R6)	
	S<-2,3,2> (R7)	1\2\3]11\22]333^A\B\C]AA^]]\XXX
N<-2,-2,3>=X	N<-2,1,3> (R5)	\\XXX

* See note on page 3-11.

DYNAMIC ARRAY REFERENCING: INSERTION AND DELETION

Insertion

An attribute, value or a subvalue may be inserted into a dynamic array by specifying a dynamic array reference as part of the DATA/BASIC INS statement.

The INS statement is explained in detail in Chapter 4, "DATA/BASIC Statements".

Deletion

An attribute, value or a subvalue may be deleted from a dynamic array by specifying a dynamic array reference as part of the DATA/BASIC DEL statement.

The DEL statement is explained in detail in Chapter 4, "DATA/BASIC Statements".

Overview

This chapter contains a brief summary of the DATA/BASIC statements, organized by function. Following the summary is a complete description of each statement in alphabetical order.

General Statements	COMMON	Passes values between programs and controls allocation of variable storage space.
	DIMENSION	Dimensions arrays.
	EQUATE	Declares a symbol to be equivalent to a variable or literal.
	INCLUDE	Stores large or commonly used sections of code outside the source code item.
	PRECISION	Sets the degree of precision to which all values are calculated.
	REM	Indicates a comment.
	SHARE	Allows multiple programs to share a single copy of constant data.
Assignment Statements	CLEAR	Initializes all program variables to zero.
	MAT	Assigns values to each element in an array.
Branching	CASE	Allows conditional selection of a sequence of statements.
	IF	Allows for conditional execution of a sequence of DATA/BASIC statements.
	GO (GOTO)	Unconditionally transfers program control to another statement in the program.
	ON GOTO	Transfers program control to a statement specified by the current value of the given expression.
Looping	FOR	Begins a loop that is subsequently terminated by a NEXT statement.
	LOOP	Constructs program loops, using WHILE and UNTIL conditions.
	NEXT	Increments specified variable in a FOR/NEXT loop.

Subroutine Branching	CALL	Transfers control to an external subroutine.
	GOSUB	Transfers control to the subroutine beginning with the specified label.
	ON GOSUB	Transfers control to an internal subroutine determined by the current value of a given expression.
	RETURN	Transfers control from a subroutine back to the main program.
	SUBROUTINE	Identifies a DATA/BASIC program as an external subroutine called by another DATA/BASIC program.
Interprogram Transfers	CHAIN	Allows a DATA/BASIC program to exit to TCL or to pass values to separately compiled programs.
	ENTER	Transfers control to a cataloged program.
Program Termination	ABORT	Halts program execution, prints optional message, and terminates driving PROC.
	END	Signifies the physical end of a program.
	STOP	Halts program execution.
Miscellaneous Control Statements	ASSIGN	Modifies system elements retrieved using the SYSTEM function.
	BREAK	Enables/disables the BREAK key.
	DEBUG	Passes control to the DATA/BASIC symbolic debugger.
	ECHO	Controls echoing of input characters.
	NULL	Specifies no operation.
	PRINTERR	Prints error messages stored in the system ERRMSG file or in a user-specified file.
	RQM	Causes a program to sleep for a specified period of time.

	SLEEP	Same as RQM.
Output Statements	CRT	Outputs data to the CRT.
	FOOTING	Causes the current output device to page and prints the specified text at the bottom of the page.
	HEADING	Causes the current output device to page and prints the specified text at the top of the page.
	PAGE	Advances the current output device to the next page and prints the heading/footing at the top/bottom of the page.
	PRINT	Outputs data to the device selected by the PRINTER statement.
	PRINTER	Selects either the user's terminal or the system printer for subsequent program output.
Input Statements	DATA	Stores values for use by subsequent requests for terminal input.
	INPUT	Prompts the user for input.
	GROUPSTORE	Inserts a string of elements into another string.
	PROMPT	Selects the character used to prompt for user input.
Accessing SCREENPRO	INPUT USING	Solicits input data (in the form of a dynamic array) from the SCREENPRO Screen Processor.
	MATINPUT USING	Solicits input data (in the form of a dimensioned array) from the SCREENPRO Screen Processor.
Accessing PROC	PROCREAD	Reads data from the PROC primary input buffer.
	PROCWRITE	Writes data to the PROC primary input buffer.
Accessing TCL	PERFORM	Lets you use TCL verbs within a DATA/BASIC program.

File I/O Statements	CLEARFILE	Clears out the data or dictionary section of a specified file.
	DELETE	Deletes a file item.
	DELETELIST	Deletes a previously saved list from the POINTER-FILE.
	GETLIST	Produces a list of item-ids for a subsequent READNEXT statement.
	MATREAD	Reads a file item and assigns each attribute to consecutive elements of a dimensioned array.
	MATWRITE	Writes a dimensioned array to a file item.
	OPEN	Selects a file for subsequent input, output or update.
	READ	Reads a file item and assigns its value, as a dynamic array, to a variable.
	READLIST	Reads a list from the POINTER-FILE and assigns it to a variable for program manipulation.
	READNEXT	Reads the next item-id from the select list.
	READV	Reads an attribute value from an item and assigns its string value to a specified variable.
	SELECT	Builds a list of item-ids for the READNEXT statement.
	WRITE	Updates a file item.
	WRITELIST	Writes a string to the POINTER-FILE as a saved list.
	WRITEV	Updates an attribute value in a file.
Tape I/O Statements	READT	Reads the next record from a magnetic tape unit.
	REWIND	Rewinds the magnetic tape unit to the Beginning-Of-Tape mark (BOT).
	WEOF	Writes an End-Of-File mark (EOF) to tape.

	WRITET	Writes a record to tape.
Setting Locks	LOCK	Sets an execution lock, so multiple programs cannot update the same file simultaneously.
	MATREADU	Locks a specific item in a file prior to updating it.
	MATWRITEU	Writes an array to a file item and, if it was locked, leaves the item locked.
	READU	Locks a specific item in a file prior to updating it.
	READVU	Locks a specific item in a file prior to updating it.
	RELEASE	Unlocks items that have been locked for update.
	UNLOCK	Resets execution locks.
	WRITEU	Updates a file item and leaves the item locked.
	WRITEVU	Updates an attribute value in a file and leaves the item locked.
Dynamic Arrays	DEL	Deletes an attribute, value or subvalue from a dynamic array.
	FIND	Finds the position of a given attribute, value or subvalue in a dynamic array.
	FINDSTR	Locates a substring within a dynamic array element.
	INS	Inserts an attribute, value or subvalue into a dynamic array.
	LOCATE	Finds the position of an attribute, value or subvalue within a dynamic array.
Dimensioned Arrays	MATBUILD	Builds a string variable from a dimensioned array.
	MATPARSE	Assigns the elements of a string variable to the elements of a dimensioned array.

ABORT

Purpose

ABORT halts program execution, prints an optional message, and terminates a driving PROC.

Syntax

```
ABORT {message-id}
ABORT {expression,...}
```

ABORT functions much the same as the STOP statement except it also terminates a driving PROC and prints an optional termination message.

Message-id contains the item-id of the item in the system message file (ERRMSG) containing the message. Message-id must be numeric.

Expression can be a variable, function, arithmetic statement or literal string that can be printed as part of the message.

The format of messages in the ERRMSG file are controlled with the following codes:

Code	**Meaning**
C	Clear screen.
H literal string	Print literal string.
L	Output line feed/return.
L(n)	Output n line feeds/ returns.
E {literal string}	Print message item-id surrounded by brackets, followed by optional literal string.
A	Output next parameter.
A(n)	Output next parameter left-justified in a field of n blanks.
R(n)	Output next parameter right-justified in a field of n blanks.
S(n)	Output n spaces, counting from the beginning of the line.
T	Print system time.
D	Print system date.

Notes: Line feed/carriage returns are not processed automatically, so they must be stated explicitly with the ERRMSG item.

It is not a good practice to use messages supplied with the system, because they may change with different software releases.

ABORT (Continued)

Examples

```
ERR = "CANNOT OPEN FILE"
OPEN "TESTFILE" TO "TEST"...
 ELSE ABORT ERR
```

If TESTFILE cannot be opened, the program displays the message "ERRMSG [CANNOT OPEN FILE]" and terminates.

***

```
IF COUNT < 10 THEN GOTO 100
PRINT "PROGRAM OVER"
ABORT
```

If COUNT = 10, the program terminates with the message "PROGRAM OVER". In this example, STOP would be a better choice of statements than ABORT.

***

Item '300' in 'ERRMSG' file:

```
E PROGRAM TERMINATED.
```

DATA/BASIC example:

```
IF COUNT < 10 THEN GOTO 100
ABORT 300
```

If count = 10, the program terminates with the message "[300] PROGRAM TERMINATED".

ASSIGN

Purpose

ASSIGN changes system elements whose values can be retrieved using the SYSTEM function.

Syntax

ASSIGN value TO SYSTEM(element)

Comments

The system elements which may be changed are listed below.

2	Current page width.
3	Current page length.
5	Current page number.
7	Terminal type.
30	Pagination in effect.
35	Language in use.
37	Thousands separator in use.
38	Decimal separator in use.
39	Money sign in use.
42	Asynchronous Comm Port Status (bit 0 only)

Examples

```
ASSIGN 12 TO SYSTEM(5)
```

Assigns the value 12 to system element 5, the current page number.

***

```
ASSIGN 66 TO SYSTEM(2)
```

Assigns the value 66 to system element 2, the current page width.

***

```
ASSIGN 54 TO SYSTEM(3)
```

Changes the value of the current page length (system element 3) to 54.

***

```
ASSIGN 1 TO SYSTEM(42)
```

Raises DTR.

BREAK

Purpose

The **BREAK** statements enable or disable the BREAK key on the terminal.

Syntax

```
BREAK {KEY} OFF
BREAK {KEY} ON
BREAK expression
```

Comments

BREAK OFF disables the BREAK key on the terminal. This can be used to prevent a user from stopping program execution during a critical process, such as a file update.

BREAK ON restores normal operation to the BREAK key (i.e., pressing BREAK causes a "break" to the DATA/BASIC symbolic debugger). Normal operation is also restored when the program ends.

BREAK expression must evaluate to a numeric. If it evaluates to zero, the BREAK key is disabled. If it evaluates to a nonzero value, the BREAK key is enabled.

The current state of the BREAK key can be determined using the SYSTEM(23) function.

Note: These statements should only be used in a debugged program. If a program gets into an endless loop while the BREAK key is disabled, you can't stop the process on the terminal on which it was started.

If this happens, you can stop the process from another terminal (logged on to an account with SYS2 privileges) with the following TCL verb:

```
ENABLE-BREAK-KEY (n
```

The value **n** is the number of the communication line to which the disabled terminal is attached.

BREAK (Continued)

Examples

```
BREAK KEY OFF
WRITE ITEM ON FILE,A
```

Disables the BREAK key on the terminal while ITEM is being written to a file. This guarantees that FILE will be updated, even if the BREAK key is pressed.

***

```
BREAK ON
```

Re-enables the BREAK key.

***

```
BREAK X+5
```

Enables the BREAK key if X+5 is nonzero, otherwise, BREAK key is disabled.

***

:ENABLE-BREAK-KEY 6 <RETURN>

Enables the BREAK key on a terminal attached to line 6.

CASE

Purpose

The CASE statement allows conditional selection of a sequence of statements.

Syntax

```
BEGIN CASE
    CASE expression
    statement(s)
    CASE expression
    statement(s)
    .
    .
    .
END CASE
```

Comments

If the value of the first **expression** is true (nonzero), then the **statement(s)** that immediately follow are executed, and control passes to the next sequential statement following the entire CASE statement sequence.

If the first expression is false (zero), then control passes to the next test expression, and so on.

Programs containing more END CASE statements than BEGIN CASE statements will not compile successfully.

Note: A test expression of 1 means always true.

Examples

```
BEGIN CASE
    CASE A < 5
    PRINT 'A IS LESS THAN 5'
    CASE A < 10
    PRINT 'A IS GREATER THAN OR EQUAL TO 5 AND...
     LESS THAN 10'
    CASE 1
    PRINT 'A IS GREATER THAN OR EQUAL TO 10'
END CASE
```

If A<5, then the first PRINT statement is executed. If 5<A<10, the second PRINT statement is executed. Otherwise, the third PRINT statement is executed.

***

CASE (Continued)

```
BEGIN CASE
    CASE Y=B
    Y=Y+1
END CASE
```

If Y is equal to B, add 1 to the value Y.

***

```
BEGIN CASE
    CASE A=0; GOTO 10
    CASE A<0; GOTO 20
    CASE 1;   GOTO 30
END CASE
```

Control branches to the statement labeled 10 if A is zero, to 20 is A is negative or to 30 if A is greater than zero.

***

```
BEGIN CASE
  CASE ST MATCHES "1A"
     MAT LET=1
  CASE ST MATCHES "1N"
     SGL=1; A.1(I)=ST
  CASE ST MATCHES "2N"
     DBL=1; A.2(J)=ST
  CASE ST MATCHES "3N"
     GOSUB 103
END CASE
```

If ST is one letter, the value 1 is assigned to all LET elements, and the case ends. If ST is one number, then 1 is assigned to SGL, ST is stored at element A.1(I), and the case ends. If ST is two numbers, 1 is assigned to DBL, ST is stored at A.2(J), and the case ends. If ST is three numbers, subroutine 103 is executed. If none of the cases is true, control passes to the statement following END CASE.

CHAIN

Purpose

The **CHAIN** statement allows a DATA/BASIC program to exit to any TCL command or to pass values to separately compiled programs.

Syntax

CHAIN expression

Comments

Expression may contain any valid verb or PROC name in the users' M/D.

The CHAIN statement allows values to be passed to the specified program via the COMMON statement. Consider the following two programs:

Program ABC in File BP

```
   COMMON A,B(2)
   A=500
   B(1)=1;B(2)=2
   CHAIN "RUN BP XYZ (I)"
   END
```

Program XYZ in File BP

```
   COMMON J(2),K
   PRINT J(1),J(2),K
   END
```

Program ABC causes program XYZ to be executed. The I option in the CHAIN statement specifies that the data area is not to be reinitialized, allowing program ABC to pass the values 500, 1 and 2 to program XYZ. Program XYZ prints the values 500, 1 and 2.

All COMMON variables form a long vector in row major order, and, on a CHAIN, are assigned left to right to the CHAINed program's COMMON variables.

Notes: Control is never returned to the DATA/BASIC program originally executing the CHAIN statement.

You may CHAIN to a program that calls a subroutine, but it is not advisable to CHAIN from a subroutine.

When CHAINING to a PROC, the contents of the primary input buffer are replaced by the value of the CHAIN expression, thereby destroying the current contents of the buffer.

CHAIN (Continued)

Examples

```
CHAIN "RUN FN1 LAX (I)"
```

Executes program LAX in file FN1. The I option specifies that the data area is not reinitialized (i.e., the program executing the CHAIN statement will pass COMMON values to program LAX).

***

```
CHAIN "LISTU"
```

Executes the LISTU SYSPROG PROC.

***

```
X = "LIST INV"
CHAIN X
```

Executes the ENGLISH verb LIST.

CLEAR

Purpose **CLEAR** initializes all program variables to zero.

Syntax CLEAR

Comments The CLEAR statement may be used at the beginning of a program to initialize all possible variables or at any point within a program to reinitialize variables.

You can also reinitialize file variables with CLEAR, but it is a time-consuming process and therefore is not recommended.

Programs should not access a variable until a value has been assigned to it.

Example

```
X = 5
Y = Y+1
.
.
.
CLEAR
```

Reinitializes all variables to zero.

CLEARFILE

Purpose — **CLEARFILE** clears out the dictionary or data section of a specified file.

Syntax — CLEARFILE {file-variable}

Comments — When a CLEARFILE statement is executed, the dictionary or data section of the file which was previously assigned to the specified **file-variable** (via an OPEN statement) is emptied (i.e., all the items in the file are deleted).

If a file-variable is not specified, the internal default file-variable is used (i.e., the file most recently opened without a file-variable).

If the specified file has not been opened prior to the execution of the CLEARFILE statement, the program aborts with an error message.

The DL/ID item is not deleted if the dictionary is being cleared. The user's M/D and the SYSTEM dictionary are protected against being cleared.

Examples

```
OPEN 'DICT' 'INV' TO D.INV ELSE PRINT "CANNOT ...
 OPEN DICT INV";  STOP
OPEN 'AFILE' TO X ELSE PRINT "CANNOT OPEN";  STOP
CLEARFILE D.INV
CLEARFILE X
```

Clears the dictionary section of the file called INV and the data section of AFILE.

***

```
OPEN 'FN1' ELSE PRINT "NO FN1"; STOP
READ I FROM 'I1' ELSE STOP
CLEARFILE
```

Opens the data section of file FN1, reads item I1 and assigns a value to variable I, and clears the data section of FN1.

***

```
OPEN '','ABC' ELSE PRINT 'NO FILE'; STOP
READV Q FROM 'IB3', 5 ELSE STOP
IF Q = 'TEST' THEN CLEARFILE
```

Clears the data section of file ABC if the fifth attribute of the item named IB3 contains the string value 'TEST'.

COMMON

Purpose

The **COMMON** statement is used for passing values between programs. It can also be used to control the order in which space for the storage of variables is allocated.

Syntax

COMMON variable {,variable}...

Comments

Variables may be simple, dimensioned or file variables. Arrays included in a COMMON statement are specified by declaring the dimensions (in parenthesis) immediately following the array name. If it is part of the COMMON statement, the array should not be declared in any DIMENSION statement.

The COMMON statement can be used to pass variables among CHAINed programs, but the I option on the RUN verb must be used to inhibit reinitialization. This guarantees that all COMMON variables refer to the same values in different called programs. There is a one-to-one correspondence between the variables listed in the COMMON statement.

Normally, variables are allocated space in the order in which they appear in the program, with all simple variables allocated space before array variables. COMMON forces variables to be allocated space in the order in which they appear in the statement. COMMON must appear before any of the variables are used in the program.

All other variables in the program which do not appear in a COMMON statement are allocated space in the normal manner.

COMMON variables may not appear as symbols in an EQUATE statement and EQUATE symbols may not appear in COMMON statements. However, you may EQUATE a symbol to a COMMON variable.

Variable names may not be any DATA/BASIC reserved word.

Note: Prior to the 3.0 release, you could dimension simple variables with (1) or (1,1). This method can no longer be used. CHAINed programs must now use the COMMON statement to replace the DIMENSION statements, and (1) or (1,1) should be omitted.

COMMON (Continued)

Use of the COMMON statement in main programs and subroutines is recommended over passing values in argument lists because of considerable speed advantages.

COMMON variables may also be used in external subroutines.

Examples

```
COMMON A,B,C(10)
COMMON X,Y,Z(10)
```

These statements, contained in different programs, cause the variables A and X, B and Y and the arrays C and Z to share the same locations.

***

Item 'PGM1' in File BP

```
COMMON A,B(3)
A=2
FOR I = 1 TO 3
B(I)=I*I
NEXT I
CHAIN "RUN BP PGM2 (I)"
END
```

Item 'PGM2' in File BP

```
COMMON X,Y(3)
FOR I = 1 TO 3
PRINT X,X*Y(I)
NEXT I
END
```

The first program declares variables A and B to be COMMON variables, and dimensions array B to three elements. The CHAIN statement exits to execute PGM2 without reinitialization.

The second program associates X and Y to A and B above and dimensions Y as an array with three elements.

CRT

Purpose The **CRT** statement outputs data to the CRT.

Syntax CRT {print-list}

Comments **Print-list** may consist of a single expression or a series of expressions, separated by commas or colons (denoting output formatting).

The expressions may be any legal DATA/BASIC statement.

The CRT statement is functionally equivalent to the PRINT statement, except that the CRT statement converts all system delimiters to printable characters. The CRT statement is not affected by the PRINTER ON/OFF statements.

Output formating, format strings, direct cursor control, and video effects operate the same with the CRT statement as with the PRINT statement.

Examples

```
CRT 5*(X+Y)
```

Displays the current value of the expression 5*(X+Y).

***

```
CRT
```

Displays a blank line.

***

```
A = 728
B = 4
CRT A/B,A+B,A,B
```

Displays the values 182 (A/B), 732 (A+B), 728 and 4.

DATA

Purpose

The **DATA** statement stores values for use by subsequent requests for terminal input.

Syntax

```
DATA expression{,expression...}
```

Comments

Each **expression** is queued as one line of input. These expressions satisfy subsequent requests for input on a first-in-first-out basis.

The DATA statement stores stacked input for use by subsequent INPUT statements.

The DATA statement can store stacked input for TCL, ENGLISH verbs or PROCs, when used in conjunction with a CHAIN or PERFORM statement. (Refer to the CHAIN statement and the PERFORM statement for further information.)

DATA can also be used to feed to input requests of other programs that are executed via the CHAIN or ENTER statements. For example:

```
   DATA 'RUN BP PROG2', REF.DATE
   CHAIN 'SSELECT INV WITH DATE < "':REF.DATE: ...
    '" BY DATE'
```

This example exits from a DATA/BASIC program, sort selects a file and begins execution of a second program. The DATA statement stacks two values (RUN BP PROG2 and REF.DATE) to be used to feed subsequent requests for terminal input. When the ENGLISH processor has selected the items, the prompt that follows a SELECT statement will receive RUN BP PROG2 as input. The INPUT statement contained in PROG2 will receive the value REF.DATE.

Note: A DATA statement must be processed before the CHAIN statement.

If a variable name appears in a DATA statement, the stacked value is the contents of the variable at the time the DATA statement is executed. For example:

```
   X=3
   DATA X
   X=4
   CHAIN "RUN BP PGM"
```

The stacked value is three, because that is the value of X at the time the DATA statement is executed.

DATA (Continued)

Examples

```
DATA X,Y,3
.
CHAIN "RUN BP CALC"
```

Stacks values X, Y and 3 for subsequent requests for input. If CALC has three input requests, they will be satisfied by the values of X, Y and 3.

***

```
DATA 'SELECT INV WITH DATE = "' :D: '"'
DATA "RUN PROGRAMS ONE", D
.
ENTER SPEC.LIST
```

Stacks the ENGLISH SELECT sentence. Stacks the string RUN PROGRAMS ONE and the value of D. Exits to execute the cataloged program SPEC.LIST.

***

```
NUMBER=12
DATA NUMBER
.
CHAIN "CHECK"
```

Stacks the present value of NUMBER for subsequent input request. Exits to run the PROC CHECK, which uses the value of NUMBER for input.

DEBUG

Purpose

DEBUG passes control to the DATA/BASIC symbolic debugger. (For more information on the debugger, refer to Chapter 7.)

Syntax

```
DEBUG
```

Example

```
.
.
DEBUG
```

Enters the system debugger.

DEL

Purpose

The **DEL** statement deletes an attribute, value or subvalue from a dynamic array using a dynamic array reference. (Use of DEL obsoletes the DELETE intrinsic function.)

Syntax

DEL dynamic-array-reference

Comments

Dynamic-array-reference specifies the dynamic array and the position of the value in the array to be deleted. Whether an attribute, value or subvalue is deleted depends on the value of the third and fourth expressions in the dynamic-array-reference, if present.

The format of dynamic-array-reference is **array<a,v,s>** where **array** is the dynamic array, **a** is the attribute, **v** is the value and **s** is the subvalue.

In deletions, values other than positive integers are illogical dynamic array indices. Invalid and illogical indices are converted as follows:

1. Nonnumeric expressions print a warning message and default to zero.
2. Noninteger expressions truncate the decimal value (i.e., 1.7 and 1.2 become 1).
3. All trailing zero-valued positional expressions (except attr#-expression) are ignored.
4. Any remaining zero-valued positional expressions are treated as ones.
5. Negative values cause the statement to have no effect.
6. If the positional expressions specify a nonexistent position or the dynamic array is initially null, the statement has no effect.
7. If you try to delete the first attribute (value or subvalue) of an item that has no attribute marks, the item is made null instead. The trailing delimiter, if any, is removed.

Examples

```
DEL ITEM<3>
```

Deletes the third attribute in the dynamic array ITEM.

DEL (Continued)

DEL A<X,Y>

Deletes the Y'th value in the X'th attribute in dynamic array A.

***

DEL ORDER<I,J,3>

Deletes the third subvalue in the value specified by J in attribute indicated by I.

***

For the following examples:

S = "1\2\3]11\22]333^XXX^A\B\C]AA"

DEL<1,2.4>

2.4 is truncated to 2 and the result is 1\2\3]333^XXX^A\B\C]AA

***

DEL S<1,2,5>

Specifies a nonexistent position, so the statement is ignored and S remains unchanged.

***

DEL S<2,1>

Because attribute 2 (XXX) has no value or subvalue marks, attribute 2 is made null, and the result is 1\2\3]11\22]333^^A\B\C]AA.

***

DEL S<3,0,0>

The third attribute is ignored, and the result is 1\2\3]11\22]333^XXX.

DELETE

Purpose

The **DELETE** statement deletes a file item.

Syntax

```
DELETE {file-variable,}item-id-expression
```

Comments

Item-id-expression specifies the item to be deleted. **File-variable** specifies the file (previously assigned to file-variable via an OPEN statement) which contains the item to be deleted. If file-variable is not specified, then the internal default file-variable (i.e., the file most recently opened without a file-variable) is used.

If the item-id specified in the DELETE statement does not exist, no action is taken.

If the specified file has not been opened prior to the execution of the DELETE statement, the program aborts with an error message.

Examples

```
DELETE X,"XYZ"
```

Deletes item XYZ in the file opened and assigned to variable X.

***

```
Q="JOB"
DELETE Q
```

Deletes item JOB in a file opened without a file variable.

DELETELIST

Purpose

DELETELIST deletes a previously saved list from the POINTER-FILE.

Syntax

```
DELETELIST item-id {account-name}
```

Comments

Item-id is the item-id of the list to be deleted.

The list may have been saved by a SAVE-LIST, EDIT-LIST or FORM-LIST command executed at TCL, or from a WRITELIST statement executed in a DATA/BASIC program.

To delete lists accessed from another account, include the **account-name** after the item-id.

Examples

```
DELETELIST 'ITEMS'
```

Deletes the list ITEMS (saved from your account) from the POINTER-FILE.

***

```
NAME = "ITEMS SYSPROG"
DELETELIST NAME
```

Deletes the list ITEMS (saved from the SYSPROG account) from the POINTER-FILE.

DIM

Purpose

The **DIM** (or **DIMENSION**) statement dimensions arrays so they can be used in a DATA/BASIC program.

Syntax

```
DIM{ENSION} variable(dimensions)
 {,variable(dimensions)...}
```

Comments

A **variable** represents the name of the array, while the **dimensions** indicate the size of the array.

An array can be one-dimensional, also called a vector, or two-dimensional, also called a matrix. (For more information on arrays, refer to Chapter 3, "DATA/BASIC Arrays".)

The maximum dimensions of an array must be specified with a DIMENSION or COMMON statement. The dimensions are declared with constant whole numbers greater than one, separated by commas. DIMENSION (or COMMON) statements must precede any array references and are usually placed at the beginning of the program.

Note: Arrays only need to be dimensioned once throughout the entire program.

Do not use any DATA/BASIC reserved words or intrinsic functions as array names.

Several arrays can be dimensioned with a single DIMENSION statement. For example:

```
DIMENSION A1(10,5), X(50)
```

This example declares Array A1 as a 10 by 5 matrix and declares Array X as a 50-element vector.

Examples

```
DIMENSION MATRIX(10,12)
```

Specifies a 10 by 12 matrix called MATRIX.

***

```
DIM Q(10), R(10), S(10)
```

Specifies 3 vectors named Q, R and S, each of which contains 10 elements.

ECHO

Purpose

The **ECHO** statement controls the echoing of input characters.

Syntax

```
ECHO ON
ECHO OFF
ECHO expression
```

Comments

ECHO ON enables the echoing of input characters to the terminal, while **ECHO OFF** suppresses echoing of input characters.

If you use **ECHO expression,** expression must evaluate to a numeric. If it evaluates to zero, echoing is disabled (same as ECHO OFF). If expression evaluates to a nonzero, echoing is enabled (ECHO ON).

The current state of the ECHO feature may be determined using the SYSTEM(24) function.

ECHO replaces the user exit (U80E0) for controlling the echoing of input characters.

Examples

```
ECHO ON
INPUT X
```

Echoes the value of X to the terminal.

***

```
ECHO OFF
INPUT Y
```

Suppresses display of the value of Y on the terminal.

***

```
A = 35
B = 42
ECHO A+B
```

Enables echoing of characters, because the value of expression A+B evaluates to a nonzero.

END

Purpose

The **END** statement specifies the physical end of a DATA/BASIC program.

Syntax

```
END
```

Comments

END must appear as the last statement in a DATA/BASIC program.

END is also used to specify the physical end of sequences of statements within the IF statement and within certain DATA/BASIC I/O statements.

For every multiline THEN and ELSE statement, there must be a corresponding END statement. If there are not enough END statements, the following error displays:

```
[B101] MISSING "END", "NEXT", "WHILE", "UNTIL", "REPEAT" OR
"ELSE"; COMPILATION ABORTED, NO OBJECT CODE PRODUCED.
```

If there are too many END statements, the program may compile successfully, but not all of the program statements may get compiled. You should verify that the line number indicated in the COMPILATION COMPLETE message matches the number of lines in your program.

Examples

```
   A=1
   B=2
   C=A+B
END
```

END signifies the physical end of this program.

***

```
IF A>B THEN
   PRINT "A GT B"
   STOP
END ELSE
   PRINT "B LE A"
END
END
```

The first END statement terminates the THEN clause. The second END statement terminates the ELSE clause. The final END statement terminates the program.

ENTER

Purpose

ENTER lets you transfer control to a cataloged program.

Syntax

```
ENTER item-id
ENTER @variable
```

Comments

Both **item-id** and **variable** represent the name of a cataloged program.

ENTER transfers control to a DATA/BASIC program that has already been compiled and cataloged. The program executing the ENTER statement must be cataloged also.

All variables to be passed between programs must be declared in a COMMON declaration in all programs involved. All other variables are initialized when the program is entered.

You may ENTER a program that calls a subroutine, but you may not ENTER a program from a subroutine or invoke a subroutine with the ENTER statement.

ENTER works faster than the CHAIN statement.

Examples

```
ENTER PGM1
```

Executes cataloged program PGM1. Any COMMON variables are passed to PGM1.

***

```
I=2
PROGRAM="PGM":I
ENTER @PROGRAM
```

Executes PGM2 and passes any COMMON variables to PGM2.

EQU

Purpose

The **EQU** (or **EQUATE**) statement declares a symbol to be equivalent to a variable or literal.

Syntax

```
EQU{ATE} symbol TO expression {,symbol TO
 expression...}
```

Comments

The **symbol** is formed like a variable, but there is no storage allocated for it. Symbol cannot be a DATA/BASIC reserved word.

Expression may be a number, literal string, character, simple variable, array element or the CHAR intrinsic function. If expression is a simple variable, it implies that the two variable names are equivalent and can be used interchangeably.

There is an advantage to equating a symbol name to a number, literal string or character, rather than assigning it to a variable. This way, the immediate value is compiled into the object text, rather than accessing a variable location each time the program is run.

The advantage of equating a symbol name to an array name is that the computation of the array's address is done once at compile time, rather than each time the array element is referenced at run time. Giving names to array elements also makes the program more readable. For example:

```
Example 1                Example 2

EQUATE QTY TO ITEM(3)    VALUE = ITEM(3) * ITEM(4)
EQUATE PRICE TO ITEM(4)  PRINT VALUE
VALUE = QTY * PRICE
PRINT VALUE
```

The first example, though slightly longer, is more readable and operates faster than the second.

The EQUATE statement must appear before the symbol is used in the program.

COMMON variables may not appear as symbols in the EQUATE statement, and EQUATE symbols may not appear in COMMON statements. However, you may equate a symbol to a COMMON variable.

EQU (Continued)

Examples

```
EQUATE X TO Y
```

Equates symbol X to variable Y, so they may be used interchangeably within the program.

***

```
EQUATE PI TO 3.1416
```

Assembles symbol PI as 3.1416 at compile time.

***

```
EQU STARS TO "*****"
```

Equates symbol STARS to a string value at compile time.

***

```
EQUATE AM TO CHAR(254)
```

Equates symbol AM to CHAR(254).

***

```
EQU PART# TO ITEM(3)
```

Equates PART# to an array element.

***

```
COMMON A
EQUATE AA TO A
```

Equates symbol AA to COMMON variable A.

FIND

Purpose

The **FIND** statement finds the position of a given attribute, value or subvalue in a dynamic array.

Syntax

```
FIND element IN dynamic{,occur} SETTING a{,v{,s}}
 THEN/ELSE
```

Comments

Occur specifies the number of the occurrence of the **element** being searched for in the **dynamic** array. If not specified, it defaults to 1. The values **a, v** and **s** specify the attribute, value and subvalue where the element is found.

The FIND statement works the same way as the LOCATE statement, except it does not require a specific format for the dynamic variable.

FIND returns the attribute and, if specified, the value and subvalue in which the element is found.

Examples

```
DYNARR = "A^B^C^D^E^F^G^A^B^C"
FIND "A" IN DYNARR,2 SETTING X ELSE ...
 PRINT "NOT FOUND"; STOP
```

Finds the second occurrence of the element A in dynamic array DYNARR. Sets X to the value indicating the position of that attribute.

***

```
ITEM = "24^34]28]31^29]22]21"
X=29
FIND X IN ITEM,1 SETTING A,B ELSE ...
 PRINT "NOT FOUND"; STOP
PRINT A,B
```

Finds the multivalue equal to 29 in attribute A, value B and prints A and B. In this example, A=3 and B=1.

***

```
ARR = "44^88^6]2]7\3\2^8^99"
FIND 3 IN ARR,1 SETTING A,B,C ELSE ...
 PRINT "CAN'T FIND"; STOP
PRINT A,B,C
```

Finds attribute 3 in attribute 3, value 3, subvalue 2 of dynamic array ARR and prints the values of A, B and C (3, 3 and 2).

FINDSTR

Purpose **FINDSTR** locates a substring within a dynamic array element.

Syntax

```
FINDSTR substr IN dynamic{,occur} SETTING
a{,v,{s}} THEN/ELSE
```

Comments **Substr** is the substring to search for within an element in the dynamic array.

Dynamic is the dynamic array in which to search.

Occur is the occurrence of substring to search for. If not specified, it defaults to 1.

The values **a, v** and **s** are the attribute, value and subvalue where the element is found. If the **ELSE** clause is taken, these variables do not change.

Examples

```
ARRAYX = "ALABAMA^CALIFORNIA^MINNESOTA"   *
FINDSTR "CA" IN ARRAYX,1 SETTING Y ...
 ELSE PRINT "NO 'CA'"; STOP
```

Finds the attribute containing substring value CA in dynamic array ARRAYX. Sets Y to the value indicating the position of that attribute. Prints error message if not found.

***

```
OPEN "FILE1" TO FILE ELSE PRINT "CANNOT OPEN";G 10
READ NUM FROM FIL,"1" ELSE PRINT "CAN'T READ";G 20
FINDSTR 33 IN NUM,1 SETTING K ...
 ELSE PRINT "NOT FOUND"; STOP
PRINT K
```

Searches for substring 33 in NUM, item 1 and prints a number representing the location in array NUM where 33 occurs.

* Attribute marks are shown in example for illustration. Please refer to note on Page 3-6.

FOOTING

Purpose

The **FOOTING** statement causes the current output device to page and prints the specified text at the bottom of the page.

Syntax

FOOTING "expression"

Comments

Expression is the string to be printed at the bottom of each page.

The page size is determined by the most recent TERM command executed at TCL. If the footing is longer than the TERM line length, the footing wraps around to the next line.

Special footing control characters may be used as part of the FOOTING statement. They are:

'C{n}'	Center line (in field of n characters).
'D' or \\	Current date.
'T' or \	Current time and date.
'L' or]	Carriage return and line feed.
'P' or ^	Current page number.
'PP' or ^^	Current page number right-justified in four spaces.
'N'	Inhibits paging.
''	Two consecutive quotes print a single quote.

When FOOTING is used, all terminal output is paged (i.e., a carriage return must be pressed when a full page has been printed), unless the N option is used.

FOOTING statements may be changed or cleared independently, without altering the page number. The first FOOTING statement issued causes a page advance and a new footing, even if the footing is null. A FOOTING statement may be cleared by placing a null string after the statement. Unless both the heading and the footing are cleared, changing an existing heading or footing does not affect the page number.

Example

```
FOOTING "'L''L' TIME & DATE: 'T'"
```

Advances current output device to top-of-page. When page is full, there will be two carriage returns/line feeds followed by the text TIME & DATE:, followed by the current time and date printed at the bottom.

FOR

Purpose

The **FOR** statement begins a loop that is terminated by a NEXT statement. FOR and NEXT loops may be nested, and they can contain WHILE and UNTIL condition clauses.

Syntax

```
FOR variable=expl TO exp2 {STEP exp3} {WHILE exp4}
FOR variable=expl TO exp2 {STEP exp3} {UNTIL exp4}
```

Comments

Variable contains the value to be incremented or decremented. **Expl** and **Exp2** indicate how many times the loop should be executed. **Exp3** specifies the number by which to increment expl, and **exp4** is an additional limiting value.

FOR and NEXT loops that are contained inside other FOR and NEXT loops are called nested loops. For example:

```
FOR I=1 TO 10
  FOR J=1 TO 10
    PRINT B(I,J)
  NEXT J
NEXT I
```

In this example, the inner loop (FOR J=1 TO 10) executes 10 times for each pass through the outer loop (FOR I=1 TO 10). The outer loop is executed 10 times also, so the statement PRINT B(I,J) is actually executed a total of 100 times. Matrix B will be printed in the following order: B(1,1), B(1,2), B(1,3),..., B(1,10), B(2,1), B(2,2),..., B(10,10).

Loops can be nested to any number of levels. However, a nested loop must be completely contained within the range of the outer loop (i.e., the ranges of the loops may not cross).

Examples

```
FOR I=1 TO 10 STEP .5 UNTIL A>100
```

Executes until I=10 or until the statements within the loop cause A to be greater than 100.

***

FOR (Continued)

```
A=20
FOR J=1 TO 10 WHILE A<25
  A=A+1
  PRINT J,A
NEXT J
```

Executes five times. Variable A reaches 25 before variable J reaches 10.

***

```
ST="X"
FOR B=1 TO 10 UNTIL ST="XXXXX"
  ST=ST: "X"
NEXT B
```

Executes four times. An X is added to the string variable ST until ST = XXXXX.

***

```
A=0
FOR J=1 TO 10 WHILE A<25
  A=A+1
  PRINT J,A
NEXT J
```

Executes 10 times. Variable J reaches 10 before variable A reaches 25.

Note: The Basic Compiler looks for a one-to-one correspondence between each FOR and each NEXT statement. It also ensures that each NEXT statement contains a variable name. However, the compiler does not compare the variable names in the NEXT statements to the variable names in the FOR statements. This necessitates careful programming to avoid improperly nesting of FOR - NEXT loops.

GETLIST

Purpose

GETLIST produces a list of item-ids for a subsequent READNEXT statement.

Syntax

GETLIST list-name {account-name} {TO select-var} {SETTING var} THEN/ELSE

Comments

The list may have been previously saved in the POINTER-FILE by a SAVE-LIST, FORM-LIST or EDIT-LIST command executed at TCL or from a WRITELIST statement executed in a DATA/BASIC program.

List-name is any variable or expression that represents the name under which the list was saved. To access lists saved from other accounts, specify **account-name**.

If the **TO** clause is specified, the list is assigned to **select-var** (as in SELECT CUST TO CUSTLIST); otherwise, the default select-variable is used.

If the **SETTING** clause is used, then **var** is assigned the number of items in the list.

If the list does not exist in the POINTER-FILE, the **ELSE** clause is executed.

Multiple pointers into the same list may be maintained by executing multiple GETLIST statements to the same list-name.

Any number of GETLIST statements can be executed in a DATA/BASIC program, and any number of lists may exist simultaneously by specifying the TO clause.

Examples

```
GETLIST "TEXT" ELSE STOP
```

Produces a list of item-ids using the default select variable.

***

GETLIST (Continued)

```
X = "SAVE.DATA"
GETLIST X TO Y SETTING NUMBER ...
 ELSE PRINT "NOT FOUND"; STOP
READNEXT ID FROM Y ELSE...
```

Selects and prints a list of item-ids in SAVE.DATA and assigns it to variable Y. The number of items in the list is assigned to NUMBER and printed. If list does not exist in the POINTER-FILE, the error message prints and the program terminates.

***

```
GETLIST "S.CARS SYSPROG" TO LIST ELSE STOP
READNEXT ID FROM LIST...
```

Selects list of item-ids previously saved in S.CARS in the SYSPROG account and assigns it to list.

GO

Purpose

The **GO** (or **GOTO**) statement unconditionally transfers program control to any statement within the DATA/BASIC program.

Syntax

GO{TO} statement-label

Comments

Control is transferred to the statement with the specified statement-label. If the label does not exist, an error is printed at compile time.

Example

```
100 A=0
    .
    .
    .
    * BRANCH TO STATEMENT 500
200 GOTO 500
    .
    .
    .
500 A=B+C
    D=100
    .
    .
    .
    * REPEAT PROGRAM
    GO 100
```

Transfers control from statement 200 to statement 500. Execution continues sequentially until GO 100 transfers control back to statement 100.

GOSUB

Purpose

The **GOSUB** statement transfers control to the subroutine beginning with the specified label.

Syntax

```
GOSUB statement-label
```

Comments

GOSUB transfers control to a subroutine specified by **statement-label**, and execution continues sequentially from that statement until a RETURN or RETURN TO statement is encountered.

The RETURN statement returns control to the statement immediately following the GOSUB statement that called the subroutine.

The RETURN TO statement returns control to a specific statement specified by statement-label.

If the GOSUB or the RETURN TO statement refers to a statement-label which does not exist, an error message is printed at compile time.

Examples

```
A=1
GOSUB 100
```

Transfers control to the statement with label 100.

***

```
10 GOSUB 30
15 PRINT X1
   .
   .
20 GOSUB 30
   .
   .
   STOP
30 * SUBROUTINE
   .
   .
   IF ERROR RETURN TO 99
40 RETURN
99 * ERROR RETURN HERE
```

Transfers control to statement 30. The subroutine is executed and statement 40 transfers control back to the statement following the original GOSUB (15 PRINT X1). Execution proceeds sequentially to statement 20, where control is again transferred to statement 30. If the logical variable error is true (1), the conditional RETURN TO 99 path is taken: otherwise, control passes to back to statement 15 once again.

GROUPSTORE

Purpose

GROUPSTORE inserts a string of elements (groups) into another string replacing all, part, or none of the string. (Contrast with the GROUP intrinsic function in Chapter 5.)

Syntax

```
GROUPSTORE substr IN string USING start#, replace#
 {,delim}
```

Comments

Substr is the string to be inserted within **string**.

Start# is the position in string to begin replacing elements. If the specified start# is 0, default is to 1. If the specified start# is greater than the number of elements in string, the replacement string will be appended to the string.

Replace# is the number of elements in string to replace with elements of substr. How these elements are replaced depends on whether replace# (r) is positive, zero or negative.

If r > 0	R elements of string are replaced by the first r elements of substr. Replacement stops if number of elements in string is exhausted.
If r = 0	All of substr is inserted before start# position in string.
If r < 0	R elements of string are deleted starting at start#, and the entire substr is inserted at this position.

Delim is a one-character delimiter to be placed between elements of string and substr. If not specified, it defaults to an AM. If more than one character is specified as delim, only the first character is used.

Note: If both start# and replace# are less than zero, substr is ignored and the number of elements specified in replace# are deleted, starting with start#.

Examples

```
X = "123DEF"
GROUPSTORE "ABC" IN X USING 1,1
```

Replaces the value 123DEF in string X with ABC, so X = "ABC".

***

GROUPSTORE (Continued)

```
A = "10X"
B = "XXXXXXXXXXX"
GROUPSTORE A IN B USING 1,3
```

Replaces the current value of B (attribute 1) with the string "10X". Although 3 replacements were specified, only 1 replacement was made as the string was exhausted.

***

```
A = "ABC456"
GROUPSTORE "123" IN A USING 1,0
PRINT A
```

Prints 123^ABC456. Because the replace# = 0, the replacement string is inserted before the existing string.

***

```
STR = "44^88^99]2\7\4^8^99
GROUPSTORE "ABC" IN STR USING 3,1
PRINT STR
```

Prints 44^88^ABC^8^99.

```
A="ABCXLMNXGHIXJKL"
B="DEF"
Z="X"
GROUPSTORE B IN A USING 2,1,Z
PRINT A
```

Prints ABCXDEFXGHIXJKL. Replaced the second element of A (delimited by X) with DEF.

HEADING

Purpose

The **HEADING** statement causes the current output device to page and prints the specified text at the top of the page.

Syntax

HEADING "expression"

Comments

Expression is the string to be printed at the top of each page.

The page size is determined by the most recent TERM command executed at TCL. If the heading is longer than the TERM line length, the heading wraps around to the next line.

Special heading control characters may be used as part of the HEADING statement. They are:

Character	**Definition**
'C{n}'	Center line (in field of n characters).
'D' or \\	Current date.
'T' or \	Current time and date.
'L' or]	Carriage return and line feed.
'P' or ^	Current page number.
'PP' or ^^	Current page number right-justified in four spaces.
'N'	Inhibits paging.
''	Two consecutive quotes print a single quote.

When HEADING is used, all terminal output is paged (i.e., a carriage return must be pressed when a full page has been printed), unless the N option is used.

HEADING statements may be changed or cleared independently, without altering the page number. The first HEADING statement issued causes a page advance and a new heading, even if the heading is null.

A HEADING statement may be cleared by placing a null string after the statement. Unless both the heading and the footing are cleared, changing an existing heading or footing does not affect the page number.

HEADING (Continued)

Examples

```
HEADING "OUTPUT"
```

Advances current output device to top-of-page, and OUTPUT is printed as the page heading.

***

```
HEADING "'T' 'P' 'L'"
```

Advances output device to top-of-page and prints time, date and page number, followed by a carriage return/line feed.

***

```
S="JANE''S REPORT'L'PAGE'PP'"
HEADING S
```

Advances to top-of-page and prints JANE'S REPORT, followed by a carriage return/line feed, and PAGE, followed by the current page number right-justified in a field of 4 spaces.

***

```
HEADING ""
```

Clears the HEADING statement currently in effect.

IF (single-line)

Purpose The single-line **IF** statement allows conditional execution of a sequence of DATA/BASIC statements.

Syntax IF expression THEN/ELSE

Comments If the test condition specified by the **expression** is true (i.e., nonzero), then the statement(s) following **THEN** are executed.

If the expression is false (zero), the statement(s) following **ELSE** are executed. If the ELSE case is omitted, then control passes to the next sequential statement following the entire IF statement.

One or more statements may follow the THEN or ELSE clause, but they must all be on the same line and separated by semicolons. For example:

```
   IF ITEM THEN PRINT X;  X=X+1 ELSE PRINT Y; GO 5
```

If the current value of item is true (nonzero), the value of X is printed and incremented by 1. Control then passes to the next statement in the program. If ITEM is false (zero), the value of Y is printed and control transfers to statement 5.

Any statements may appear in THEN and ELSE clauses, including more IF statements.

Examples

```
IF A="STRING" THEN PRINT "MATCH"
```

Prints MATCH if value of A is STRING.

***

```
IF X>5 THEN IF X<9 THEN GOTO 10
```

Transfers control to statement 10 if X is greater than 5 but less than 9.

***

```
IF Q THEN PRINT A ELSE PRINT B; STOP
```

Prints value of A if Q is a nonzero integer or if Q evaluates to a null string. If Q=0, value of B is printed and program terminates. If Q is not an integer, an error message is returned.

***

IF (single-line) (Continued)

```
IF A=B THEN STOP ELSE IF C THEN GOTO 20
```

Terminates program if A=B; if A does not equal B and C is nonzero, control passes to statement 20.

***

```
IF A = 0 ELSE PRINT A
```

Prints value of A if it is nonzero. If A is zero, control passes to next statement.

IF (multiline)

Purpose

The multiline **IF** statement is functionally identical to the single-line IF statement; however, the statement sequences may be placed on multiple program lines.

Syntax

There are four possible forms for the multiline IF statement. They are outlined below:

```
IF expression THEN
statement(s)
.
.
END {ELSE statement(s)}

IF expression THEN
statement(s)
.
.
END {ELSE
statement(s)
.
.
END}

IF expression THEN statement(s) {ELSE
statement(s)
.
.
END}

IF expression ELSE
statement(s)
.
.END
```

Comments

The **statement** sequences in the **THEN** and **ELSE** clauses may be placed on multiple program lines, with each sequence terminated by an END.

For every multiline THEN and ELSE statement, there must be a corresponding END statement. If there are not enough END statements, a compilation error results. If there are too many END statements, the program may compile successfully, but the program may terminate early and not all of the program statements may get compiled.

IF (multiline) (Continued)

Examples

```
IF ABC=ITEM+5 THEN
   PRINT ABC
   STOP
END ELSE PRINT ITEM; GOTO 10
```

Prints value of ABC and terminates program if ABC=ITEM+5; otherwise, the value of ITEM is printed and control passes to statement 10.

***

```
IF NUM THEN
   PRINT MESSAGE
   PRINT NUM
   NUM = 100
END
```

Prints value of MESSAGE and NUM and assigns 100 to NUM if value of NUM is nonzero.

***

```
10 IF S="XX" THEN PRINT "OK" ELSE
     PRINT "NO MATCH"
     PRINT S
     STOP
   END
20 REM REST OF PROGRAM
```

Prints OK and passes control to statement 20 if s is XX; otherwise, NO MATCH and the value of S are printed, and the program terminates.

***

```
IF X>1 THEN
   PRINT X
   X=X+1
END ELSE
   PRINT "NOT GREATER"
   GOTO 75
END
```

If X>1, prints value of X, increments X and passes control to statement following the second END; Otherwise, prints NOT GREATER and passes control to statement 75.

INCLUDE

Purpose

The **INCLUDE** statement stores large or commonly used sections of code, such as COMMON or EQUATE areas, outside the source code item.

Syntax

INCLUDE item-name {FROM {DICT} filename}

Comments

INCLUDE may be used anywhere within a DATA/BASIC program as often as desired, so the source code may expand to any size.

INCLUDE must be the only statement on the line. **Filename** is optional. If omitted, the **item-name** is retrieved from the file containing the item being processed.

An included item may contain INCLUDE statements in addition to other code, up to 150 levels. An item may contain any number of INCLUDE statements, which altogether count as one of the 150 levels.

When you compile a program containing INCLUDE statements, an initial pass is made on the source item to merge in the included items. Since the compiler processes only the resulting program, any line number references will be different from the attribute numbers in the actual item(s). The only way to obtain the correct line numbers is to use the 'L' option when compiling the program or to use BLIST with the 'M' option.

Examples

```
INCLUDE ITEMA
```

Includes the item called ITEMA from the file containing the program being compiled.

***

```
INCLUDE ITEM1 FROM TESTFILE
```

Includes ITEM1 in the current program.

***

```
INCLUDE A FROM DICT FILEN
```

Includes item A, found in the dictionary level of file FILEN, in the program being compiled.

INPUT

Purpose

The **INPUT** statement prompts the user for input.

Syntax

```
INPUT variable{,length}{:}{_} {WITH expression}
 {FOR time THEN/ELSE}
```

Comments

INPUT displays a prompt character at the user's terminal. The user types in a numeric quantity or a string, which is assigned to **variable**.

If the optional **length** is specified, an automatic RETURN is executed as soon as that many characters have been entered. This is useful when programming fixed length input fields, because it eliminates the need to enter a RETURN. If length is not specified, the maximum input is 140 characters.

If the optional colon (**:**) is used, the carriage return is inhibited, and the cursor remains positioned after the value input. This is useful when programming multiple inputs on one line.

The optional backarrow (<) or underline (_) is used in conjunction with the length specification. When the specified number of characters has been input, the program waits for a carriage return to be entered. If the user tries to enter more characters, the bell sounds at the terminal.

The **WITH expression** lets you specify 1 or 2 optional delimiters for terminating input from the terminal.

FOR time specifies how long the system waits for input before transferring program control. Time is specified in tenths of seconds. The maximum number that may be used is 32,767, corresponding to 54 minutes and 36 seconds.

If the input is entered within the specified time, control transfers to the **THEN** clause. Otherwise, control transfers to the **ELSE** clause.

The options must appear in the order shown in the syntax above.

If only a carriage return is entered in response to the prompt, a null string is assigned to variable.

INPUT (Continued)

Examples

```
INPUT VAR
```

Requests a value for variable VAR.

***

```
L=3
INPUT X,L
```

Requests input for variable X. When three characters have been entered, a carriage return is executed automatically.

***

```
INPUT Y:
```

Requests input for variable Y. No carriage return is executed after the value is entered.

***

```
EQU CR TO CHAR(13)
INPUT STATE WITH CR
```

Requests input for STATE. When you want to terminate input, just press the carriage return key.

***

```
INPUT VAR,2_ FOR 50 ELSE GO 99
```

Waits five seconds for user to enter a carriage return after a two-character value is input for STATE. If the five seconds elapse and nothing is input, control transfers to statement 99.

***

```
INPUT STRING,8 WITH ".":"/" FOR 100 THEN
 PRINT "STRING WAS INPUT IN TIME"
END ELSE
 PRINT "INPUT TOO SLOW"
END
```

Prompts for a string no longer than 8 characters. If no input has been entered after 10 seconds, the ELSE clause is taken. Otherwise, the THEN clause is executed. To terminate input, type either a . or a /.

INPUT USING

Purpose

The **INPUT USING** statement solicits input data from the terminal under control of the SCREENPRO Screen Processor.

Syntax

```
INPUT var1 USING var2 {,source-exp} {AT step#}
{SETTING var3} THEN/ELSE
```

Comments

Var1 is the destination variable or array where data returned from the Screen Processor goes. It also identifies the type of data structure being used (in this case, a dynamic array).

Var2 specifies the name of the variable containing the compiled screen definition item.

Note: The compiled screen definition item must have been read and assigned to a variable previously, via a DATA/BASIC READ statement.

Source-exp specifies the data structure where the data that is passed to the Screen Processor for updating resides. Data passed to the Screen Processor must be in the same format as data returned from the Screen Processor.

AT step# specifies the screen's step number at which to begin processing. If not specified, processing begins with Step 1.

The **SETTING** clause specifies a variable (**var3**) that will be assigned to the screen's current step number when a screen exit occurs.

The AT and SETTING clauses provide a means to exit from the screen, perform auxiliary operations in the program and return to the screen either at the point where the exit occurred, or at another screen step.

The **ELSE** clause is executed when an exit from the screen is processed. If the screen steps end normally or a File Item command is processed, the ELSE clause is skipped, and control is passed to the next statement in the program.

INPUT USING (Continued)

Examples

```
OPEN "C/M.SCREENS" ELSE
  STOP 201,"C/M.SCREENS"
  END
READ CUST.UPD.SCRN FROM "#C/M.UPDATE" ELSE
  PRINT "COMPILED SCREEN MISSING"
  STOP
  END
OPEN "CUST/MASTER" TO CM ELSE
  STOP 201,"CUST/MASTER"
  END
PRINT "ENTER CUSTOMER NUMBER:":
INPUT C.NUMBER
READ ITEM FROM CM,C.NUMBER ELSE
  STOP 202,C.NUMBER
  END
INPUT ITEM USING CUST.UPD.SCRN,ITEM ELSE
  GOTO 10
  END
WRITE ITEM ON CM,C.NUMBER
```

Opens C/M.SCREENS file and reads the compiled screen definition. Opens CUST/MASTER file and prompts fro a customer number. Input customer number, reads data item and updates data item using a predefined screen. Writes updated item to file.

INS

Purpose

The **INS** statement inserts an attribute, value or subvalue into a dynamic array using a dynamic array reference. (The INS statement makes the INSERT intrinsic function obsolete.)

Syntax

INS expression BEFORE dyn-array-reference

Comments

Expression specifies the value to be inserted into the dynamic array at the position specified by **dyn-array-reference**. Expression may be a dynamic array reference itself.

Dynamic array indices are evaluated as follows:

1. Nonnumeric expressions print a warning message and default to zero.

2. Noninteger expressions truncate the decimal value (i.e., 1.7 and 1.2 become 1).

3. All trailing zero-valued positional expressions (except attr#-expression) are ignored.

4. Any remaining zero-valued positional expressions are treated as ones.

5. If multiple negative positional values occur, the first occurrence remains negative and subsequent values are converted to 1.

6. The single remaining negative value creates a new position according to the following:

   If the dynamic array is not null, the insertion value (preceded by a delimiter) is added as a new attribute, value or subvalue (depending on which position was negative) at the end of the existing item, attribute or value, respectively. The remaining positional values, if any, are treated as usual.

   If the dynamic array is null, the negative value is converted to 1, and Rule 7 below is applied.

7. If the positional expression specifies a nonexistent position (i.e., greater than the number of attributes, values or subvalues below), or if the dynamic array is initially null, then nulls are created where necessary to put the insertion value in the position specified by the expressions.

INS (Continued)

Examples

```
INS 123 BEFORE ITEM<3>
```

Inserts attribute value 123 before the third attribute in dynamic array ITEM.

***

```
INS PRICE<3> BEFORE A<X,Y>
```

Inserts value of attribute 3 in array PRICE before the Y'th multivalue in the X'th attribute in dynamic array A.

***

```
INS YR*365 BEFORE ORDER<I,J,3>
```

Inserts value of YR*365 before the third subvalue in the specified multivalue and attribute.

***

```
IF PART<1,X+2> = 0 THEN
   INS 1 BEFORE PART<1,X+2>
   END
```

If the multivalue is zero, then a new multivalue (1) is created before the indicated multivalue.

***

For the following examples:

```
   N = ""
   S = "1\2\3]11\22]333^A\B\C]AA"
   X = "XXX"
```

```
INS X BEFORE S<0,2,0>
```

Result = 1\2\3]XXX]11\22]333^A\B\C]AA. (See Rules 3 and 4 above.)

***

```
INS X BEFORE S<1,-2,-5>
```

Result = 1\2\3]11\22]333XXX^A\B\C]AA. (See Rules 5 and 6 above.)

***

```
INS X BEFORE N<1,2,0>
```

Result =]XXX. (See Rules 3 and 7.)

LOCATE

Purpose

The **LOCATE** statement finds the position of an expression within a dynamic array or within an attribute or value of a dynamic array.

Syntax

```
LOCATE exp1 IN exp2{<attr# {,value#}>}{,start-
position} {BY seq} SETTING variable THEN/ELSE

LOCATE(exp1,exp2{,attr#{,value#}};setting-
variable{;seq}) THEN/ELSE
```

Note: The syntax of the LOCATE statement is different on releases prior to 3.2. If you are running a release older than 3.2, use the FIX-LOCATE verb to update old programs.

Comments

Exp1 can be a literal string, variable, array element or function that specifies the value or string to be located. It must not contain attribute marks. If exp1 is null, the position returned indicates the last position, unless there is a null element in the string being searched. The up arrow (^) and brackets ([,]) are reserved.

Exp2 is the dynamic array being searched.

If neither **attr#** nor **value#** is specified, an attribute search takes place, with **start-position** specifying the attribute where the search begins. This way, you can skip over unwanted attributes. A value of 1 searches the entire dynamic array.

Note: If you use the second form of the LOCATE statement, the start-position value always defaults to 1.

If only attr# is supplied, a value search is performed within the specified attribute only. In this case, start-position refers to a value. This lets you skip over unwanted values in that attribute. When searching values, exp1 must not contain any value marks.

If both attr# and value# are specified, a subvalue search occurs, within the specified multivalue, which is contained inside the specified attribute. Start-position refers to the subvalue mark where the search is to begin. Exp1 may not contain any subvalue marks.

LOCATE (Continued)

Note: If the start-position number is greater than the number of elements specified, then variable will be set to this number. If used with a dynamic array insertion, the value is placed at this position with null attributes, values or subvalues in between.

BY seq specifies that the elements in the dynamic array are sorted in ascending or descending order. The following sequences are available and must be enclosed in double quotes:

AL	ascending, left-justified (standard alphanumeric sort).
AR	ascending, right-justified (useful for numerics with different lengths).
DL	descending, left-justified (standard alphanumeric sort).
DR	descending, right-justified (useful for numerics with different lengths).

Variable (or **setting-variable**) is the name that will contain the position number of the value being searched for. If the value is not found, the THEN/ELSE clause is executed and one of two things may happen.

If no sort order was specified, the variable is set to the position past the last attribute.

If a sort order was specified, then variable is set to the correct position where the value should go, so the elements remain in order.

Examples

```
LOCATE "CALIF" IN DA SETTING POS ...
 ELSE PRINT 'NO "CALIF"'; STOP
```

Locates attribute with string value CALIF in dynamic array DA, starting the search with the first attribute. If CALIF is not found, error message is printed. POS is set to a value indicating the position of the attribute in DA.

***

LOCATE (Continued)

```
ITEM = "24^RESISTOR^243]523]311]10]3"
A = 10
LOCATE(A,ITEM<3>,2;K) ELSE PRINT "NOT FOUND"; STOP
```

Locates multivalue equal to 10 in the third attribute of dynamic array ITEM. Search begins with the second multivalue. K is set to the value indicating the position of 10 in the third attribute (i.e., 4).

***

```
I=3
A=2
V=5
PRINT "ENTER QTY TO DELETE":
INPUT QTY
LOCATE QTY IN ARRAY(I)<A,V> SETTING J ...
 ELSE PRINT "QTY NOT FOUND"; STOP
DEL ARRAY(I)<A,V,J>
```

Prompts for terminal input, then locates the subvalue QTY in the second attribute and the fifth multivalue in the dynamic array assigned to the third element of ARRAY. Scanning begins with the first subvalue. J is set to a value indicating the position of the value of QTY. If found, it is deleted; otherwise, the message is printed.

***

```
DEPT="PERSNL"
LOCATE(DEPT,ITEM<2>;K;"AL") ELSE
   ITEM=INSERT(ITEM,2,X,0,DEPT)
END
```

Locates multivalue PERSNL in second attribute of dynamic array ITEM. The multivalues in the second attribute are sorted in ascending order, left-justified. If the value of DEPT (PERSNL) is not found, it is inserted in the proper order, as defined by X (i.e., if ITEM=19^ACCT]ENG]PROD^12]6, it would now be 19^ACCT]ENG]PERSNL]PROD^12]6.

***

LOCATE (Continued)

In the following examples, dynamic array INFO contains "22^76^24]523]21^9]7\4\54".

```
LOCATE(76,INFO;K) ELSE PRINT "NOT THERE"; STOP
```

Searches first attribute of dynamic array INFO for 76. Returns a 2.

***

```
LOCATE 54 IN INFO<4,2> SETTING K ...
 ELSE PRINT "NOT FOUND"; STOP
```

Searches all subvalues within attribute 4, value 2 and returns a 3.

***

```
LOCATE(523,INFO<3>;K) ELSE GOTO 10
```

Searches values in attribute 3 and returns a 2.

LOCK

Purpose

The **LOCK** statement sets an execution lock, so multiple DATA/BASIC programs cannot update the same file simultaneously.

Syntax

LOCK expression {THEN/ELSE}

Comments

Expression specifies which execution lock is to be set. This is determined by the programmer.

If the specified lock has already been set by another concurrently running program, and an ELSE clause is not provided, execution halts temporarily, until the lock is reset by the other program.

If an **ELSE** clause has been provided, the statements in the ELSE clause are executed and the program continues.

Another DATA/BASIC program cannot set the same lock, until it is reset with the UNLOCK statement, by the program that originally issued the LOCK.

To lock single items in a file, use the READU, READVU and MATREADU statements.

Note: The DATA/BASIC and PROC processors use the same 256 execution locks, numbered 0-255.

Examples

```
LOCK 15 ELSE STOP
```

Sets execution lock 15. If lock 15 is already set, the program terminates.

***

```
LOCK 2
```

Sets execution lock 2.

***

```
LOCK 10 ELSE PRINT X; GOTO 5
```

Sets execution lock 10. If it is already set, X is printed and control transfers to statement 5.

LOOP

Purpose

The **LOOP** statement constructs program loops, using either WHILE or UNTIL conditions.

Syntax

```
LOOP {statement(s)} WHILE expression DO
 {statement(s)} REPEAT
LOOP {statement(s)} UNTIL expression DO
 {statement(s)} REPEAT
```

Comments

If specified, the **statement(s)** following LOOP are executed first. Then the **expression** is evaluated.

If the expression following **WHILE** or **UNTIL** evaluates to true (nonzero), the statement(s) following **DO**, if any, are executed and control goes back to the beginning of the loop. If the expression evaluates to false (zero), control passes to the next sequential statement following **REPEAT**.

Statements used within the loop may be placed on one line separated by semicolons, or they may be placed on multiple lines.

Because the loop statement requires a logical condition (one which evaluates to 0 or 1), the compiler allows a READNEXT or a LOCATE statement to provide this conditional. The READNEXT statement always returns a 0 or 1, depending on whether or not it can assign an item-id from a select-list.

Examples

```
A=0
LOOP UNTIL A=4 DO A=A+1; PRINT A REPEAT
```

Prints sequential values of A from 1 through 4. (Loop executes 4 times.)

***

```
J=0
LOOP
PRINT J
J=J+1
WHILE J<4 DO REPEAT
```

Prints sequential values of J from 0 through 3. (Loop executes 4 times.)

***

LOOP (Continued)

```
X=100
LOOP X=X-10 WHILE X>40 DO PRINT X REPEAT
```

Prints values of X from 90 down through 50 in increments of -10, so the loop executes 5 times.

***

```
Q=6
LOOP Q=Q-1 WHILE Q DO
 PRINT Q
REPEAT
```

Prints value of Q in this order: 5, 4, 3, 2, 1.

***

```
B=1
LOOP UNTIL B=6 DO
 B=B+1
 PRINT B
REPEAT
```

Prints values of B from 2 through 6, as loop executes 5 times.

***

```
LOOP I=I+1 WHILE READNEXT ID DO REPEAT
```

Increments I as long as item-ids are read from select-list.

MAT

Purpose

The **MAT** assignment and copy statements are used to assign values to each element in an array.

Syntax

```
MAT variable = expression
MAT variable = MAT variable
```

Comments

The MAT assignment statement (first syntax above) assigns a single value (the result of **expression**) to all elements in an array (specified by **variable**). The specified array must have been previously dimensioned by a DIMENSION or COMMON statement.

The MAT copy statement copies one array to another. The first element of the array on the right becomes the first element of the array on the left, and so on. Each variable must have been dimensioned, and the number of elements in the two arrays must match; if not, an error message displays and the program transfers to the debugger.

Arrays are copied in row major order. For example:

Program Code　　　**Resulting Array Values**

```
DIM X(5,2), Y(10)      X(1,1) = Y(1) = 1
FOR I=1 TO 10          X(1,2) = Y(2) = 2
  Y(I)=I               X(2,1) = Y(3) = 3
NEXT I                          .
.                               .
.                               .
MAT X = MAT Y          X(5,2) = Y(10) = 10
```

This example dimensions two arrays, both having 10 elements, initializes array Y to the numbers 1 through 10, then copies array Y to array X.

Examples

```
MAT TABLE=1
```

Assigns a value of 1 to each element of array TABLE.

***

```
MAT XYZ=A+B/C
```

Assigns the value of expression A+B/C to each element of array XYZ.

***

MAT (Continued)

```
DIM A(20), B(20)
.
.
MAT A = MAT B
```

Dimensions two vectors of equal length and assigns the values of the elements in array B to the corresponding elements in array A.

***

```
DIM TAB1 (10,10), TAB2(50,2)
.
.
MAT TAB1 = MAT TAB2
```

Dimensions two arrays to the same number of elements and copies TAB2 values to TAB1 in row major order.

MATBUILD

Purpose

The **MATBUILD** statement builds a string variable from a dimensioned array. (MATBUILD performs the opposite function of the MATPARSE statement.)

Syntax

MATBUILD variable FROM array{,start{,end}} {USING character}

Comments

Array must be a dimensioned array.

Start and **end** are the optional starting and ending positions from which to start and stop retrieving elements from array.

If start is <= 0, it defaults to 1. If end is < 0 or > the size of array, it defaults to the size of array. If start is > end and end is >= 0, no operation takes place. If end is a negative number, it indicates that assignment should continue through the end of the array.

Variable is the destination variable for data built from elements of array.

Character is an optional delimiter to be inserted between elements of array when building variable. It can be any value from hex 00 to hex FE, and must be enclosed in single quotes.

If it is omitted or if it is specified, but null, character defaults to an attribute mark. If more than one character is specified, only the first one is used.

During the MATBUILD process, a string is built and assigned to variable by concatenating all elements of array together, separated by character. The process terminates when the last element of array has been processed, or when all remaining elements of array are null. That way, variable contains no trailing, null elements.

No run time error messages are ever generated.

Examples

MATBUILD VAR1 FROM ARRAY1

Builds a string VAR1 from the values of ARRAY1 concatenated together.

***

MATBUILD (Continued)

```
DIM ARR(5)
ARR(1) = 'A3'
ARR(2) = 'FE'
ARR(3) = '56'
ARR(4) = 'C7'
ARR(5) = '3D'
.
MATBUILD X FROM ARR,2
```

Builds the string "FE^56^C7^3D", starting with the second element of ARR through the end, and assigns it to X.

***

```
MATBUILD B FROM ARR USING ','
```

Builds string B from ARR using a comma as delimiter. String B contains "A3,FE,56,C7,3D".

***

```
DIM Y(4)
Y(1) = 'THIS'
Y(2) = 'IS'
Y(3) = 'A'
Y(4) = 'TEST'
.
MATBUILD SENTENCE FROM Y USING ' '
```

Takes elements of array Y and builds them into the string SENTENCE with a blank space between each word.

MATINPUT USING

Purpose

The **MATINPUT USING** statement solicits input data from the terminal under control of the SCREENPRO Screen Processor. The input/output data for SCREENPRO is in the form of a dimensioned array.

Syntax

```
MATINPUT var1 USING var2 {,source-exp} {AT step#}
{SETTING var3} THEN/ELSE
```

Comments

Var1 is the destination variable or array where data returned from the Screen Processor goes. It also identifies the type of data structure being used (in this case, a dimensioned array).

Var2 specifies the name of the variable containing the compiled screen definition item.

Note: The compiled screen definition item must have been read and assigned to a variable previously, via a DATA/BASIC READ statement.

Source-exp specifies the data structure where the data that is passed to the Screen Processor for updating resides. Data passed to the Screen Processor must be in the same format as data returned from the Screen Processor.

AT step# specifies the screen's step number at which to begin processing. If not specified, processing begins with Step 1.

The **SETTING** clause specifies a variable (**var3**) that will be assigned to the screen's current step number when a screen exit occurs.

The AT and SETTING clauses provide a means to exit from the screen, perform auxiliary operations in the program and return to the screen either at the point where the exit occurred, or at another screen step.

The **ELSE** clause is executed when an exit from the screen is processed. If the screen steps end normally or a File Item command is processed, the ELSE clause is skipped, and control is passed to the next statement in the program.

Example

```
MATINPUT REC USING INVOICE.SCRN ...
  SETTING RETURN.STEP ELSE GOTO 25
```

Updates array REC using a predefined screen.

MATPARSE

Purpose

MATPARSE assigns the elements of a string variable to the variables of a dimensioned array. (MATPARSE performs the opposite function of the MATBUILD statement.)

Syntax

MATPARSE array{,start{,end}} FROM variable {USING character} {SETTING nmelements}

Comments

Array must be a dimensioned array.

Start and **end** are the optional starting and ending positions from which to start and stop assigning elements within array.

If start is <= 0, it defaults to 1. If end is omitted, or if it is < 1 or greater than the size of the array, assignment continues to the end of the array. If start > end and end is >= 0, no operation is performed.

Variable is the source variable from which data is assigned to the elements of array.

Character is the optional delimiter found between elements of the variable used to build the array. It can be any value from hex 00 to hex FE, and must be enclosed in single quotes.

If it is omitted or if it is null, character defaults to an attribute mark. If more than one character is specified, only the first one is used.

The value **nmelements** is the number of elements of array that are assigned a value from variable.

During the MATPARSE process, each element of variable is assigned to a successive element or array. If the size of array is greater than the number of parsed elements of variable, the remaining elements of array are assigned a null value. The process terminates when the last element of array has been assigned a value, even if variable has not been exhausted yet.

No run time error messages are ever generated.

MATPARSE (Continued)

Examples

```
X = "1.2.2.1.2.3.4.4.8.2"
DIM ARR(10)
MATPARSE ARR FROM X USING '.'
```

Assigns each element of X, separated by a period, to dimensioned array ARR.

***

```
A = "THIS IS A TEST FOR YOU."
DIM B(12)
MATPARSE B FROM A USING ' ' SETTING Y
PRINT Y
```

Assigns the elements of variable A to array B and prints the value 6, the number of elements assigned to array B.

***

```
VAR = "3,2,5,9,9,2,8"
DIM ARR(4)
MATPARSE ARR FROM VAR USING ','
```

Assigns only the first four values of variable VAR to array ARR, because ARR has only four elements.

***

```
V1 = "ABC^DEF^GHI^JKL^MNO^PQR"
DIM ARR1(15)
MATPARSE ARR1,5,10 FROM V1
```

Assigns the values of variable V1 to dimensioned array ARR1, starting with the fifth element of ARR1 and ending with the tenth element.

MATREAD

Purpose

The **MATREAD** statement reads a file item and assigns each attribute to consecutive elements of a dimensioned array.

Syntax

```
MATREAD array FROM {file-variable,}item-id
{SETTING var} THEN/ELSE
```

Comments

File-variable specifies the file previously assigned to that file-variable via an OPEN statement.

If a file-variable is not specified, the internal default file-variable is used (i.e., the file most recently opened without a file-variable).

If the **SETTING** clause is used and the read is successful, **var** is set to the number of attributes in the item assigned to **array**. For example, if the array was dimensioned for 100 elements and the item read in was only 12 attributes long, var would be set to 12.

Trailing, null attributes in the file item are counted, because they were assigned to the element array.

If a nonexistent item is specified, the **ELSE** clause is executed.

If the number of attributes in the item is less than the dimensioned size of the array, the remaining elements are assigned null values. If it is greater than the size of the array, the remainder of the item is assigned to the last dimensioned array element.

Examples

```
MATREAD ITEM FROM F1,'AB-123' ELSE STOP
```

Reads item 'AB-123' from file F1 into array ITEM. IF AB-123 does not exist, the program terminates.

***

```
DIM ITEM(20)
OPEN 'LOG' TO F1 SETTING K ELSE STOP
MATREAD ITEM FROM F1, 'TEST' ELSE STOP
```

Reads the item named TEST from the data file LOG and assigns the string value of each attribute to consecutive elements of array ITEM. K is set to the number of attributes read in.

MATREADU

Purpose

MATREADU locks a specific item in a file prior to updating it.

Syntax

MATREADU array FROM {file-var,} item-id, {SETTING var} {LOCKED statement(s)} THEN/ELSE

Comments

MATREADU works the same way as the MATREAD statement, except that it locks the item to be updated. This prevents an item from being updated by two or more items simultaneously.

Other processors which encounter an item lock are suspended until the item becomes unlocked, unless the optional LOCKED clause is specified. If it is, the statements following LOCKED are executed.

The item can be unlocked in any of the following ways:

The process which has the item locked completes its update.

The program is terminated.

A RELEASE statement is issued.

The item is written back to the file with a MATWRITE statement (without the optional U).

The number of item locks is user-defined.

Examples

```
MATREADU ARR1 FROM X, ITEM3 ELSE GOTO 120
```

Reads ITEM3 from file X into array ARR1 or, if ITEM3 does not exist, transfers control to statement 120.

***

```
MATREADU ARRAY FROM FIL1, "REC" SETTING K ...
 LOCKED PRINT "CANNOT ACCESS" ELSE STOP
```

Reads item REC from file FIL1 and sets K to the number of items read in. If REC cannot be read, error message displays and program terminates.

MATWRITE

Purpose

The **MATWRITE** statement writes a dimensioned array to a file item.

Syntax

```
MATWRITE variable ON {file-var,} item-id
```

Comments

Variable is the name of the array whose elements are assigned to **item-id**.

If a **file variable** is specified, the item is written to the file previously assigned to that file variable via the OPEN statement. Otherwise, the internal default file variable is used.

If the specified item-id does not exist, a new item is created.

The number of attributes in the item is determined by the dimensioned size of the array.

Null trailing vector elements are not written as null attributes in the file.

Note: MATWRITE no longer issues an error message when dimensioned array elements contain attribute marks.

Examples

```
MATWRITE ITEM ON 'AB-123'
```

Writes the contents of array ITEM to the file previously opened without a file variable as an item with an id of AB-123.

***

```
DIM ITEM (10)
OPEN '', 'TEST' ELSE STOP
FOR I=1 TO 10
   ITEM(I)=I
NEXT I
MATWRITE ITEM ON "JUNK"
```

Writes the contents of ITEM to an item named JUNK in the file named TEST. JUNK now contains 10 attributes whose string values are 1 through 10.

MATWRITEU

Purpose

MATWRITEU works just like the MATWRITE statement, except that it leaves a previously locked item locked at the end of the write.

Syntax

```
MATWRITEU variable ON {file-var,} item-id
```

Comments

MATWRITEU does not actually lock an item. It simply does not unlock an item that is already locked.

Example

```
FOR I=1 TO 5
   ITEM(I) = I
NEXT I
MATWRITEU ITEM ON "TJUNK"
```

Writes 10 attributes with values 1 through 10 to item named TJUNK. If TJUNK was locked before the write, it remains locked afterwards.

NEXT

Purpose

The **NEXT** statement is used in conjunction with the FOR statement. NEXT increments the specified variable by the increment value and determines where control should pass.

Syntax

NEXT variable

Comments

The **variable** in the NEXT statement must be the same as the variable in the FOR statement.

Any statements may appear between a FOR and NEXT statement, including statements that transfer control out of the loop. However, no statement should transfer control into a FOR-NEXT loop, except for the FOR statement itself.

For more information on FOR-NEXT loops, refer to the FOR statement.

Examples

```
FOR J=2 TO 11 STEP 3
   PRINT J+5
NEXT J
```

Prints 7, 10, 13 and 16. Each time the loop is executed, 3 is added to J until the value of J exceeds 11.

***

```
FOR K=10 TO 1 STEP -1
   .
   .
NEXT K
```

Passes through loop 10 times. Each time the value of K is decreased by 1.

***

```
LIMIT = 1
FOR VAR= 0 TO LIMIT STEP .1
   .
   .
NEXT VAR
```

Passes through loop 11 times, and value VAR is increased by .1 each time.

NULL

Purpose The **NULL** statement specifies no operation. It is used anywhere a DATA/BASIC statement is required, but no operation is desired.

Syntax

```
NULL
```

Comments A NULL statement may be used anywhere in a program where a DATA/BASIC statement is required.

Examples

```
10 NULL
```

Results in no operation; however, because it has a statement label, it may be used as an entry point for a GOTO or GOSUB statement.

***

```
IF A=0 THEN NULL ELSE
  PRINT "A NONZERO"
  GOSUB 45
  STOP
  END
```

Executes the statements in the ELSE clause if the value of A is nonzero. If A=0, no action is taken and control passes to the statement following END.

***

```
READ A FROM "ABC" ELSE NULL
```

Reads file item ABC and assigns it to variable A. If ABC does not exist, no action is taken.

***

```
IF X1 MATCHES "9N" ELSE GOTO 100
```

Branches to statement 100 if the current value of X1 does not contain 9 numeric characters. If it does, then no action is taken and control passes to the next sequential statement.

ON GOSUB

Purpose

ON GOSUB transfers control to an internal subroutine determined by the current value of the given expression.

Syntax

ON expression GOSUB statement-label {,statement-label...}

Comments

The ON GOSUB statement evaluates the **expression** and truncates it to an integer value. Control is then transferred to the **statement-label** whose number equals the value of the expression.

If the expression evaluates to less than one, the following message displays and the branch is taken to the first statement label:

```
[B22] BRANCH INDEX OF x IS ILLEGAL;
      BRANCH TAKEN TO FIRST STATEMENT-LABEL!
```

If the expression exceeds the number of statement labels in the list, the following message displays and the branch is taken to the last statement label:

```
[B23] BRANCH INDEX OF x EXCEEDS NUMBER OF STATEMENT-LABELS;
      BRANCH TAKEN TO LAST STATEMENT-LABEL!
```

The RETURN statement returns control to the statement immediately following the ON GOSUB statement that called the subroutine.

The RETURN TO statement returns control to a specific statement specified by statement-label.

Examples

```
ON X+Y GOSUB 101, 117, 103, 216
PRINT Y
```

Transfers control to the internal subroutine with statement label 101, 117, 103 or 216, depending on whether the value of X+Y is 1, 2, 3 or 4.

***

```
ON Z GOSUB 20, 20, 29
INPUT A
```

Transfers control to label 20 if Z <= 2, or to label 29 in all other cases. The system prompts for input when control returns via a RETURN or RETURN TO statement.

***

ON GOSUB (Continued)

```
IF T GE 1 AND T LE 3 THEN
   ON T GOSUB 110, 120, 130
   END
```

The IF statement guarantees that T is in the range of the computed GOSUB statement.

ON GOTO

Purpose

The **ON GOTO** statement transfers control to a statement-label selected by the current value of expression.

Syntax

ON expression GOTO statement-label {,statement-label...}

Comments

ON GOTO evaluates the **expression** and truncates it to an integer value. Control is then transferred to the **statement-label** with the corresponding value.

If the expression evaluates to less than one, the following message displays and the branch is taken to the first statement label:

```
[B22] BRANCH INDEX OF x IS ILLEGAL;
      BRANCH TAKEN TO FIRST STATEMENT-LABEL!
```

If the expression exceeds the number of statement labels in the list, the following message displays and the branch is taken to the last statement label:

```
[B23] BRANCH INDEX OF x EXCEEDS NUMBER OF STATEMENT-LABELS;
      BRANCH TAKEN TO LAST STATEMENT-LABEL!
```

Note: The subroutine specified by statement label does not have to occur after the ON GOTO statement. It may occur anywhere in the program.

Examples

ON M+N GOTO 40, 61, 5, 7

Transfers control to statement 40, 61, 5 or 7, depending on whether the value of M+N is 1, 2, 3 or 4.

***

ON C GOTO 25, 25, 20

Transfers control to statement 25 if C<2, to statement 20 in all other cases.

***

ON GOTO (Continued)

```
IF A GE 1 AND A LE 3 THEN
   ON A GOTO 110, 120, 130
END
```

The IF statement assures that A is in the range for the computed ON GOTO statement.

OPEN

Purpose

The **OPEN** statement selects a file for subsequent input, output or update.

Syntax

```
OPEN {DICT,} filename {TO file-variable} THEN/ELSE
```

Comments

If **DICT** is specified, the dictionary portion of the file is opened. If omitted, the data section is opened. DICT and **filename** can be either literals or variables.

If **TO file-variable** is specified, the dictionary or data section of the file is assigned to the specified variable for subsequent reference. The file variable can be passed to other programs to eliminate the need to open a file many times.

If the TO file variable option is omitted, an internal default file variable is generated. Subsequent I/O statements not specifying a file variable then automatically default to this file.

If the specified file does not exist, the **ELSE** clause is taken.

If a file is retrieval protected and security codes do not match, an error message displays and the program terminates. Files with update protection may still be opened if update codes do not match, but any attempt to write to the file causes an error message to display, and control transfers to the debugger.

A maximum of 3,224 files can be open at one time.

Examples

```
OPEN 'DICT','QA4' TO F1 ELSE PRINT "NO FILE"; STOP
```

Opens the dictionary portion of file QA4 and assigns it to file variable F1. If QA4 does not exist, the message NO FILE is displayed and the program terminates.

***

```
OPEN 'ABC' TO D5 ELSE
   STOP 201, "ABC"
   END
```

Opens data section of file ABC and assigns it to variable D5. If ABC does not exist, an error message displays and the program terminates.

***

OPEN (Continued)

```
OPEN '', 'TEST' ELSE PRINT "DOES NOT EXIST"; GO 10
```

Opens data section of file TEST and assigns it to an internal default file variable. If TEST does not exist, an error message displays and control transfers to statement 10.

***

```
OPEN 'TEST' ELSE PRINT "DOES NOT EXIST"; GO 10
```

This example functions identically to the one above.

PAGE

Purpose The **PAGE** statement advances the current output device to the next page and prints the heading (footing) at the top (bottom) of the page.

Syntax

```
PAGE
```

Comments The PAGE statement is only valid for print report zero and only functions if a heading or footing is in effect.

Example

```
HEADING="MONTHLY REPORT'L'PAGE'PP'"
.
.
.
PAGE
```

Current output device advances to top-of-form and prints MONTHLY REPORT heading.

PERFORM

Purpose

The **PERFORM** statement allows you to use TCL verbs within a DATA/BASIC program. PERFORM also sends lists to and from TCL verbs and returns error message strings to the program.

Syntax

```
PERFORM <TCL exp> {PASSLIST {<select-var>}}
                  {RTNLIST {<select-var>}}
                  {CAPTURING <var>}
                  {SETTING <var>}
```

Comments

<TCL exp> is any valid expression containing the TCL command to be used. The PERFORM statement makes DATA/BASIC more interactive with the data base management structure of REALITY systems. Virtually any statement that may be executed at TCL may be executed within a DATA/BASIC program using the PERFORM statement.

Note: Any filename passed to a PERFORMed verb must be the name of a file in that account, not a file variable which is the result of an OPEN statement.

PASSLIST <select-var> specifies the variable which contains the select list to be passed to the called processor. Select list must be the result of a SELECTE or GETLIST statement or the RTNLIST clause of a previous PERFORM statement. If PASSLIST is used without a select-var, the default select variable of that program is used.

when used w/ select-var list returns w/ extra ^254

RTNLIST <select-var> specifies the variable where the select list will be returned. If select-var is omitted, the generated select list replaces the contents of the default select variable. The resulting list may be used in a READNEXT statement or in the PASSLIST clause of a subsequent PERFORM.

CAPTURING <var> specifies an alternate destination for text that would otherwise be displayed on the terminal. Each line of output becomes one attribute in the capturing variable. Output directed to the printer still goes to the printer.

SETTING <var> specifies the variable where error messages and their parameters are to be returned. Each error message is returned as a separate attribute, and the parameters within it are separated by value marks. The first value of each attribute (error message) is the referenced error message number.

PERFORM (Continued)

You can use the SETTING clause to capture the text of the error message and obtain the parameter string which caused the error message. (This variable is truncated at approximately 32,000 bytes.) Each attribute may then be examined or printed using the PRINTERR statement.

The error message is still sent to the terminal or printer (or captured) whether or not the SETTING clause is used.

There is no limit to the number of PERFORM statements that may be issued in a program, and up to 32 levels of PERFORM statements may be nested.

The DATA statement is used to specify arguments that are to be passed to a PERFORMed statement. The arguments are specified as an ordered list, separated by commas. Individual arguments must be less than 140 bytes long. For example:

```
PERFORM 'SELECT CUST WITH ORDER-DATE < ...
 "4/11/85"' RTNLIST CUST.LIST
DATA "(CUST-HIST"
PERFORM 'COPY CUST' PASSLIST CUST.LIST
```

Examples

```
STMT = 'SSELECT CUST WITH PAST-DUE BY LASTNAME'
PERFORM STMT RTNLIST LATECUST
LOOP WHILE READNEXT ID FROM LATECUST DO
.
.
REPEAT
```

Creates a list of customers with late payments which can then be used throughout the program.

***

```
STMT = 'SORT CUST BY DUE-DATE NAME DUE-DATE ...
 PAST-DUE (P'
PERFORM STMT PASSLIST LATECUST
```

Passes the previously assigned list of customers with late payments to a PERFORMed process.

***

```
STMT = "SSELECT CUST WITH PAST-DUE BY LAST-NAME"
ST = "SORT CUST BY DUE-DATE NAME DUE-DATE ...
 PAST-DUE (P"
PERFORM STMT RTNLIST
PERFORM ST PASSLIST
```

PERFORM (Continued)

In this example, the list is saved in and passed from the default list.

***

```
GETLIST "LISTNAME" TO INT.LIST.NAME
PERFORM "LIST FILENAME" PASSLIST ...
   INT.LIST.NAME
```

The GETLIST verb passes the list to the LIST verb for processing.

***

```
ST1="SORT CUST WITH "
ST2="PAST-DUE "
ST3="DUE-DATE "
ST4="BY LAST-NAME"
IF CHOICE = 1 THEN
     STMT=ST1:ST2:ST4
     PERFORM STMT RTNLIST LATECUST
     END ELSE
     STMT=ST1:ST3:ST4
     PERFORM STMT RTNLIST CUSTDUE
END
```

This example shows how the TCL exp may be built separately from the PERFORM statement. Selection criteria for an ENGLISH statement can be tailored depending on variables within the DATA/BASIC program.

The value of CHOICE could be determined by a menu selection. If the value is 1, the ENGLISH statement will be SORT CUST WITH PAST-DUE BY LAST-NAME. If the value is not 1, the ENGLISH statement will be SORT CUST WITH DUE-DATE BY LAST-NAME.

PRECISION

Purpose

The **PRECISION** statement determines the degree of precision to which all values will be calculated within a program.

Syntax

```
PRECISION n
```

Comments

The value **n** is the number (from 0 through 6) of decimal places to which all values are calculated and to which all values are truncated.

Only one PRECISION statement is allowed in a program.

If no PRECISION statement is included in a program, values are calculated to 4 decimal places.

Programs that pass values via the COMMON statement and all called subroutines MUST have the same precision.

Programs that use math functions (e.g., SQRT, EXP, etc.) should have a precision in the range 3-5.

If the number of decimal places assigned to a variable is greater than the number specified in the PRECISION statement, the values are truncated to the number specified by PRECISION. Rounding does not take place.

Examples

```
PRECISION 5
PRINT SQRT(5)
```

Prints 2.23606, because 5 decimal places were specified by the PRECISION statement.

***

```
PRECISION 3
X=9.123456
PRINT X
```

Truncates X to 3 decimal places and prints 9.123.

***

PRECISION (Continued)

```
PRECISION 3
Y="4.723428"
X=Y
Z=Y+0
PRINT X,Y,Z
```

Prints 4.723428 as the value of X and Y and 4.723 as the value of Z.

PRINT

Purpose

The **PRINT** statement outputs data to the device selected by the PRINTER statement.

Syntax

PRINT {ON expression} {print-list}

Comments

When PRINTER ON is in effect, **ON expression** is used to send output to multiple print reports, where expression indicates the print report number (from 1 to 127). If ON expression is omitted, print report 0 is used.

Print-list may consist of a single expression or a series of expressions, separated by commas or colons, used to denote formatting. (For more information on formatting output, refer to the next topic, PRINT Using Output Formatting). If print-list is not specified, a blank line is printed.

The HEADING statement affects only print report zero; however, pagination and forms control must be used for other print reports. This can be done by keeping track of the number of lines output and printing top-of-form characters (CHAR(12)) to start new pages. Lack of pagination results in continuous printing across page boundaries or in multiple reports running together.

When PRINTER OFF is in effect, the ON expression option has no effect.

The contents of all print reports used by the program, including print report zero, are output to the printer in sequence when a PRINTER CLOSE is issued, or when the program terminates.

Direct cursor control and video effects may be accomplished by using the @ function within the print-list. The @ function is explained in Chapter 5.

Examples

```
PRINTER ON
N=50
PRINT ON 24 X
PRINT ON N Y
```

Outputs value of X to print file 24 and value of Y to print file 50.

***

PRINT (Continued)

```
PRINTER ON
PRINT ON 10 F1,F2,F3
PRINT ON 20 M,N,P
PRINT ON 10 F4,F5,F6
```

Outputs the values of F1 through F6 to print file 10 and M, N and P to print file 20.

PRINT Using Output Formatting

Purpose The print-list of the PRINT statement may specify tabulation or concatenation when printing multiple items.

Tabulation Output values may be aligned at tab positions across the output page by using commas (,) to separate the print-list expressions. Tab positions are preset at every 18 character positions. For example:

```
PRINT (50*3)+2, A, "END"
```

If A = 37, this statement prints the values across the page as follows:

```
152                37                 END
```

Concatenation Output values can be printed contiguously across the page by using colons (:) to separate the print-list expressions. For example, the following statement prints the message "THE VALUE OF A IS 5010":

```
PRINT "THE VALUE OF A IS ":50:5+5
```

After the entire print-list has been printed, a carriage return and line feed is executed, unless the print-list ends with a colon. In that case, the next value in the next PRINT statement is printed on the same line in the very next character position. For example:

```
PRINT A:B:,C,D:
PRINT E,F,G
```

These statements produce the same output as the following statement:

```
PRINT A:B,C,D:E,F,G
```

Format Strings The output in a PRINT statement may be formatted using Format Strings. Format Strings are explained in Chapter 2.

Examples

```
PRINT A:B:
PRINT C:D:
PRINT E:F
```

Prints the current value of A, B, C, D, E and F contiguously across the output page, each value concatenated to the next.

PRINT Using Output Formatting (Continued)

```
PRINT A*100,Z
```

Prints the value of A*100 starting at column position 1; prints the value of Z on the same line, starting at column position 18 (the first tab position).

```
PRINT "ENTER NAME":
```

Prints the text "ENTER NAME" but does not execute a carriage return or line feed.

***

```
PRINT " ",B
```

Prints the value of B starting at column position 18.

PRINTER

Purpose

The **PRINTER** statement selects either the user's terminal or the system printer for subsequent program output.

Syntax

```
PRINTER [ON] [OFF] [CLOSE]
```

Comments

When a **PRINTER ON** statement is issued, program output data (specified by subsequent PRINT, HEADING, FOOTING or PAGE statements) is not printed immediately, unless immediate printing is forced via an SP-ASSIGN statement with option I or N specified. Instead, the data is stored in the spooler and printed when the program terminates, or when a PRINTER CLOSE statement is issued.

The **PRINTER OFF** statement directs subsequent program output to the terminal.

Once executed, a PRINTER ON or a PRINTER OFF statement remains in effect until another PRINTER (ON or OFF) statement is executed. If neither a PRINTER OFF nor a PRINTER ON statement has been executed, output is to the user's terminal.

The **PRINTER CLOSE** statement causes all data currently stored in the spooler to be printed immediately. PRINTER CLOSE applies only to output directed to the line printer.

Examples

```
PRINTER ON
PRINT A
PRINTER OFF
PRINT B
```

Displays value of variable B on the terminal immediately. The value of variable A is printed when the program ends.

***

```
PRINTER ON
PRINT A
PRINTER CLOSE
PRINTER OFF
PRINT B
```

Prints value of variable A immediately. Value of variable is then displayed on the terminal.

PRINTERR

Purpose

PRINTERR prints error messages stored in the system ERRMSG file or in a user-specified file without exiting DATA/BASIC.

Syntax

PRINTERR x {FROM file-var}

Comments

The value **x** is any expression that evaluates to a dynamic array where each parameter is an error message element, with the first as the item-id.

Error message libraries can be kept separate from the system ERRMSG file and can be used by PRINTERR. If the **FROM** clause is used, PRINTERR prints the error messages stored in **file-var**.

Examples

```
OPEN "MYERRS" TO ERRORFILE ELSE STOP 201,"MYERRS"
.
.
X = "511":VM:PARAMETER
PRINTERR X FROM ERRORFILE
```

Prints error message 511 found in the user-specified file ERRORFILE.

***

```
PRINTERR "201":AM:"TESTFILE"
```

Prints message [201]'TESTFILE' IS NOT A FILE NAME found in the system ERRMSG file.

PROCREAD

Purpose The **PROCREAD** statement reads data from the PROC primary input buffer.

Syntax PROCREAD variable ELSE statement(s)

Comments PROCREAD assigns the string value of the PROC primary input buffer to the **variable**.

If the program has not been run from a PROC, the **ELSE** clause is executed.

This command is particularly useful if attribute marks are used as delimiters in the PROC, because then the variable can be manipulated as a dynamic array.

Example

```
PROCREAD BUF ELSE
   PRINT "MUST EXECUTE FROM PROC"
   STOP
   END
```

Assigns string value of PROC primary input buffer to variable BUF. Message is printed if program was not executed from a PROC.

PROCWRITE

Purpose

The **PROCWRITE** statement writes data to the PROC primary input buffer.

Syntax

```
PROCWRITE expression
```

Comments

PROCWRITE writes the string value of the **expression** to the PROC primary input buffer.

If the program was not executed from a PROC, an error message is displayed.

This command is especially useful if attribute marks are used as delimiters in the PROC, because then the expression can be a dynamic array.

Example

```
BUF = "LIST":AM:FILE
PROCWRITE BUF
```

Writes dynamic array BUF to the PROC primary input buffer. If the program was not executed from a PROC, no operation takes place.

PROMPT

Purpose

The **PROMPT** statement selects the character which is used to prompt the user for input.

Syntax

```
PROMPT expression
```

Comments

The value of **expression** becomes the input prompt character. If expression has more than one character, only the first character is used.

Once a PROMPT statement has been executed, it remains in effect until another PROMPT statement is issued.

If no PROMPT statement has been issued, the INPUT statement uses a question mark (?) as the default prompt character.

Examples

```
PROMPT "@"
```

Specifies that the character @ will be used as the prompt character for subsequent INPUT statements.

***

```
PROMPT 5*5
```

Selects the digit 2 as the prompt character, (i.e., 5*5=25 and only the first character is used).

***

```
PROMPT A
```

Specifies that the current value of A will be used as the prompt character.

READ

Purpose

The **READ** statement reads a file item and assigns its value, as a dynamic array, to a variable.

Syntax

```
READ variable FROM {file-var,} item-id THEN/ELSE
```

Comments

The READ statement reads the file item specified by **item-id** and assigns its string value to the first **variable**.

If **file-var** is used, the item is read from the file previously assigned to the file variable via an OPEN statement. If filename is omitted, the internal default file variable is used (i.e., the file most recently opened without a file variable).

If the item-id does not exist, the **ELSE** clause is executed.

If the specified file has not been opened prior to the READ statement, the program aborts with an error message.

Examples

```
READ X1 FROM W,"TEMP" ELSE PRINT ...
 "NON-EXISTENT"; STOP
```

Reads the item named TEMP from the file opened and assigned to file variable W and assigns its string value to variable X1. If TEMP does not exist, the message NON-EXISTENT displays and the program terminates.

***

```
A="TEST"
B="1"
READ X FROM C,(A CAT B) ELSE STOP
```

Reads item TEST1 from file opened and assigned to file variable C and assigns its value to variable X. Program terminates if TEST1 does not exist.

***

```
READ Z FROM "Q1" THEN PRINT X; STOP
```

Reads item Q1 from the file opened without a file variable and assigns its value to variable Z. Prints value of X and terminates program if Q1 does not exist.

READLIST

Purpose

The **READLIST** statement reads a list from the POINTER-FILE and assigns it to a variable for program manipulation.

Syntax

```
READLIST variable FROM list {SETTING var}
 THEN/ELSE
```

Comments

Each item-id in **list** is separated by attribute marks.

READLIST allows a DATA/BASIC program to read in a saved list, place the contents in a **variable**, and then modify or search its contents.

If you specify the **SETTING** clause, the list is assigned to a select list specified by **var**.

Examples

```
READLIST A FROM ITEMX ELSE STOP
```

Reads the list in ITEMX into variable A.

***

```
READLIST LIST1 FROM X ELSE ...
 PRINT "CANNOT READ"; STOP
```

Reads the list from POINTER-FILE X into variable LIST1. If the list cannot be read, an error message displays and the program terminates.

***

```
READLIST LIST1 FROM '60-90' ELSE GOTO 100
READLIST LIST2 FROM 'OVER-90' ELSE GOTO 100
LIST3 = LIST1:AM:LIST2  ; * Combine the lists
WRITELIST LIST3 ON 'OVER-60'
.
.
GETLIST 'OVER-60' TO ALIST SETTING ACOUNT THEN
  FOR I = 1 TO ACOUNT
    READNEXT ID FROM ALIST ELSE PRINT 'ERROR'
    READ AITEM FROM AFILE,ID ELSE AITEM = ''
    .
    .
  NEXT I
END
```

Reads the lists saved in 60-90 and OVER-90 and combines them into a new list. The new list is then written back to the POINTER-FILE using the WRITELIST statement. This new list can then be manipulated by the program.

READNEXT

Purpose

The **READNEXT** statement reads the next item-id from the select list.

Syntax

```
READNEXT variable{,variable} {FROM select-var}
 THEN/ELSE
```

Comments

READNEXT reads the next item-id and assigns its string value to the first **variable**.

The second **variable** may be used when an exploding sort has been processed using BY-EXP or BY-EXP-DSND in a SELECT, SSELECT, GET-LIST or FORM-LIST statement issued at TCL or in a PROC.

The second variable is assigned the value count indicating the position of the multivalue within the attribute specified after the BY-EXP or BY-EXP-DSND connective.

This value can be used in a dynamic array extraction after the item has been read (via a READ statement) to obtain multivalues in exploded sort order.

If the **FROM** clause is specified, the item-id is read from the list assigned to **select-var**.

If the FROM clause is omitted, the item-id is read from the last SELECT, SELECTE or GETLIST statement executed without a TO clause. If none of these statements have been executed, the item-id comes from an externally-generated list executed at TCL immediately before running the program. (Such lists can be generated by a SELECT, BSELECT, SSELECT, FORM-LIST, GET-LIST, SEARCH or ESEARCH.)

If the list of item-ids has been exhausted, or if no selection has been performed, the ELSE clause is executed.

Note: READNEXT retrieves only item-ids from a list. In order to retrieve other values from a list, a READ statement must be executed following the READNEXT.

The READNEXT statement always returns a 0 or 1, depending on whether or not it can assign an item-id from the select list. Therefore, READNEXT can also be used as the conditional in a LOOP statement. (Refer to the LOOP statement for more information.)

READNEXT (Continued)

Examples

```
READNEXT ID ELSE STOP
```

Assigns value of next item-id from the default select variable to ID. If the list is exhausted, or if no SELECT, GET-LIST or SELECTE statement has been executed, and there is no external select list, the program terminates.

***

```
READNEXT VAR1 ELSE PRINT "CANNOT READ"; GO 10
```

Reads next item-id and assigns its string value to VAR1. If the list of item-ids has been exhausted, or if no selections have been performed (either in the program, at TCL or in a PROC), the message CANNOT READ is displayed and control transfers to statement 10.

***

```
FOR X=1 TO 10
  READNEXT B(X) ELSE STOP
NEXT X
```

Reads the next ten item-ids and assigns the values to matrix elements B(1) through B(10).

***

```
FOR I=1 TO 999
  READNEXT ITEM.ID, VALUE ELSE STOP
  READ ITEM FROM INV,ITEM.ID ELSE NULL
  PRINT ITEM<I, VALUE>
NEXT I
```

Reads and prints multivalues in exploded sort order.

***

```
LOOP WHILE READNEXT ID FROM SV DO
  PRINT ID
REPEAT
```

Prints all the item-ids in the list SV. The program terminates when the list is exhausted.

READT

Purpose

The **READT** statement reads the next record from a magnetic tape unit. (For more information on tape handling, refer to the manual titled, Using the Magnetic Tape System.)

Syntax

```
READT variable THEN/ELSE
```

Comments

READT reads the next record and assigns its value to **variable**.

If the tape unit has not been attached, or if an end-of-file mark (EOF) is read, the **ELSE** clause is executed.

DATA/BASIC attempts to read a label (if present) on the first READT command, on the first READT after a rewind or after sensing an EOF.

The maximum record size that can be read is set by the T-ATT verb. For example, T-ATT 2048 only allows reads of 2048 bytes.

The minimum size record that can be read is 20 bytes.

Examples

```
READT X ELSE PRINT "CANNOT READ"; STOP
```

Reads the next tape record and assigns it to X. If an EOF is detected or the tape is not attached, CANNOT READ is printed and the program terminates.

***

```
READT B ELSE
   PRINT "NO GOOD"
   GOTO 5
END
```

Reads next tape record and assigns its value to B. If it cannot be read, the message NO GOOD is printed and control transfers to statement 5.

READU

Purpose

READU locks a specific item in a file prior to updating it.

Syntax

READU variable FROM {file-var,} item-id, {LOCKED statement(s)} THEN/ELSE

Comments

READU works the same way as the READ statement, except that it locks the item to be updated. This prevents an item from being updated by two or more items simultaneously.

Other processors which encounter an item lock are suspended until the item becomes unlocked, unless the optional LOCKED clause is specified. If it is, the statements following LOCKED are executed.

The item can be unlocked in any of the following ways:

The process which has the item locked completes its update.

The program is terminated.

A RELEASE statement is issued.

The item is written back to the file with a WRITE statement (without the optional U).

The number of item locks is user-defined.

Examples

```
READU ITEM FROM INV, "S5" ELSE GOSUB 4
```

Locks item S5, then reads S5 to variable ITEM. If S5 does not exist, control is transferred to subroutine 4.

***

```
READU A FROM FILE1,"REC", LOCKED PRINT ...
 "LOCKED" ELSE STOP
```

Locks item REC in file FILE1, then reads it to variable A. If REC cannot be read, an error message is displayed and the program terminates.

READV

Purpose

The **READV** statement reads an attribute value from an item and assigns its string value to a specified variable.

Syntax

```
READV variable FROM {file-var,} item-id, attr#
 THEN/ELSE
```

Comments

READV reads the attribute specified by **attr#** from the **item-id** and assigns it to **variable**.

If a **file variable** is used, the attribute is read from the file previously assigned to that file variable via an OPEN statement.

If the file variable is omitted, the internal default file variable is used (i.e., the file most recently opened without a file variable).

If the specified item does not exist, the ELSE clause is executed.

Examples

```
READV A FROM F, "XYZ", 3 ELSE STOP
```

Reads the third attribute of item XYZ in the file specified by F and assigns it to variable A. If XYZ does not exist, the program terminates.

***

```
READV X FROM A, "TEST", 5 ELSE
  PRINT ERR
  GOTO 70
  END
```

Reads the fifth attribute of item TEST (in the file opened and assigned to variable A) and assigns its value to variable X. If TEST does not exist, then the value of ERR is printed and controls transfers to statement 70.

READVU

Purpose

READVU locks a specific item in a file prior to updating it.

Syntax

```
READVU variable FROM {file-var,} item-id, attr#
 {LOCKED statement(s)} THEN/ELSE
```

Comments

READVU works the same way as the READV statement, except that it locks the item to be updated. This prevents an item from being updated by two or more items simultaneously.

Other processors which encounter an item lock are suspended until the item becomes unlocked, unless the optional LOCKED clause is specified. If it is, the statements following LOCKED are executed.

The item can be unlocked in any of the following ways:

The process which has the item locked completes its update.

The program is terminated.

A RELEASE statement is issued.

The item is written back to the file with a WRITEV statement (without the optional U).

Note: The number of item locks is user-defined.

Example

```
READVU ATTR FROM B, "RECORD", 6
   LOCKED PRINT "UNABLE TO ACCESS"
   ELSE STOP
```

Locks item RECORD in file B. Reads attribute 6 of RECORD to variable ATTR or, if RECORD does not exist, an error message is displayed and the program terminates.

RELEASE

Purpose

The **RELEASE** statement unlocks items that have been locked for update.

Syntax

RELEASE {{file-variable,} item-id}

Comments

Individual items can be unlocked by specifying a **file-variable** and **item-id.**

If a file variable is specified, it represents the file previously assigned to that file variable via an OPEN statement.

If a file variable is not specified, the internal default file variable is used (i.e., the file most recently opened without a file variable).

If neither a filename nor an item-id is specified, RELEASE unlocks all items locked by the program.

Examples

```
RELEASE
```

Unlocks all items that have been locked by the program.

***

```
RELEASE F1,"RECORD"
```

Unlocks the item RECORD in the file specified by F1.

REM

Purpose

The **REM** (REMARK) statement lets you place comments anywhere in a program without affecting program execution.

Syntax

```
REM
*
!
```

Comments

To place a comment in a program, type the letters REM, an asterisk (*) or an exclamation point (!) at the beginning of the statement, followed by the text of your comment.

If a comment does not fit on one line, you must use two REM statements.

Note: Do not follow a REM with an equal sign (=) unless the entire string that follows is enclosed in quotes.

Note: If you use an exclamation point (!) at the beginning of a comment line (or all by itself) it causes a line of asterisks to be printed if the program is listed using the BLIST verb. If it follows another DATA/BASIC statement, it is treated as a normal comment line.

Examples

```
REM These DATA/BASIC statements
REM do not affect program execution.
```

In this case, two REM statements were necessary to complete the comment.

***

```
IF Y < 2 GO 10 ; ! Transfers control back to start
```

Clarifies the operation.

***

```
GOSUB 30
.
.
30 * SUBROUTINE TO PRINT RESULTS
PRINTER ON
PRINT ...
```

Identifies the subroutine at statement label 30.

RETURN

Purpose

The **RETURN** statement transfers control from a subroutine back to the main program.

Syntax

RETURN {TO statement-label}

Comments

RETURN transfers control from a subroutine back to the statement immediately following the GOSUB statement that called it.

RETURN TO transfers control from the subroutine to the statement with the specified **statement-label**.

If the statement label does not exist, an error message is displayed at compile time.

Every subroutine must return to the calling program by using a RETURN or RETURN TO statement, not a GOTO statement. This will ensure proper flow control.

Examples

```
   GOSUB 15
   PRINT "BACK FROM SUBROUTINE"
   .
   .
15 * SUBROUTINE XYZ
   .
   .
   RETURN
```

Control returns to the PRINT statement following the GOSUB that originally called the subroutine.

***

```
10 GOSUB 50
   PRINT "SUBROUTINE EXECUTION COMPLETE"
   .
   .
50 * SUBROUTINE HERE
   .
   .
   IF ERROR RETURN TO 99
75 RETURN
99 PRINT "ERROR ENCOUNTERED HERE"; STOP
```

If an error occurs in the subroutine, control transfers to label 99, an error message is displayed and the program terminates. Otherwise, control is returned to the statement following the GOSUB that called the subroutine.

REWIND

Purpose	The **REWIND** statement rewinds the magnetic tape unit to the Beginning-Of-Tape mark (BOT).
Syntax	REWIND THEN/ELSE
Comments	If the tape unit has not been attached, the **ELSE** clause is executed.
Example	REWIND ELSE STOP Tape is rewound to BOT. If tape unit is not attached, the program terminates.

RQM

Purpose

The **RQM** (SLEEP) statement causes a program to sleep for a specified period of time, terminating the program's current timeslice.

Syntax

[RQM] [SLEEP] {expression}

Comments

Expression may be either the number of seconds to sleep or a wakeup time specified in 24-hour format. If expression is omitted, the default is one second.

If 24-hour format is used, the wakeup time must be enclosed in quotes.

Examples

```
* PROGRAM SEGMENT TO SOUND TERMINAL BELL
* FIVE TIMES
*
EQU BELL TO CHAR(7)
FOR I=1 TO 5
  PRINT BELL:
  RQM
NEXT I
END
```

Sounds terminal bell five times, pausing long enough so bell is heard as five discrete beeps.

***

```
SLEEP "12:11"
```

Causes the program to sleep until 12:11 PM.

***

```
RQM 10
```

Causes program to sleep for 10 seconds.

***

```
X = "13:22:56"
SLEEP X
```

Program will sleep until 1:22:56 PM.

SELECT

Purpose

SELECT builds a list of item-ids for the READNEXT statement.

Syntax

```
SELECT [{file-var}] [{variable}] {TO select-var}

SELECTE {TO select-var}
```

Comments

If **file-var** is used, the list of item-ids is created for the file previously opened to that file variable via the OPEN statement.

If file variable is omitted, the internal default file variable is used, (i.e., the file most recently opened without a file variable).

When the **TO** clause is specified, the select list is assigned to a special type of variable called a select variable. **Select-var** is used in the FROM clause of a READNEXT statement to access the item-ids from that list.

If a normal string **variable** is specified, rather than a file variable, each attribute, value or subvalue in the string becomes an item-id in the list.

The **SELECTE** statement selects a list generated externally by any of the following: SELECT, BSELECT, SEARCH, ESEARCH, SSELECT, FORM-LIST or GET-LIST command executed at TCL. If SELECTE is used, it must be executed before any other SELECT or READNEXT statements.

Note: Any file or variable may be selected any number of times and used independently. This can be done to have several pointers into a list at the same time.

DATA/BASIC selects item-ids one group at a time as needed, rather than all at once as in ENGLISH. Therefore, if you change an item's id, it could be selected again. For this reason, if you are adding items or changing item-ids, you should perform an ENGLISH SELECT statement prior to executing the program.

SELECT (Continued)

Examples

```
    OPEN 'BP' ELSE STOP
    SELECT
10  READNEXT ID ELSE STOP
    PRINT ID
    GOTO 10
```

Selects BP as the default file variable and assigns the first item-id in BP (in hash order) to ID. Loops back to statement 10 until all the item-ids found in the file BP are printed.

***

```
OPEN 'CUST' TO CUSTF ELSE STOP 201
.
.
SELECT CUSTF TO CUSTLIST
READNEXT ID FROM CUSTF THEN ...
 FOUND = 1 ELSE FOUND = 0
```

Selects the CUST file and assigns it to CUSTLIST. Assigns the first item-id from the CUST file (in hash order) to ID.

***

```
SELECTE TO EXTERNAL
READNEXT ID FROM EXTERNAL ELSE ...
 PRINT 'NO ':ID; GO 10
END
```

Assigns the external select list to EXTERNAL. Assigns the first item-id from the external select list to ID.

***

```
X = 'B':VM:'C':VM:'D':VM:'E1':VM:'E2':VM:'E3'

ATTR4 = X<4>
SELECT ATTR4 TO VMLIST
READNEXT ID FROM VMLIST THEN
   READ ITEM FROM CUSTF,ID ELSE ITEM=''
END ELSE ITEM=''
```

Assigns the list E1, E2, E3 to VMLIST. Assigns the string E1 to ID.

SHARE

Purpose

The **SHARE** statement allows multiple programs to share a single copy of constant data.

Syntax

SHARE variable WITH list-name {account-name}

Comments

Variable is a simple variable or a subscripted array element and **list-name** is the name of the cataloged item. **Account-name** can be used to share data that was cataloged from a different account.

Allowing programs to share data makes them more efficient. Less workspace is used, which results in fewer frame faults and increased performance.

SHARE is typically used to allow programs to share copies of tables, item-lists, parameters and compiled screen definition items. Only data that remains constant may be shared.

Data that is to be shared must first be cataloged via the SHARE verb. This places a pointer in the form account-name*c*list-name in the system POINTER-FILE.

The syntax of the SHARE verb is:

```
   SHARE filename item-id
```

Examples

```
:SHARE TABLES TAX.RATE <RETURN>
[241] 'TAX.RATE' CATALOGED.
.
.
.
DIM TABLE(3)
NAME='TAX.RATE GARY'
SHARE TABLE(2) WITH NAME
DAY.NUM=DATE()-ICONV("1JAN":
  FIELD(ICONV(TIMEDATE(),
  "D2")," ",3),"D")+1
PRINT TABLE(2)<DAY.NUM>
END
```

Item TAX.RATE in file TABLES is cataloged with the SHARE verb. Array TABLE is dimensioned. TAX.RATE on GARY is assigned to NAME, and the table is shared as TABLE(2). Calculates the day number and prints the corresponding tax rate.

***

SHARE (Continued)

```
:SSELECT INV WITH QTY < "50" <RETURN>
20 ITEMS SELECTED.

:SAVE-LIST LOW.INV <RETURN>
[241] 'LOW.INV' CATALOGED.

:COPY-LIST LOW.INV <RETURN>
:TO (DICT INV)

:SHARE DICT INV LOW.INV <RETURN>
[241] 'LOW.INV' CATALOGED
```

```
SHARE LIST WITH "LOW.INV"
FOR I=1 TO 9999 WHILE LIST # ""
  IF LIST<I> = "" THEN STOP
  PRINT I,LIST<I>
NEXT I
END
```

The SSELECT statement forms the item-list. SAVE-LIST saves the item-list, which is then copied to the file item with the COPY-LIST verb. The item-list is cataloged with the SHARE verb. The subsequent DATA/BASIC program assigns the item-list to variable LIST. This program prints sequential numbers followed by their corresponding item-ids.

SLEEP

Purpose

The **SLEEP** statement is identical to the RQM statement. For information regarding the SLEEP statement, please refer to the explanation of the RQM statement.

STOP

Purpose

The **STOP** statement halts execution of a DATA/BASIC program.

Syntax

STOP {message-id {,expression...}}

Comments

The optional message clause displays messages from the system ERRMSG file or a message you create. (See Example 2.)

Message-id is an expression containing the item-id of a message found in the system ERRMSG file. This message is printed when the STOP is executed.

Expression may contain variables, functions, arithmetic statements or literal strings used as parameters to be printed in the message. They are processed on a first-in first-out basis.

The following codes determine the format of the messages:

C	Clear screen.
H literal string	Print literal string.
L	Output carriage return/line feed.
L(n)	Output n carriage returns/ line feeds.
E {literal string}	Enclose message-id in brackets. Follow with optional literal string.
A	Output next parameter.
A(n)	Output next parameter left-justified in a field of n blanks.
R(n)	Output next parameter right-justified in a field of n blanks.
S(n)	Output n spaces, counting from the beginning of the line.
T	Print system time.
D	Print system date.

Notes: Carriage returns/line feeds are not processed automatically, so you must state them explicitly within the ERRMSG item.

Messages supplied with the system may change from time to time with different software releases, so it is better to create your own.

STOP (Continued)

Examples

```
A=50; B=750; C=235; D=1300
REVENUE = A+B; COST = C+D
PROFIT = REVENUE - COST
IF PROFIT > 1 THEN GOTO 100
PRINT "ZERO PROFIT OR LOSS"
STOP
PRINT "POSITIVE PROFIT"
```

If PROFIT is less than or equal to 1, ZERO PROFIT OR LOSS is printed and program execution stops; otherwise, POSITIVE PROFIT is printed and the program continues.

***

```
OPEN "INV" TO INV
   ELSE STOP "T11","INV"
.
.
ERR="T12"
FOR I=1 TO 10
  ITEM.ID = "A*":I
  READ ITEM FROM INV, ITEM.ID
    ELSE STOP ERR,ITEM.ID
  WRITE ITEM ON TEST,I
NEXT I
END
```

Item T11 in ERRMSG file / **Item T12 in ERRMSG file**

```
L (2)                        L (2)
E COULD NOT FIND             H***
A                            H ITEM
H FILE!                      R (5)
L                            H DOES NOT EXIST!
H     CHECK FILE             L
H DEFINITION!                D
                             L
                             T
```

If INV does not exist, the following message is displayed:

```
[T11] COULD NOT FIND INV FILE!
      CHECK FILE DEFINITION!
```

If item A*7 is not present, this message is displayed:

```
*** ITEM A*T DOES NOT EXIST!
12 NOV 1986
11:35:32
```

SUB

Purpose

The **SUB** (or **SUBROUTINE**) statement identifies a DATA/BASIC program as an external subroutine called by another DATA/BASIC program.

Syntax

```
SUB{ROUTINE} name {(argument-list)}
```

Comments

A DATA/BASIC CALL statement transfers control to a cataloged subroutine **name**.

The **argument-list** consists of one or more expressions, separated by commas, that represent the values passed to the subroutine. The number of parameters passed from the CALL statement to the SUBROUTINE statement must match. If not, an error message displays and the program enters the debugger.

An external subroutine must contain a SUBROUTINE statement, a RETURN statement and an END statement. SUBROUTINE must be the first statement in the program.

Only arguments and COMMON variables can be passed between the calling program and the subroutine.

GOSUB and RETURN combinations may be used in a subroutine. If a RETURN is executed and there is no corresponding GOSUB statement, the program returns control to the statement following CALL in the calling program.

If a STOP, CHAIN or ENTER statement is executed before the subroutine's END statement, control never returns to the calling program.

The CHAIN statement should not be used to chain from an external subroutine to another DATA/BASIC program.

The ENTER statement should not be used to execute a SUBROUTINE.

A calling program and the corresponding subroutine must have the same precision and both must be cataloged.

SUB (Continued)

Examples

```
CALL REPORT
.
.
SUBROUTINE REPORT
```

Called subroutine REPORT has no parameters.

***

```
CALL ADD (A+2,F,395)
.
.
SUBROUTINE ADD (X,Y,Z)
```

Subroutine ADD returns three values.

***

```
CALL VENDOR (NAME, ADDRESS, NUMBER)
.
.
SUBROUTINE VENDOR (NAME, ADDR, NUM)
```

Subroutine VENDOR accepts and returns three values.

UNLOCK

Purpose The **UNLOCK** statement resets execution locks.

Syntax

```
UNLOCK {expression}
```

Comments **Expression** specifies which lock to reset. If expression is omitted, all execution locks previously set by the program are reset.

A warning message is displayed if you try to unlock an execution lock which the program did not lock.

All execution locks set by a program are automatically reset when the program terminates.

The TCL verb CLEAR-BASIC-LOCKS can be used to reset all 256 execution locks. This verb is present on the SYSPROG account.

Examples

```
UNLOCK
```

Resets all execution locks previously set by the program.

***

```
UNLOCK 63
```

Resets execution lock 63.

***

```
UNLOCK (5+A)*(B-2)
```

Resets the execution lock specified by the value of the expression (5+A)*(B-2).

WEOF

Purpose

The **WEOF** statement writes an End-Of-File mark (EOF) to the tape.

Syntax

```
WEOF THEN/ELSE
```

Comments

If the tape unit has not been attached, the ELSE clause is taken.

Examples

```
WEOF ELSE STOP
```

Writes an EOF mark. If the tape unit is not attached, the program terminates.

***

```
WEOF THEN GOTO 100 ELSE
   PRINT "TAPE NOT ATTACHED"
   STOP
   END
```

Writes an EOF mark and transfers control to statement 100. If tape is not attached, displays message and terminates program.

WRITE

Purpose

The **WRITE** statement updates a file item.

Syntax

WRITE expression ON {file-var,} item-id

Comments

WRITE replaces the contents of the specified **item** with the string value (or dynamic array) of **expression.**

If **file-var** is used, it specifies the file previously assigned to that variable via an OPEN statement. If file-var is omitted, the internal default file variable is used (i.e., the file most recently opened without a file variable).

If the item-id does not exist, a new item is created.

If a file is update protected and security codes do not match, an error message is displayed and control transfers to the debugger.

Note: The WRITE statement does not delete trailing attribute marks before filing an item. If you wish to delete trailing attribute marks, use the TRIM function.

Examples

```
WRITE "XXX" ON A, "ITEM5"
```

Replaces the current contents of ITEM5 in the file opened and assigned to variable A with string value XXX.

***

```
A="123456789"
B="X55"
WRITE A ON FN1,B
```

Replaces the current contents of item X55 in the file opened and assigned to variable FN1 with string value 123456789.

***

```
WRITE 100*5 ON "EXP"
```

Replaces the current contents of item EXP in the file most recently opened without a file variable with string value 500.

WRITELIST

Purpose

The **WRITELIST** statement writes a string to the POINTER-FILE as a saved list.

Syntax

```
WRITELIST string ON sav-list
```

Comments

Each attribute, value or subvalue in the **string** is used as an item-id when the **sav-list** is later used.

Lists produced by a WRITELIST statement can be used later in the same program by executing a GETLIST or a READLIST statement.

Examples

```
READLIST LISTA FROM X
WRITELIST LISTA ON "ITEM"
```

Reads list from X and assigns it to variable LISTA. Writes contents of LISTA to the POINTER-FILE called ITEM.

***

```
EQU AM TO CHAR(254)
READLIST LIST1 FROM '15-40' ELSE...
READLIST LIST2 FROM 'OVER.40' ELSE...
LIST3 = LIST1:AM:LIST2
X = 'OVER.15'
WRITELIST LIST3 ON X
PRINT LIST3
```

Reads LIST1 and LIST2 and combines them into one list (LIST3). LIST3 is then written to the POINTER-FILE as a saved list, and its contents are displayed on the terminal.

***

```
A = "ITEM1":AM:"ITEM2":AM:"ITEM3"
WRITELIST A ON "ITEMS"
```

```
:GET-LIST ITEMS

3 ITEMS SELECTED
>
```

Writes contents of A to POINTER-FILE called ITEMS. The list can later be retrieved with a GET-LIST command issued at TCL or with a GETLIST or READLIST statement executed in the same program.

WRITET

Purpose **WRITET** writes a record to tape.

Syntax WRITET expression THEN/ELSE

Comments The string value of **expression** is written as the next record on the tape.

If the tape unit has not been attached, or if the string value of expression is a null string, the **ELSE** clause is executed.

DATA/BASIC writes a label when the first WRITET statement is executed or when a WRITET is executed after a rewind or after an EOF has been written.

The T option on the RUN verb inhibits writing the tape label to ensure compatibility with previous releases and other devices.

The maximum record size for systems with 32K bytes or more of main memory is 8192 bytes. For systems with less than 32K bytes of memory, maximum record size is 4096.

Given the above constraints, the maximum record size that can be written to tape can be controlled by the T-ATT verb. For example, T-ATT 2048 sets the maximum record size written to 2048.

When writing, records are padded to 20 bytes with segment marks. They are deleted when read.

Examples

```
WRITET A+5 THEN PRINT "WRITTEN TO TAPE" ELSE STOP
```

Writes the value of A+5 as the next record on tape and prints WRITTEN TO TAPE on the terminal. If the tape unit is not attached or if A+5 is a null value, the program terminates.

***

```
FOR I=1 TO 5
  WRITET A(I) ELSE STOP
NEXT I
```

Writes the values of array elements A(1) through A(5) onto 5 tape records. If one of the array elements has a value of '' (null) or if the tape unit is not attached, the program terminates.

WRITEU

Purpose

The **WRITEU** statement functions just like the WRITE statement, except that it leaves a previously locked item locked at the end of the write.

Syntax

WRITEU expression ON {file-var,} item-id

Comments

Note: WRITEU does not actually lock an item. It simply does not unlock an item that is already locked.

Examples

WRITEU "XYZ" ON FILEA, ITEM4

Replaces the current contents of ITEM4 in FILEA with the string XYZ. If ITEM4 was locked previously, it remains locked after the write.

***

WRITEU 20*4 ON "REC"

Replaces the current contents of item REC in the file most recently opened without a file variable with string value 80. If REC was already locked, it remains locked after the write.

WRITEV

Purpose

The **WRITEV** statement updates an attribute value in a file.

Syntax

```
WRITEV expression ON {file-var,} item-id, attr#
```

Comments

WRITEV replaces the value of **attr#** in the **item-id** with the value of **expression**.

If **file-var** is specified, it represents the file previously assigned to that variable via an OPEN statement. If file-var is omitted, the internal default file variable is used (i.e., the file most recently opened without a file variable).

If the specified item or attribute is nonexistent, a new item or attribute is created.

If the specified file has not been opened prior to executing the WRITEV, or if an attribute of less than one is specified, an error message displays and the program terminates.

Note: WRITEV does not delete trailing attribute marks before filing an item. To delete trailing attribute marks, use the TRIM function.

Examples

```
X1 = "XXX"
WRITEV X1 ON A2, "ABC", 4
```

Replaces the fourth attribute of item ABC in the file opened and assigned to variable A2 with the string value XXX.

***

```
Z=2
Y="THIS IS A TEST"
WRITEV Y ON X, "PROG", Z+3
```

Replaces attribute 5 of item PROG in the file opened and assigned to variable X with the string value "THIS IS A TEST".

***

```
WRITEV "XYZ" ON "A7", 4
```

Replaces attribute 4 of item A7 in the file most recently opened without a file variable with the string value XYZ.

WRITEVU

Purpose

WRITEVU works the same as the WRITEV statement, except that it leaves a previously locked item locked at the end of the write.

Syntax

```
WRITEVU expression ON {file-var,} item-id, attr#
```

Comments

Note: WRITEVU does not actually lock an item. It simply does not unlock an item that is already locked.

Examples

```
Z=2
A="TESTING PROGRAM A"
WRITEVU A ON FIL2, "PROG", Z+1
```

Replaces attribute 3 of item PROG in file FIL2 with the string value TESTING PROGRAM A. If PROG was locked before this write, it remains locked afterward.

***

```
WRITEVU "ABC" ON "ITEMA", 8
```

Replaces attribute 8 of ITEMA in the file most recently opened without a file variable with the string value ABC. If ITEMA was already locked, it remains locked after the write.

Chapter 5
DATA/BASIC Intrinsic Functions

Overview

This chapter contains a brief summary of the DATA/BASIC functions, grouped together by their logical function. Following the summary is a complete description of each function in alphabetical order.

String/ Substring Manipulation

CHANGE	Replaces a substring with a new string.
CHECKSUM	Returns the positional checksum of the specified string.
COL1	Returns the numeric value of the column position immediately preceding the substring specified by the FIELD function.
COL2	Returns the numeric value of the column position immediately following the substring specified by the FIELD function.
COUNT	Counts the number of times a substring occurs within a string.
DCOUNT	Counts the number of elements in a string, which are separated by a specified delimiter.
FIELD	Returns a substring from within the specified string.
INDEX	Searches a string for a specified substring and returns the starting column position of that substring.
LEN	Determines the length of a string.
SPACE	Generates a string value containing a specified number of blank spaces.
STR	Generates a string value containing a specified number of occurrences of a string.
TRIM	Removes unnecessary blank spaces from a specified string.

Math Functions	COS	Calculates the cosine of an angle.
	EXP	Raises 'e' to a specified value.
	LN	Calculates logarithms to base e.
	MOD	Calculates the modulo of two expressions.
	PWR	Calculates a variable raised to a power.
	REM	Calculates the modulo of two expressions.
	SIN	Calculates the sine of an angle.
	SQRT	Calculates the square root of an expression.
	TAN	Calculates the tangent of an angle.
Format Conversions	ASCII	Converts a string value from EBCDIC to ASCII.
	CHAR	Converts a specified numeric value to its corresponding ASCII character string value.
	EBCDIC	Converts a string value from ASCII to EBCDIC.
	SEQ	Converts an ASCII character to its corresponding numeric value.
Time and Date	DATE	Returns a string value containing the internal system date.
	TIME	Returns the internal time of day.
	TIMEDATE	Returns the current time and date in external format.
I/O Conversion	ICONV	Performs input conversions like those used in ENGLISH.
	OCONV	Performs output conversions like those used in ENGLISH.
Numeric Capabilities	ABS	Generates the absolute (positive) numeric value of an expression.

	INT	Returns an integer value.
	RND	Returns a random number.
Logical Capabilities	ALPHA	Searches for alphabetic characters in a string.
	NOT	Returns the logical inverse of the specified expression.
	NUM	Determines the data type of the specified expression.
Bit Manipulation	BITCHANGE	Toggles the state of the specified bit and returns the value of the bit before it was changed.
	BITCHECK	Returns the current value of the specified bit.
	BITLOAD	Assigns values to the entire bit table or retrieves the current value of the entire table.
	BITRESET	Resets the value of the specified bit to 0 and returns the value of the bit before it was changed.
	BITSET	Sets the value of the specified bit to 1 and returns the value of the bit before it was changed.
Manipulating Dynamic Array Elements	DELETE	Deletes an attribute, value or subvalue from a dynamic array.
	EXTRACT	Extracts an attribute, value or subvalue from a dynamic array.
	INSERT	Inserts an attribute, value or subvalue into a dynamic array.
	MAXIMUM	Returns the maximum numeric element in a specified dynamic array.
	MINIMUM	Returns the minimum numeric element found in the specified dynamic array.
	REPLACE	Replaces an attribute, value or subvalue in a dynamic array.
	SUMMATION	Returns the sum of all elements of a dynamic array.

Miscellaneous Functions	@	Sets the cursor to a specified position on the terminal or printer. @ is also used to generate video effects characters.
	DQUOTE	Returns a specified string surrounded by double quotes.
	GETMSG	Retrieves messages installed in the system denationalization language tables.
	GROUP	Performs group extractions with any delimiter specified as the group separator.
	SPOOLER	Returns spooler status information.
	SQUOTE	Returns the specified string enclosed in single quotes.
	SYSTEM	Retrieves the current state of various system elements.
	UNASSIGNED	Determines whether or not a value is assigned to a variable.

@

Purpose

The @ function used with the PRINT statement sets the cursor to a specified position on the terminal or printer and used with the CRT statement sets the cursor to a specified position on the terminal. The @ function is also be used to generate video effects characters.

Syntax

```
@(column-exp{,line-exp})
@(-exp)
```

Comments

@(column-exp) sets the cursor to the column specified by the expression, on the current line.

@(-exp) generates an extended cursor addressing code or a video effects code.

Line-exp specifies a different line number on which to position the cursor.

Values of the expressions must be within the row and column limits defined by page width and depth set by the most recent TERM verb.

The left-most column is numbered column 0; the top line is numbered line 0. Therefore, if the terminal screen is 80 columns wide, the columns are numbered 0 thru 79.

The @ function may not be used to format spooled printer output.

The @ function may appear anywhere that a legal expression is allowed, including assignment statements. For example:

```
START = @(10,1)
.
.
PRINT START:"text..."
```

If a Matrix printer is being used, the matrix parameter must be set to 1 via the TERM verb. Rows and columns start at zero on the terminal and one on the Matrix printer. Matrix converts zero addresses to one.

Extended Cursor Addressing

In order to provide a consistent way of emitting terminal-independent character strings (clear-to-EOL, etc.), DATA/BASIC supports an extended cursor addressing process. The @ function allows a single, negative parameter which returns the correct cursor control string for the terminal defined by TERMTYPE. The returned string is based on the following parameters:

@ (Continued)

-1	Clear screen sequence.
-2	Cursor-home sequence.
-3	Clear-to-end-of-screen sequence.
-4	Clear-to-end-of-line sequence.
-5	Reserved.
-6	Reserved.
-7	Reserved.
-8	Reserved.
-9	Cursor-back sequence.
-10	Cursor-up sequence.
-11	Cursor on.
-12	Cursor off.
-13	Status line on.
-14	Status line off.
-15	Cursor forward.
-16	Cursor down.
-17	Slave port on.
-18	Slave port off.
-19	Screen dump.

The clear screen sequence includes the number of pad characters specified in the FF DELAY field of the TERM setting.

Different terminal types are indicated by changing the TERMINAL parameter (#8) of the TERM statement. The following types are defined.

0	PRISM I and II
1	Microdata Scribe
2	ADDS Viewpoint
3	Invalid. (This setting should be used when an unknown or hardcopy device is being used. This setting causes all cursor control requests to be sent as a carriage return/line feed.)
4	PRISM IV and V
5	Visual VT52

These settings also indicate whether or not the terminal handles visual mode characters; Types 4 and 5 are set to YES. Terminal parameter #8 should be set to 4 for PRISM 7 terminals.

Example

```
HEAD = @(35,0):"MAIN MENU":@(35,1):STR("-",9)
PRINT @(-1),HEAD
```

Clears the screen and prints heading at the top of screen starting at the 35th column.

@ (Continued)

Video Effects Table 5.1 lists the video effects that can be achieved using the @ function.

These codes send an eight-bit character to the terminal. In order to receive an eight-bit character, the PCI SETTING (TERMINAL Parameter #10) must be set to 78. The normal configuration is a setting of 74 for a seven-bit character.

The video effects can also be achieved by using the code (without the minus sign) as the value in a CHAR() function. PRINT CHAR(130) is the same as PRINT @(-130). The same IS NOT true for extended cursor addressing codes.

All of the video effects shown in Table 5-1 may not work will all terminals. For instance, the PRISM 9 does not provide for Dimmed and Blanked Video while the PRISM 4 and PRISM 7 do not provide for Bold Video. Consult the appropriate terminal documentation for complete information.

Table 5-1. Video Effects Codes

-128					
-129					Dimmed
-130				Flashing	
-131				Flashing	Dimmed
-132			Reversed		
-133			Reversed		Dimmed
-134			Reversed	Flashing	
-135			Reversed	Flashing	Dimmed
-136		Blanked			
-137		Blanked			Dimmed
-138		Blanked		Flashing	
-139		Blanked		Flashing	Dimmed
-140		Blanked	Reversed		
-141		Blanked	Reversed		Dimmed
-142		Blanked	Reversed	Flashing	
-143		Blanked	Reversed	Flashing	Dimmed
-144	Underlined				
-145	Underlined				Dimmed
-146	Underlined			Flashing	
-147	Underlined			Flashing	Dimmed
-148	Underlined		Reversed		
-149	Underlined		Reversed		Dimmed
-150	Underlined		Reversed	Flashing	
-151	Underlined		Reversed	Flashing	Dimmed
-152	Underlined	Blanked			
-153	Underlined	Blanked			Dimmed

@ (Continued)

Table 5-1. Video Effects Codes (Continued)

-154		Underlined	Blanked		Flashing	
-155		Underlined	Blanked		Flashing	Dimmed
-156		Underlined	Blanked	Reversed		
-157		Underlined	Blanked	Reversed		Dimmed
-158		Underlined	Blanked	Reversed	Flashing	
-159		Underlined	Blanked	Reversed	Flashing	Dimmed
-160	Bold					
-161	Bold					Dimmed
-162	Bold				Flashing	
-163	Bold				Flashing	Dimmed
-164	Bold			Reversed		
-165	Bold			Reversed		Dimmed
-166	Bold			Reversed	Flashing	
-167	Bold			Reversed	Flashing	Dimmed
-168	Bold		Blanked			
-169	Bold		Blanked			Dimmed
-170	Bold		Blanked		Flashing	
-171	Bold		Blanked		Flashing	Dimmed
-172	Bold		Blanked	Reversed		
-173	Bold		Blanked	Reversed		Dimmed
-174	Bold		Blanked	Reversed	Flashing	
-175	Bold		Blanked	Reversed	Flashing	Dimmed
-176	Bold	Underlined				
-177	Bold	Underlined				Dimmed
-178	Bold	Underlined			Flashing	
-179	Bold	Underlined			Flashing	Dimmed
-180	Bold	Underlined		Reversed		
-181	Bold	Underlined		Reversed		Dimmed
-182	Bold	Underlined		Reversed	Flashing	
-183	Bold	Underlined		Reversed	Flashing	Dimmed
-184	Bold	Underlined	Blanked			
-185	Bold	Underlined	Blanked			Dimmed
-186	Bold	Underlined	Blanked		Flashing	
-187	Bold	Underlined	Blanked		Flashing	Dimmed
-188	Bold	Underlined	Blanked	Reversed		
-189	Bold	Underlined	Blanked	Reversed		Dimmed
-190	Bold	Underlined	Blanked	Reversed	Flashing	
-191	Bold	Underlined	Blanked	Reversed	Flashing	Dimmed

ABS

Purpose The **ABS** function generates the absolute (positive) numeric value of an expression.

Syntax ABS(expression)

Comment Any valid DATA/BASIC **expression** is acceptable.

Examples

```
A = 100
B = 25
C = ABS(B-A)
```

Assigns the value 75 to variable C.

***

```
A = ABS(Q)
```

Assigns the absolute value of variable Q to variable A.

***

```
X = 600
Y = ABS(X-1000)
```

Assigns the value 400 to variable Y.

ALPHA

Purpose The **ALPHA** intrinsic function looks for alphabetic characters in a string.

Syntax ALPHA(expression)

Comments Any valid DATA/BASIC **expression** is acceptable.

ALPHA returns a 1 (true) if expression is a string containing only upper and/or lower case alphabetic characters.

Examples

```
X = "123 WINDSOR AVE."
IF ALPHA(X) THEN PRINT "OKAY" ELSE GO 99
```

Returns a 0, because string X contains numeric values. Control transfers to statement 99.

***

```
CITY = "London"
IF ALPHA(CITY) THEN PRINT "ALL LETTERS" ELSE STOP
```

Returns a 1 and prints ALL LETTERS, because CITY contains only upper and lower case alphabetic characters.

ASCII

Purpose

The **ASCII** function converts a string value from EBCDIC to ASCII.

Syntax

```
ASCII(expression)
```

Comments

Expression can be any valid DATA/BASIC expression.

This function is useful when reading EBCDIC format tapes with the READT statement.

Examples

```
READT X ELSE STOP
Y = ASCII(X)
```

Reads a record from the magnetic tape unit and assigns the value to variable X. Assigns ASCII value of record to variable Y.

***

```
A = "Z"
B = ASCII(A)
PRINT B
```

Assigns the ASCII value of variable A to variable B and prints B.

BITCHANGE

Purpose

The **BITCHANGE** function toggles the state of the specified bit and returns the value of the bit before it was changed.

Syntax

```
BITCHANGE(expression)
```

Comments

Expression specifies the bit to be changed.

BITCHANGE operates on a table of 128 bits, numbered 1 to 128, unique to each process. Each bit is available as a two-state flag (i.e., the value returned by the function is always either zero or one.)

Because BITCHANGE returns the value of the bit before it was changed, the function of checking-and-setting (or resetting) a flag can be accomplished in one step.

Special functions can be performed by setting expression to the following values:

-1	Allows access to the system INHIBIT bit (which controls inhibiting of the BREAK key).
-2	Allows access to the system TRSTRFLG bit (which controls TCL restart).
-3	Allows access to the system RSTRTFLG bit (which controls BREAK-END restart).

Examples

If bit 100 = 0,

```
X = BITCHANGE(100)
PRINT X
```

Sets bit 100 to 1 and prints 0.

***

If bit 68 = 1,

```
A = BITCHANGE(68)
PRINT A
```

Sets bit 68 to 0 and prints 1.

BITCHECK

Purpose

The **BITCHECK** intrinsic function returns the current value of the specified bit.

Syntax

```
BITCHECK(expression)
```

Comments

Expression specifies the bit to be checked.

BITCHECK operates on a table of 128 bits, numbered 1 to 128, unique to each process. Each bit is available as a two-state flag (i.e., the value returned by the function is always either zero or one.)

Special functions can be performed by setting expression to the following values:

-1	Allows access to the system INHIBIT bit (which controls inhibiting of the BREAK key).
-2	Allows access to the system TRSTRFLG bit (which controls TCL restart).
-3	Allows access to the system RSTRTFLG bit (which controls BREAK-END restart).

Example

```
Y = BITCHECK(76)
PRINT Y
```

If the value of bit 76 was 0, then a 0 is printed. If the value was 1, then 1 is printed.

BITLOAD

Purpose

The **BITLOAD** function assigns values to the entire bit table or retrieves the current value of the entire table.

Syntax

BITLOAD({expression})

Comments

BITLOAD operates on a table of 128 bits, numbered 1 to 128, unique to each process. Each bit is available as a two-state flag (i.e., the value returned by the function is always either zero or one.)

Expression is an ASCII string representing a hexadecimal value. It is used as a bit pattern to assign values to the table from left to right. Assignment stops when the string runs out or when a non-hex character is encountered. If the string defines less than 128 bits, the remaining bits in the table are reset.

If expression is omitted or evaluates to null, an ASCII hex character string is returned, which defines the value of the table. Any trailing zeros in the string are truncated.

Examples

```
X = "AB233FA22AD498BA12348A"
A = BITLOAD(X)
```

Loads the bit table with the value of ASCII hex string X. The contents of the bit table will be:

```
1010 1011 0010 0011 0011 1111 1010 0010
0010 1010 1101 0100 1001 1000 1011 1010
0001 0010 0011 0100 1000 1010 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
```

Note the remianing zeroes (reset bits). Only 88 of the 128 bits were set because only 22 characters were contained in the input expression.

***

```
TABLE = BITLOAD()
```

Loads variable TABLE with the hexadecimal value of the bit table.

BITRESET

Purpose

The **BITRESET** function resets the value of the specified bit to 0 and returns the value of the bit before it was changed.

Syntax

```
BITRESET(expression)
```

Comments

Expression specifies the bit to be reset. If expression evaluates to zero, all elements in the table are cleared and the returned value is zero.

BITRESET operates on a table of 128 bits, numbered 1 to 128, unique to each process. Each bit is available as a two-state flag (i.e., the value returned by the function is either zero or one.)

Because BITRESET returns the value of the bit before it was changed, the function of checking-and-setting (or resetting) a flag can be accomplished in one step.

Special functions can be performed by setting expression to the following values:

0	Resets the entire table.
-1	Allows access to the system INHIBIT bit (which controls inhibiting of the BREAK key).
-2	Allows access to the system TRSTRFLG bit (which controls TCL restart).
-3	Allows access to the system RSTRTFLG bit (which controls BREAK-END restart).

Examples

If bit 126 = 0

```
A = BITRESET(126)
PRINT A
```

Returns a value of 0 and resets bit 126 to 0. Prints 0.

***

If bit 113 = 1

```
VALUE = BITRESET(113)
PRINT VALUE
```

Returns a value of 1 and resets bit 113 to 0. Prints 1.

BITSET

Purpose

The **BITSET** function sets the value of the specified bit to 1 and returns the value of the bit before it was changed.

Syntax

```
BITSET(expression)
```

Comments

Expression specifies the bit to be set. If expression evaluates to zero, all elements in the table are cleared and the returned value is zero.

BITSET operates on a table of 128 bits, numbered 1 to 128, unique to each process. Each bit is available as a two-state flag (i.e., the value returned by the function is either zero or one.)

Because BITSET returns the value of the bit before it was changed, the function of checking-and-setting (or resetting) a flag can be accomplished in one step.

Special functions can be performed by setting expression to the following values:

0	Sets the entire table.
-1	Allows access to the system INHIBIT bit (which controls inhibiting of the BREAK key).
-2	Allows access to the system TRSTRFLG bit (which controls TCL restart).
-3	Allows access to the system RSTRTFLG bit (which controls BREAK-END restart).

Examples

If bit 10 = 1

```
STATUS = BITSET(10)
PRINT STATUS
```

Sets the bit to 1 and prints 1.

***

If bit 48 = 0

```
J = BITSET(48)
PRINT J
```

Sets bit 48 to 1 and prints 0.

CHANGE

Purpose

The **CHANGE** intrinsic function lets you replace a substring with a new substring.

Syntax

```
new.str=CHANGE(old.str, old.substr, new.substr)
```

Comments

All occurrences of **old.substr** in **old.str** are replaced by **new.substr**. The result is **new.str**.

If old.substr is null, old.str is returned with no changes.

Examples

```
OLDSTR = "IRVINE, CA 92714"
A = "92714"
B = "92720"
NEWSTR = CHANGE(OLDSTR, A,B)
```

Replaces substring A (92714) in OLDSTR with substring B (92720) and assigns the new value to NEWSTR. NEWSTR contains IRVINE, CA 92720.

***

```
X = "31AA42BB53AAA"
Y = "A"
Z = "C"
NEWVALUE = CHANGE(X,Y,Z)
```

Changes the value of 31AA42BB53AAA to 31CC42BB53CCC.

CHAR

Purpose The **CHAR** function converts the specified numeric value to its corresponding ASCII character string value.

Syntax CHAR(expression)

Comment **Expression** can be any valid DATA/BASIC expression.

Examples

```
VM = CHAR(253)
```

Assigns the string value of a value mark to variable VM.

***

```
X = 252
SVM = CHAR(X)
```

Assigns the ASCII string value for a subvalue mark to variable SVM.

CHECKSUM

Purpose

The **CHECKSUM** intrinsic function returns a number equal to the positional checksum of the specified string.

Syntax

```
CHECKSUM(expression)
```

Comment

Expression can be any valid DATA/BASIC expression.

Examples

```
CKSUM = CHECKSUM("123")
```

Returns the positional checksum of string 123.

***

```
Y = CHECKSUM(X)
PRINT Y
```

Prints the positional checksum of string X.

COL1/COL2

Purpose

The **COL1** and **COL2** functions return the numeric values of the column positions immediately preceding and following the substring specified by the FIELD function.

Syntax

```
COL1()
COL2()
```

Comments

COL1() returns the numeric value of the column position immediately preceding the substring selected by the most recently executed FIELD function.

COL2() returns the numeric value of the column position immediately following the substring selected by the most recently executed FIELD function.

If a FIELD function has not been executed in a program before executing a COL1() or COL2() function, a warning message displays and COL1() or COL2() is assigned a value of zero.

If the most recent FIELD function delimiter is null, then COL2() = COL1() + 1. If the specified delimiter cannot be found in the string, COL2() returns the number of characters in the string plus one.

Examples

```
Q = FIELD("ABCBA","B",2)
R = COL1()
S = COL2()
```

Assigns the string value C to variable Q, the numeric value 2 to variable R and the numeric value 4 to variable S.

***

```
X = "K"
A = FIELD("123K456K789",X,2)
B = COL1()
PRINT B
C = COL2()
PRINT C
```

Assigns the string value of 456 to variable A. Assigns the numeric values of 4 to variable B and 8 to variable C. Prints B and C.

COS

Purpose **COS** calculates the cosine of an angle.

Syntax COS(expression)

Comments **Expression** must be stated in degrees. The relation between radians and degrees is:

2 Pi radians = 360 degrees

Examples

```
A = 35
B = COS(A)
PRINT B
```

Calculates and prints the cosine of 35 degrees. B = 0.8192

***

```
X = 42
SECANT = COS(X) / SIN(X)
```

Calculates the secant of 4 degrees. SECANT = 1.1105.

COUNT

Purpose

The **COUNT** function counts the number of times a substring occurs within a string.

Syntax

```
COUNT(string,substring)
```

Comments

If the specified **substring** is not contained in the **string,** a value of zero is returned.

If the specified substring is null, the value returned is equal to the length of the string.

Substrings may overlap in the string. For example, substring AA occurs three times in string AAAA.

Examples

```
M = "SMITH, JOHN"
X = COUNT(M,"H")
PRINT X
```

Prints 2, because there are two occurrences of the letter H in string M.

***

```
A = "714-854-8388"
B = "8"
C = COUNT(A,B)
```

Assigns a value of 4 to variable C, because substring B occurs in string A four times.

DATE

Purpose The **DATE** function returns a string value containing the internal system date.

Syntax

```
DATE()
```

Comments The internal date represents the number of days since December 31, 1967.

Internal dates are converted to external form with the ENGLISH 'D' conversion.

Examples

```
Q = DATE()
PRINT Q
```

Assigns the string value of internal date to variable Q and prints Q.

***

```
WRITET DATE() ELSE STOP
```

Writes the string value of the current internal date onto a magnetic tape record. If the tape unit is not attached, the program terminates.

DCOUNT

Purpose

The **DCOUNT** function counts the number of elements in a string, which are separated by a specified delimiter.

Syntax

```
DCOUNT(string,delimiter)
```

Comments

Because DCOUNT returns the number of elements in the **string** that are surrounded by the specified **delimiter**, it can be used to count the number of elements in a dynamic array.

If the delimiter is not contained in the string, a value of 1 is returned.

If the string is null, a value of 0 is returned.

If either string or delimeter is a literal, it must be enclosed in quotes.

Examples

```
EQU AM TO CHAR(254)
A = "123":AM:"456":AM:"789"
B = DCOUNT(A,AM)
```

Assigns a value of 3 to variable B, because there are three elements separated by attribute marks in string A.

***

```
X = ""
M = DCOUNT(X,"/")
```

Assigns a value of 0 to variable M, because X is a null string.

***

```
A = "THIS.IS.A.TEST"
B = DCOUNT(A,":")
```

Assigns a value of 1 to variable B, because string A does not contain the specified delimiter.

DELETE

Purpose

The **DELETE** function deletes an attribute, value or subvalue from a dynamic array. DELETE has been replaced by the DEL statement but is maintained for compatibility.

Syntax

```
DELETE (expression,attr#,value#,subvalue#)
```

Comments

Expression specifies the dynamic array.

If you specify a nonzero value for **attr#** and a zero for both the value and subvalue, the entire attribute is deleted.

If you specify a nonzero value for both attr# and **value#** and a zero for the subvalue, then the specified value is deleted.

If you specify nonzero values for attr#, value# and **subvalue#**, then the specified subvalue is deleted.

Note: Refer to the DEL statement in Chapter 4 for information regarding invalid and illogical dynamic array references.

Examples

```
OPEN '','INVENTORY' ELSE STOP
READ VALUE FROM 'ITEM2' ELSE STOP
VALUE = DELETE(VALUE,1,2,3)
WRITE VALUE ON 'ITEM2'
```

Deletes the third subvalue of the second value of the first attribute in item ITEM2 in the INVENTORY file. The delimiter associated with the third subvalue is also deleted.

***

```
OPEN 'TEST' ELSE STOP
READ X FROM 'NAME' ELSE STOP
WRITE DELETE(X,2,0,0) ON 'NAME'
```

Deletes the second attribute and its associated delimiter in item NAME in file TEST.

***

DELETE (Continued)

```
X = "ABC]123^DEF]456^GHI]789"
A = "1"
B = "0"
PRINT DELETE(X,A,A-B,B)
```

Deletes value 1 of attribute 1 in dynamic array X and prints the result.

DQUOTE

Purpose

The **DQUOTE** intrinsic function returns a specified string surrounded by double quotes.

Syntax

```
DQUOTE(expression)
```

Comment

Expression can be any valid DATA/BASIC expression.

Examples

```
R = 'X.25 Emulation'
A = DQUOTE(R)
PRINT A
```

Prints "X.25 Emulation".

***

```
X = "TESTING DQUOTE FUNCTION"
Y = DQUOTE(X)
```

Assigns the string value "TESTING DQUOTE FUNCTION" to variable Y.

EBCDIC

Purpose **EBCDIC** converts a string value from ASCII to EBCDIC.

Syntax EBCDIC(expression)

Comments **Expression** can be any valid DATA/BASIC expression.

This function is useful when writing EBCDIC format tapes with the WRITET statement.

Examples

```
X = "!"
Y = EBCDIC(X)
PRINT Y
```

Prints the EBCDIC value of variable X, which is a Z.

***

```
B = EBCDIC(A)
WRITET B ELSE STOP
```

Converts the value of string A from ASCII to EBCDIC and assigns it to variable B. The contents of B are then written to tape.

EXP

Purpose The **EXP** function raises 'e' to a specified value.

Syntax EXP(expression)

Comments **Expression** can be any valid DATA/BASIC expression.

No indication is made if overflow occurs.

Examples

```
A = "32"
B = EXP(A)
PRINT B
```

Prints the value of e raised to 32.

***

```
Z = LN(X)
Y = EXP(Z)
```

Assigns the anti-log of Z to variable Y.

EXTRACT

Purpose

EXTRACT returns an attribute, value or subvalue from a dynamic array. The EXTRACT function has been replaced by the dynamic array reference function described in Chapter 3 but is maintained here for compatibility.

Syntax

```
EXTRACT(expression,attr#,value#,subvalue#)
```

Comments

Expression specifies the dynamic array.

If you specify a nonzero value for **attr#** and a zero for both the value and subvalue, the entire attribute is retrieved.

If you specify a nonzero value for both attr# and **value#** and a zero for the subvalue, then the specified value is retrieved.

If you specify nonzero values for attr#, value# and **subvalue#**, then the specified subvalue is retrieved.

Note: Refer to Chapter 3, "DATA/BASIC Arrays" for information regarding invalid and illogical dynamic array references.

Examples

```
OPEN '',"TEST" ELSE STOP
READ X FROM 'NAME' ELSE STOP
PRINT EXTRACT(X,3,2,0)
```

Prints the second value in attribute three of item NAME in file TEST.

***

```
OPEN 'ACCOUNT' ELSE STOP
READ ITEM1 FROM 'ITEM1' ELSE STOP
IF EXTRACT(ITEM1,3,2,1)=25 THEN PRINT "MATCH"
```

Prints MATCH if the first subvalue in the second value in attribute three in ITEM1 equals 25.

FIELD

Purpose

The **FIELD** intrinsic function returns a substring from within a string.

Syntax

```
FIELD(string,delimiter,occurrence)
```

Comments

FIELD returns the substring immediately preceding the specified **delimiter** in the specified **string**. **Occurrence** specifies which substring to return.

If occurrence = 1, FIELD returns the substring starting from the beginning of the string up to the first occurrence of the delimiter.

If occurrence = 2, the substring surrounded by the first and second occurrences of delimiter is returned.

Examples

```
T = "12345.67891.98765"
G = FIELD(T,".",1)
```

Returns the substring 12345 to variable G.

***

```
A=FIELD("XXX:YYY:ZZZ:555",":",3)
```

Assigns the value ZZZ to variable A.

***

```
X = "77$ABC$XX"
Y = "$"
Z = "ABC"
IF FIELD(X,Y,2)=Z THEN STOP
```

Terminates program because the FIELD function returns the value of ABC which is equal to Z.

GETMSG

Purpose

The **GETMSG** function retrieves messages installed in the system denationalization language tables.

Syntax

```
GETMSG(class-number,message-number)
```

Comments

Class-number is the number of the message class in the denationalization language table.

Message-number is the number of the message within the specified class in the denationalization language table.

This function is used with denationalization. (For more information, refer to "Appendix G., Using Denationalization from DATA/BASIC".)

Example

```
PRINT GETMSG(1,1)
```

Prints message 1 in class 1 of the denationalization language table.

GROUP

Purpose

The **GROUP** intrinsic function is similar to the group extract conversion (option G of the OCONV function) except that it allows any delimiter to be specified as the group separator, including system delimiters.

Syntax

```
GROUP(string,delimiter,start.group,return.group)
```

Comments

Delimiter is the character that separates each group (element) in **string**.

Start.group is the number of the first group to return.

Note: This is different than the OCONV function, which specifies the number of leading groups to skip.

Return.group specifies the number of groups to return. If the number of groups to be returned is greater than the number of groups in the string, the specified groups are returned until the string is exhausted.

Examples

```
CITY = "IRVINE/ORANGE/TUSTIN"
Y = GROUP(CITY,"/",3,2)
```

Returns the value TUSTIN, because it is the third element in string CITY (start.group = 3). Only one element is returned because the string has been exhausted.

***

```
EQU AM TO CHAR(254)
A = "123":AM:"456":AM:"789":AM:"123"
X = GROUP(A,AM,1,2)
```

Returns the value 123^456, because they are the first two elements in string A.

***

```
B = "A/B/C/D/E"
Y = GROUP(B,"/",3,2)
```

Returns two groups, starting with the third group. Y now has the value C/D.

ICONV

Purpose The **ICONV** function performs input conversions just like those used in ENGLISH.

Syntax ICONV(expression, conversion)

Comments The **expression** to be converted may not contain any system delimiters.

Conversion may be any of the following:

D	Convert date to internal format. Date conversion to internal format can also be accomplished as an output conversion using the **DI** conversion.
MC	Perform character conversion. (See Appendix E for a list of mask conversions.)
MD	Convert decimal number to integer.
MF	Remove commas, decimal points and blanks from expression.
MP	Convert integer to packed decimal.
MT	Convert time to internal format.
MX	Convert hexadecimal to ASCII.
T	Convert by table/file translation. (Do not use if you need to access several items or attributes.)

These conversions are formed exactly like ENGLISH conversions and have the same capabilities.

The conversion must be specified as a string surrounded by quotes.

Examples

```
PENNIES = ICONV("1234.00","MD2")
```

Assigns the string value 123400 to the variable PENNIES.

***

```
IDATE = ICONV("7-29-85","D")
```

Assigns the internal date value 6420 to variable IDATE.

INDEX

Purpose

INDEX searches a string for a specified substring and returns the starting column position of that substring.

Syntax

```
INDEX(string,substring,occurrence)
```

Comments

INDEX searches for a specified **occurrence** of a **substring** contained in **string**.

If the substring is not found, a value of zero is returned.

If the substring is a null string, then the specified occurrence number is returned.

Examples

```
START = INDEX("ABCDEFGHI","DEF",1)
```

Assigns the value 4 to the variable START.

***

```
A = INDEX("CCXXCCXXCC","XX",2)
```

Assigns the value 7 to A, because the second occurrence of substring XX starts at column 7.

***

```
VAR = INDEX("ABC123","Z",1)
```

Assigns a value of 0 to VAR, because Z is not present in the string ABC123.

***

```
X = "1234ABC"
Y = "ABC"
IF INDEX(X,Y,1)=5 THEN GOTO 3
```

Transfers control to statement 3, because ABC starts at column position 5 of string X.

***

```
S = "X1XX1XX1XX"
FOR I=1 TO INDEX(S,"1",3)
.
NEXT I
```

Executes the loop 8 times, because the third occurrence of 1 appears in column 8 of string S.

INSERT

Purpose

The **INSERT** function inserts an attribute, value or subvalue into a dynamic array. INSERT has been replaced by the INS statement but is maintained for compatibility.

Syntax

```
INSERT(array,attr#,value#,subvalue#,expression)
```

Comments

If you specify a nonzero value for **attr#** and a zero for both the value and subvalue, the specified attribute is inserted into the specified dynamic **array**.

If you specify a nonzero value for both attr# and **value#** and a zero for the subvalue, then the specified value is inserted.

If you specify nonzero values for attr#, value# and **subvalue#**, then the specified subvalue is inserted.

Expression is the value to be inserted. It may not contain any system delimiters.

Note: If the attr#, value# or subvalue# contains a -1, the value specified by expression is inserted after the last attribute, value or subvalue indicated. Otherwise, expression is inserted before the specified attribute, value or subvalue.

Refer to the INS statement for information regarding invalid and illogical dynamic array references.

Examples

```
OPEN '','TEST-FILE' ELSE STOP
READ X FROM 'NAME' ELSE STOP
X = INSERT(X,10,0,0,'XXXXX')
WRITE X ON 'NAME'
```

Inserts the value XXXXX before attribute 10 of item NAME, creating a new attribute.

***

```
OPEN 'FN1' ELSE STOP
READ B FROM 'IT5' ELSE STOP
A = INSERT(B,-1,0,0,'EXAMPLE')
WRITE A ON 'IT5'
```

Inserts the string value EXAMPLE after the last attribute of item IT5 in file FN1.

INSERT (Continued)

```
Y=INSERT(X,3,2,0,"XYZ")
```

Inserts the string value XYZ before the second value of attribute 3 of dynamic array X and assigns the resulting array to variable Y.

***

```
A = "123456789"
B = INSERT(B,3,-1,0,A)
```

Inserts 123456789 after the last value of attribute 3 of dynamic array B.

***

```
Z = INSERT(W,5,1,1,"B")
```

Inserts the string value B before the first subvalue of the first value of attribute 5 in array W and assigns the resulting array to Z.

INT

Purpose The **INT** function returns an integer value.

Syntax INT(expression)

Comments INT returns the integer value of the specified **expression** (i.e., the fractional portion of expression is truncated).

If expression is a fraction, INT returns a value of 0.

Examples

```
I = 5/3
J = INT(I)
```

Assigns the value 1 to variable J.

***

```
PRINT INT(.25)
```

Prints the value 0, because expression is a fraction.

***

```
A = 3.65
B = 3.6
C = INT(A+B)
D = INT(A-B)
```

Assigns the value 7 to variable C and the value 0 to variable D.

LEN

Purpose

The **LEN** function determines the length of a string.

Syntax

LEN(expression)

Comments

LEN returns the numeric value of the length of the string specified by **expression.**

Examples

```
A = "1234ABC"
B = LEN(A)
```

Assigns the value 7 to variable B.

***

```
Q = LEN("123")
```

Assigns the value 3 to variable Q.

***

```
X = "123"
Y = "ABC"
Z = LEN(X:Y)
```

Assigns the value 6 (variable Y concatenated to variable X) to variable Z.

LN

Purpose

The **LN** function calculates logarithms to base e.

Syntax

LN(expression)

Comments

LN returns the natural logarithm of the specified **expression**.

If expression evaluates to a number less than or equal to zero, a warning message displays and a value of zero is returned.

Examples

```
Z = LN(X)
```

Assigns the natural log of variable X to variable Z.

***

```
F = "12"
G = "14"
A = LN(F-G)
PRINT A
```

Prints a warning message and assigns a zero to variable A, because the expression F-G evaluates to a negative number.

MAXIMUM

Purpose

The **MAXIMUM** intrinsic function returns the maximum numeric element in a specified dynamic array.

Syntax

```
MAXIMUM(array)
```

Comments

Array elements may contain nonnumeric values, which are ignored.

Null elements evaluate to a zero.

Array delimiters do not need to be the same character, but they must be either an attribute mark (AM), a value mark (VM) or a subvalue mark (SVM).

Examples

```
EQU AM TO CHAR(254)
A = "1":AM:"2":AM:"7":AM:"5"
B = MAXIMUM(A)
```

Assigns a 7, which is the maximum numeric value in array A, to variable B.

***

```
EQU AM TO CHAR(254)
EQU VM TO CHAR(253)
EQU SVM TO CHAR(252)
S="TEST":VM:"1":AM:"DATE":VM:"3":SVM:"14":SVM:"86"
T = MAXIMUM(S)
PRINT T
```

Prints the value 86, which is the maximum numeric value in string S.

***

```
EQU AM TO CHAR(254)
Q = "THIS":AM:"IS":AM:"FRIDAY"
P = MAXIMUM(Q)
```

No assignment is made, because there are no numeric values and all nonnumeric values are ignored.

***

```
F = "TESTING":AM:"MAXIMUM":AM:"":AM:"FUNCTION"
G = MAXIMUM(F)
```

Assigns a 0 to variable G, because a null value evaluates to a numeric zero.

MINIMUM

Purpose

The **MINIMUM** function returns the minimum numeric element found in the specified dynamic array.

Syntax

```
MINIMUM(array)
```

Comments

Nonnumeric elements of **array** are ignored.

Null elements evaluate to zero.

Array delimiters do not need to be the same character, but they must be either an attribute mark (AM), a value mark (VM) or a subvalue mark (SVM).

Examples

```
EQU AM TO CHAR(254)
ZIP = "92626":AM:"92714":AM:"92720"
X = MINIMUM(ZIP)
```

Assigns the value 92626 to variable X.

***

```
EQU AM TO CHAR(254)
EQU VM TO CHAR(253)
EQU SVM TO CHAR(252)
ARR = "4":AM:"3":VM:"67":VM:"9":SVM:"39":SVM:"11"
Y = MINIMUM(ARR)
PRINT Y
```

Prints the value 3, the minimum numeric value in dynamic array ARR.

***

```
W = "RECORD":AM:"":AM:"334"
X = MINIMUM(W)
PRINT X
```

Prints a zero, because null elements evaluate to zero.

***

```
A = "BOSTON":VM:"MA":AM:"SACRAMENTO":VM:"CA"
B = MINIMUM(A)
```

Nonnumeric elements are ignored, so no assignment is made.

MOD

Purpose

The **MOD** (REM) function calculates the modulo of two expressions.

Syntax

```
MOD(exp1,exp2)
REM(exp1,exp2)
```

Comments

The MOD (or REM) statement returns the modulo of the specified **expressions**.

The second expression cannot be zero.

Because MOD returns a zero when exp1 is evenly divisible by exp2, this function is typically used to test whether or not a number is a multiple of another number.

MOD calculates the modulo in the following way:

```
   MOD(A,B) = A - (INT(A/B) * B)
```

Examples

```
IF MOD(INT(A),2) THEN PRINT 'ODD' ELSE
  PRINT 'EVEN'
  END
```

Determines if a number is odd or even and prints appropriate message.

***

```
INPUT NUM
IF REM(INT(NUM),3) THEN STOP ELSE GO 100
```

Tests to see if NUM is a multiple of 3. If it is, the program terminates; otherwise, control transfers to statement 100.

NOT

Purpose

The **NOT** function returns the logical inverse of the specified expression.

Syntax

```
NOT(expression)
```

Comments

NOT returns a value of true (1) if **expression** evaluates to 0.

NOT returns a value of false (0) if expression evaluates to a nonzero quantity.

Expression must evaluate to a numeric quantity or a numeric string.

Examples

```
X = NOT(X="")
```

Assigns the value 1 to variable X, because NOT(X="") is a true statement.

***

```
A = 1
B = 5
PRINT NOT(A AND B)
```

Prints a zero, because the logical inverse of A AND B evaluates to false.

***

```
IF NOT(X1) THEN STOP
```

Terminates program if current value of variable X1 is 0.

***

```
PRINT NOT(M)
```

Prints a value of 1 if the current value of variable M is 0 or a null string; otherwise, prints a zero. If M is non-numeric, an error message is returned.

***

```
PRINT NOT(5 LT 1)
```

Prints a value of 1.

NUM

Purpose

The **NUM** intrinsic function determines the data type of the specified expression.

Syntax

```
NUM(expression)
```

Comments

NUM returns a value of 1 (true) if **expression** evaluates to a number or a numeric string.

NUM returns a value of 0 (false) if expression evaluates to a nonnumeric string.

A null string is considered to be a numeric string.

A string containing a system delimiter is considered to be nonnumeric. Also, a string containing leading or trailing zeros is nonnumeric.

Examples

```
A1=NUM(123)
```

Assigns a value of 1 to variable A1.

***

```
A2=NUM("123")
```

Assigns a value of 1 to variable A2.

***

```
IF NOT(NUM(VALUE)) THEN PRINT "NON-NUMERIC DATA"
```

Prints the message NON-NUMERIC DATA if VALUE is not a number or a numeric string.

***

```
IF NUM(I CAT J) THEN GOTO 5
```

Transfers control to statement 5 if the values of both I and J are numbers or numeric strings.

OCONV

Purpose	The **OCONV** function performs output conversions just like those used in ENGLISH.
Syntax	OCONV(expression, conversion)

Comments

The **expression** to be converted may not contain any system delimiters.

Conversion may be any of the following:

D	Convert date to external format. (See Appendix F for a list of date conversions.)
G	Perform group extraction.
MC	Perform character conversion. (See Appendix E for a list of mask conversions.)
MD	Convert integer to decimal number.
MF	Perform literal insertions or currency formatting.
MP	Convert packed decimal number to integer.
MT	Convert time to external format.
MX	Convert ASCII to hexadecimal.
T	Convert by table/file translation. (Do not use if you need to access several items or attributes.)

These conversions are formed exactly like ENGLISH conversions and have the same capabilities.

The conversion must be specified as a string surrounded by quotes.

Examples

```
DOLLARS = OCONV("123400","MD2")
```

Assigns the string value 1234.00 to the variable DOLLARS.

***

```
PRICE = OCONV("3*179*7","G1*1")
```

Extracts 179 from the string and assigns it to PRICE.

OCONV (Continued)

```
A = "6240"
B = "D"
XDATE = OCONV(A,B)
```

Assigns the string value 29 JUL 1985 (the external date) to XDATE.

***

```
A = "A31-1"
Y = OCONV(A,"TINV;X;3;3")
PRINT Y
```

Retrieves and prints the third attribute of item A31-1 in the INV file.

PWR

Purpose

The **PWR** function calculates a variable raised to a power.

Syntax

```
PWR(exp1,exp2)
exp1^exp2
```

Comments

PWR raises the value of **exp1** to the value of **exp2**.

Negative values can be raised to negative or positive integer values, but a negative value raised to a noninteger value is undefined in DATA/BASIC.

If you try to raise a negative number to a noninteger value, a warning message displays and a value of zero is returned.

The PRECISION must fall in the range of 3 to 5 for the results to be meaningful.

Examples

```
PRINT PWR(2,5)
```

Prints the value 32 (2 to the fifth power).

***

```
X = -3
Y = -6
Z = PWR(X,Y)
```

Assigns the value 0.0014 to variable Z.

***

```
F = 4
G = 2^F
```

Assigns the value 16 to variable G (2 to the fourth).

REM

Purpose The **REM** intrinsic function is identical to the MOD function. For information about REM, please refer to the explanation of the MOD function.

REPLACE

Purpose

The **REPLACE** function replaces an attribute, value or subvalue in a dynamic array. The REPLACE function has been replaced by the dynamic array reference function described in Chapter 3 but is maintained here for compatibility.

Syntax

```
REPLACE(array,attr#,value#,subvalue#,expression)
```

Comments

If you specify a nonzero value for **attr#** and a zero for both the value and subvalue, then the entire attribute is replaced in **array**.

If you specify a nonzero value for both attr# and **value#** and a zero for the subvalue, then the specified value is replaced.

If you specify nonzero values for attr#, value# and **subvalue#**, then the specified subvalue is replaced.

Expression is the replacement value. It may not contain any system delimiters.

Note: If attr#, value# or subvalue# contain a -1, the replacement value specified by expression is inserted after the last attribute, value or subvalue indicated. Otherwise, expression is inserted before the specified attribute, value or subvalue.

Refer to Chapter 3, "DATA/BASIC Arrays" for information regarding invalid and illogical dynamic array references.

Examples

```
OPEN 'INVENTORY' ELSE STOP
READ X FROM 'NAME' ELSE STOP
X = REPLACE(X,4,0,0,'EXAMPLE')
WRITE X ON 'NAME'
```

Replaces attribute 4 of item NAME in file INVENTORY with the string value EXAMPLE.

***

```
OPEN '','XYZ' ELSE STOP
READ B FROM 'ABC' ELSE STOP
WRITE REPLACE(B,3,-1,0,'NEW VALUE') ON 'ABC'
```

Inserts the string value NEW VALUE after the last value of attribute 3 of item ABC in file XYZ.

REPLACE (Continued)

```
X=REPLACE(ARR,2,0,0,'')
```

Replaces attribute 2 of dynamic array ARR with a null string and assigns new array to variable X.

***

```
VALUE = "TEST STRING"
DA = REPLACE(DA,4,3,2,VALUE)
```

Replaces subvalue 2 of value 3 of attribute 4 in dynamic array DA with the string value TEST STRING and assigns the resulting array to DA.

***

```
X = "ABC123"
Y = REPLACE(Y,1,1,-1,X)
```

Inserts the value ABC123 after the last subvalue of value 1 of attribute 1 in dynamic array Y.

***

```
A=REPLACE(B,2,3,0,"XXX")
```

Replaces value 3 of attribute 2 of dynamic array B with the value XXX and assigns the resulting array to variable A.

RND

Purpose

RND returns a random number.

Syntax

```
RND(expression)
```

Comments

RND generates a random number between zero and the number specified by **expression** minus one.

Expression must be a positive number. If a negative number is used, it will be converted to the absolute value (the program performs an implicit ABS on the input argument).

Examples

```
Z = RND(11)
```

Assigns a random number between 0 and 10 (inclusive) to variable Z.

***

```
R = 100
Q = 50
B = RND(R+Q+1)
```

Assigns a random number between 0 and 150 (inclusive) to variable B.

***

```
Y = RND(-51)
```

Assigns a random number between 0 and 50 inclusive to variable Y.

SEQ

Purpose

The **SEQ** function converts an ASCII character to its corresponding numeric value.

Syntax

SEQ(expression)

Comments

SEQ converts the first character of the string value of **expression** to its corresponding numeric value.

SEQ is the inverse of the CHAR function.

Examples

```
PRINT SEQ('I')
```

Prints the number 73.

***

```
DIM C(50)
S = 'THE GOOSE FLIES SOUTH'
FOR I=1 TO LEN(S)
C(I) = SEQ(S[I,1])
NEXT I
```

Places the decimal equivalents of the characters in string S into vector C.

SIN

Purpose

The **SIN** function calculates the sine of an angle.

Syntax

```
SIN(expression)
```

Comments

Expression must be stated in degrees. The relation between radians and degrees is:

2 Pi radians = 360 degrees

Examples

```
DEG = 3.1416 * RAD/180
SINE = SIN(DEG)
```

Determines the sine of RAD radians.

***

```
A = 42
B = SIN(A)
PRINT B
```

Prints the sine of variable A.

SPACE

Purpose The **SPACE** function generates a string value containing a specified number of blank spaces.

Syntax

```
SPACE(expression)
```

Comments **Expression** specifies the number of blank spaces.

Use this function if creating more than 6 spaces. Otherwise, just enclose the spaces in quotes.

Examples

```
PRINT SPACE(10):"HELLO"
```

Prints 10 blanks followed by the word HELLO.

***

```
DIM M(10)
MAT M = SPACE(20)
```

Assigns a string consisting of 20 blanks to each of 10 elements of array M.

***

```
S = SPACE(7)
L = "SMITH"
C = ","
F = "JOHN"
N = L:S:C:S:F
```

Assigns the following string to variable N:

```
   "SMITH       ,       JOHN".
```

SPOOLER

Purpose The **SPOOLER** function returns spooler status information.

Syntax SPOOLER(X{,Y})

Comments **X** determines which spooler function to return. It must be a number from 1 to 4.

If X is less than 1 or greater than 4, a null string is returned.

Y specifies an account name or line number, depending on the value of X. If Y is omitted, a null value is used.

All returned information is in the form of a dynamic array.

Following is the information returned for each value of X.

X=1 (Y is ignored)	Returns the same information as the SP-STATUS menu. Each attribute corresponds to a separate device and returns the information as values in the following format:

```
Form Q name]Form type]Device assigned]Device
type]Device status]# entries for this
device]Page skip
```

X=2
Y={"account name"}
	Returns the same information as the SP-JOBS menu plus some password information. Each attribute corresponds to a separate job queue entry and returns the information as values in the following format:

```
Form Q name]Job #]Generating account]Generating
line number]Creation date]Current print status]
Current options]Total job size(frames)]#
copies to print{]password flag}
```

Note: The password is only returned if the account executing the program has SYS2 privileges. If the account has SYS1 or SYS0 privileges, a one is returned as the password flag. If the job entry does not require a password, no value is returned.

Spooler (Continued)

X=3
Y={line number}
	Returns the current spooler assignment information for the calling process or for the specified line number. Each attribute of the returned string is in the following multivalued format:

```
BASIC print file #]Form Q name]Options]Copies
```

Note: If a default print assignment is used, a null string is returned until at least one job has been queued for output.

X=4
Y={line number}
	Returns any currently open, queued jobs for the calling process or for the specified line number. The multivalued format of the returned variable is:

```
BASIC print file #]Spooler print file #]Current
size in frames
```

Note: The BASIC print file number is the number of the print file specified in the PRINT ON n statement in the program executing the print job. The spooler print file number is the queue entry number the spooler automatically assigns to a new job. This is the same number displayed in a SP-JOBS request and as HOLD ENTRY #n when a hold file is generated.

Examples

```
INFO = SPOOLER(1)
PRINT INFO
```

Returns the following sample information:

```
STANDARD}}1}LPTR}ASSIGNED}0}0
```

**

```
INFO = SPOOLER(2,SYSPROG)
```

Returns the following type of information:

```
STANDARD}0}SYSPROG}3}24 SEP}PRINT}}32}1}PSWD
```

SQRT

Purpose

The **SQRT** intrinsic function calculates the square root of an expression.

Syntax

```
SQRT(expression)
```

Comments

Expression must be greater than or equal to zero. An expression that evaluates to a negative number returns a zero, and a warning message is displayed.

The PRECISION must be in the range of 3 to 5.

Examples

```
X = "64"
Y = SQRT(X)
```

Assigns the value 8 (the square root of 64) to variable Y.

***

```
C = SQRT(A*A + B*B)
```

Finds the hypotenuse of a right triangle and assigns the value to variable C.

SQUOTE

Purpose The **SQUOTE** function returns the specified string enclosed in single quotes.

Syntax

```
SQUOTE(expression)
```

Comment Any valid DATA/BASIC **expression** is acceptable.

Examples

```
L = "JOHN SMITH"
M = SQUOTE(L)
```

Assigns the value 'JOHN SMITH' to variable M.

***

```
T = "THIS IS A TEST."
S = SQUOTE(T)
PRINT T
```

Prints the string 'THIS IS A TEST.'

STR

Purpose

STR generates a string value containing a specified number of occurrences of a string.

Syntax

```
STR(expl,exp2)
```

Comments

Expl specifies the character(s) from which to create the string. **Exp2** is the number of occurrences of expl to be included in the string.

If exp2 is less than zero, a null string is created.

Use the STR function if you are generating more than 9 characters; otherwise, just enclose the string in quotes.

Examples

```
S=STR('*',12)
```

Creates a string consisting of 12 asterisks (*) and assigns it to variable S.

***

```
PRINT STR('ABC',3)
```

Prints the string ABCABCABC.

***

```
VAR = STR("A",2)
A = "AAA"
C = VAR:A
```

Assigns the string value AAAAA to variable C.

***

```
N = STR("?%?",4)
```

Assigns the string value ?%??%??%??%? to variable N.

SUMMATION

Purpose	The **SUMMATION** function returns the sum of all elements of a dynamic array.
Syntax	SUMMATION(array)

Comments

SUMMATION returns the sum of all numeric elements found in the specified dynamic **array**.

Nonnumeric elements of array are ignored.

Null elements evaluate to zero.

Array delimiters do not need to be the same character, but they must be either an attribute mark (AM), a value mark (VM) or a subvalue mark (SVM).

Examples

```
D = 4]1]6^4^6]1
C = SUMMATION(D)
```

Returns the sum (22) of all the numeric elements of dynamic array D.

***

```
AGES = 22^34^25^57^54
A = SUMMATION(AGES)
```

Assigns the value 192 (the sum of all the ages) to variable A.

***

```
LIST = SMITH]JOHN^JONES]BILL^JENKINS]SAM
S = SUMMATION(LIST)
```

No operation is performed, because all nonnumeric values are ignored.

SYSTEM

Purpose	The **SYSTEM** function allows access to the current state of various system elements.
Syntax	SYSTEM(expression)
Comments	These values of **expression** return the following information:

1	Returns a 1 if the current PRINT statement destination is to the printer.
2	Returns the current page width as defined by the TERM statement.
3	Returns the current page length as defined by the TERM statement.
4	Returns the number of lines remaining to print on the current page, based on the parameters defined by the current TERM statement.
5	Returns the current page number.
6	Returns the current line count.
7	Returns the terminal type as defined in the TERM statement.
8	Not used; returns a zero.
9	Returns the current CPU millisecond count for the calling process.
10	Returns a 1 if the stack (STON) is currently enabled (i.e., stacked input is currently available).
11	Returns a 1 if an external list is currently active; otherwise, returns a zero. The list must have been generated by a SELECT, SSELECT, FORM-LIST or GET-LIST executed immediately prior to running the current program.
12	Returns the current system time in 1/10 second format.
13	Returns a zero.
14	Returns the type ahead count.

SYSTEM (Continued)

15 Returns the options specified as part of the last TCL statement in the following three attributes:

- a string of letters corresponding to the alpha options typed.
- the first numeric parameter.
- the second numeric parameter.

Attributes 2 and 3 appear only if they were specified at TCL.

Note: The internal variables containing the two numeric subroutines are used by some system subroutines. They should be retrieved early in the program to avoid getting incorrect values.

16 Returns the current level of a PERFORM statement. PERFORMs can be nested 32 levels deep.

17 Not used and reserved.

18 Returns the port number.

19 Returns the account name. This is retrieved directly from SYSTEM.

20 Returns a 1 if the program currently running is cataloged.

21 Returns the type of visual characteristics supported by the current TERMTYPE:

0	Invalid.
1	Video characteristics not supported.
2	Video character requires a CRT position.
3	CRT position is not required.

SYSTEM (Continued)

22	Returns the following system configuration parameters as a dynamic array. 1 System serial number 2 Firmware type 3 Firmware version number 4 Wordmate (1 if allowed; otherwise 0) 5 Number of ABS frames 6 Number of active lines 7 Spoolers line number 8 Maximum FID 9 Number of workspace frames
23	Returns a 0 if the BREAK key is enabled. Returns a 1 if the BREAK key is disabled by a DATA/BASIC statement. In this case, BREAK is reenabled automatically when the program is finished. Returns a 2 if BREAK is disabled from TCL. In this case, it cannot be reenabled from a DATA/BASIC program. Returns a 3 if the BREAK key has been disabled both by TCL and a DATA/BASIC statement.
24	Returns a 1 if character echoing is enabled.
25	Returns a 1 if this is a phantom process.
26	Returns the current prompt character.
27	Returns a 1 if running from a PROC.
28	Returns the system privilege level (0,1,2).
29	Returns the number of bytes in a standard system frame (500 or 1000).
30	Returns a 1 if pagination is in effect; 0 otherwise.
31	Reserved.
32	Reserved.
33	Reserved.
34	Reserved.

SYSTEM (Continued)

35 Returns the number of the language currently in use. (Denationalization)

36 Returns the number of the default collation table. (Denationalization)

37 Returns the thousands separator currently in use. (Denationalization)

38 Returns the decimal separator currently in use. (Denationalization)

39 Returns the money sign currently in use. (Denationalization)

40 Returns the name of the program currently executing.

41 Returns the release number of the operating system (e.g., the literal 2.2).

42 Returns Asynchronous Communication Port status byte according to the following bit pattern:

```
LSB Bit 0 = 1 if DTR raised
    Bit 1 = 1 if DCD raised
    Bit 2 = 1 if CTS raised
    Bit 3 = 1 if currently outputting
    Bit 4 = 1 if output buffer full
    Bit 5 - Not used
    Bit 6 - Not used
    Bit 7 - Not used
```

Examples

```
Y = SYSTEM(26)
PRINT Y
```

Prints the current prompt character (? for example).

***

```
A = SYSTEM(22)
B = EXTRACT(A,2,0,0)
PRINT B
```

Prints the value of the second attribute (firmware type) of the array containing the system configuration parameters.

TAN

Purpose

The **TAN** function calculates the tangent of an angle.

Syntax

```
TAN(expression)
```

Comments

Expression must be stated in degrees. The relation between radians and degrees is:

2 Pi radians = 360 degrees

Tangent is undefined for angles which are odd multiples of 90 degrees (90*1=90, 90*3=270, etc.). If the expression used is an odd multiple of 90, a warning message is displayed and a value of zero is returned.

Examples

```
ANGLE = 25
A = TAN(ANGLE)
```

Calculates the tangent of an angle of 25 degrees and assigns the value to A.

***

```
Y = TAN(30)
```

Assigns the tangent of an angle of 30 degrees to variable Y.

TIME

Purpose

The **TIME** function returns the internal time of day.

Syntax

```
TIME()
```

Comments

This function returns a string value containing the internal time of day.

The internal time is the number of seconds past midnight.

Internal times are converted to external format with the ENGLISH MT conversion.

Examples

```
X = TIME()
```

Assigns the string value of the internal time to variable X.

***

```
IF TIME() > 1000 THEN GOTO 10
```

Branches to statement 10 if more than 1000 seconds have passed since midnight.

TIMEDATE

Purpose

The **TIMEDATE** function returns the current time and date in external format.

Syntax

```
TIMEDATE()
```

Comments

External format is:

```
hh:mm:ss  dd mmm yyyy
```

where hh = hours
      mm = minutes
      ss = seconds
      dd = day
      mmm = month
      yyyy = year

Examples

```
B = TIMEDATE()
```

Assigns the string value of the current time and date to variable B.

***

```
PRINT TIMEDATE()
```

Prints the current time and date in external format. For example:

```
11:27:51  20 NOV 1986
```

TRIM

Purpose

The **TRIM** function removes unnecessary blank spaces from a specified string.

Syntax

```
TRIM(string,{,char{,type}})
```

Comments

TRIM deletes preceding, trailing and redundant blanks from the literal or variable **string**.

Char lets you specify another character to remove from the string, instead of the default blank.

Type can be one of the following specifications:

L = Remove all leading occurrences of char.
T = Remove all trailing occurrences of char.
B = Remove both leading and trailing occurrences of char.
A = Remove all occurrences of char.
R = Remove redundant occurrences of char. (R is the default type.)

If char is null, no operation is performed, and the original string is returned.

If type evaluates to null, type R is assumed.

Examples

```
NEW.STR = TRIM(OLD.STR,CHAR(254),'T')
```

Removes all trailing attributes from OLD.STR and assigns the new string to NEW.STR.

***

```
N = "     SMITH,     JOHN"
M = TRIM(N)
```

Assigns the value SMITH , JOHN to variable M.

***

```
STR1 = "1230010029911"
X = TRIM(STR1,"1",'A')
PRINT X
```

Removes all occurrences of the number 1 and prints the string 230000299.

UNASSIGNED

Purpose

The **UNASSIGNED** function determines whether or not a value is assigned to a variable.

Syntax

```
UNASSIGNED(address)
```

Comments

Address is a single variable reference only.

UNASSIGNED returns a 1 if that variable is currently assigned a value; otherwise, it returns a zero.

Example

```
X = UNASSIGNED(FEFE)
```

Returns a 1 if the variable at address FEFE is currently assigned a value; returns a zero otherwise.

Chapter 6
DATA/BASIC Commands Entered at TCL

Overview

This chapter presents TCL verbs and PROCs that are used to manipulate DATA/BASIC programs. These verbs and PROCs let you do things like compile, catalog, and execute a program. The following commands (which must be entered at TCL) are listed in alphabetical order.

BASIC

Definition

The **BASIC** verb compiles a DATA/BASIC program that was created using the EDITOR. BASIC creates a new file item containing the compiled DATA/BASIC program. It appends a dollar sign ($) to the newly-created item-id.

Syntax

```
BASIC file-name [item-list {(options)}] [*]
```

Comments

File-name is the name of the file in which the DATA/BASIC program was created.

If you specify an **item-list,** it must consist of one or more item-ids, separated by one or more blanks.

If used, any **option(s)** must be enclosed in parenthesis. If option is not followed by any other specification, the closing parenthesis is not required. Multiple options must be separated by commas. Valid options include:

B	Backwards compatibility. Use option B to compile programs written prior to the current REALITY O/S version to make them compatible for variable allocation when CHAINING programs.
E	Compile program without end-of-line (EOL) opcodes in object text. Option **E** reduces the size of debugged programs by eliminating one byte from the object text for each line in the program. Remember that any run-time error messages will indicate a line number of one, because EOLs are used to count lines.
L	Print **L**isting of the DATA/BASIC program.
M	Create symbol table and print **M**ap of DATA/BASIC program. The M option must be used if you are using the debugger. (See the next topic "BASIC Verb with Map Option" for more information.)

BASIC (Continued)

N No page. This option inhibits automatic paging on the terminal when either the L or M option is used.

P Print DATA/BASIC compiler error messages on the printer.

X Create a cross reference of variables and labels in the program.

Note: A file called CSYM must be present on the account before performing this process. If it is not already there, you must create it. **X** clears the CSYM file and then creates an item for each variable and label in the program. Each created item contains the line number(s) on which that label or variable is referred to. An asterisk (*) is appended to line numbers where the variable value may change, such as an assignment statement (with the exception of function and subroutine parameters).

If the **X** option is used with an item-list mass compile, the **X** option will only be performed while compiling the first item in the item-list.

If you specify an asterisk (*) rather than an item-list, all object code (items beginning with $) and map items (items beginning with *) are ignored.

If a DATA/BASIC program is written incorrectly, compilation error messages are displayed as the program compiles. The program does not compile successfully, and the following message is displayed:

```
LINE xxx [B100] COMPILATION ABORTED; NO OBJECT CODE PRODUCED
```

If there are no errors in the program, it compiles successfully, with the following message:

```
LINE xxx [B0] COMPILATION COMPLETED
```

BASIC (Continued)

Example

```
:EDIT PROGRAMS TESTING
NEW ITEM
TOP
.I
001 PRINT "THIS IS "
002 PRINT "A TEST"
003 END
004
TOP
.FI
'TESTING' filed in file 'PROGRAMS'.

:BASIC PROGRAMS TESTING
***
LINE 003 [B0]  COMPILATION COMPLETED
```

BASIC Verb with Map Option

Purpose

The **M(ap)** option of the BASIC verb creates a symbol table and prints a map of the program, consisting of a variable map and a statement map.

Definition

The M option creates a new item in the file, whose item-id consists of an asterisk concatenated to the program item-id. This new item is the symbol table that must be available for the debugger to reference variables by name. If a program is subsequently recompiled without the M option, the old symbol table map item is deleted.

Variable Map

The variable map lists the offset of every DATA/BASIC variable in the program. The offset is in decimal form. It starts at the beginning of the seventh frame of the IS buffer. For example:

```
   20 xxx    30 yyy
```

This entry indicates that the variable descriptor xxx starts at byte 20 of the seventh frame of the IS buffer and variable descriptor yyy starts at byte 30. Descriptors are ten bytes long. Descriptor format is illustrated in Table 6-1.

Statement Map

The statement map shows which frame number contains which statement numbers of the DATA/BASIC program. When the program is run, frame 01 is the seventh frame in the OS buffer. If the program is cataloged, frame 01 is specified in the catalog pointer item in the POINTER-FILE.

The statement map can be used to determine if frequently-executed loops cross any frame boundaries.

Sample Map

Table 6-1 shows the map that might display when you enter **:BASIC PROGRAMS PYTHAG (M)**

Table 6-1. Sample Map

```
0020 A        0030 B        0040 CC       0050 C
0060 I        0070 X

FRAME LINES
 01  001-023

LINE 23 [BO] COMPILATION COMPLETED
```

Descriptor

Table 6-2 outlines the format of a variable descriptor.

Table 6-2. Map Descriptor Formats

Descriptor Name	Bytes	Contents
Unassigned	1-10	Zeros
Direct number	1	Type code = X'01'
	2	Unused
	3-8	Binary number
	9-10	Unused
Direct string	1	Type code = X'02'
	2	Unused
	3-10	Short string terminated by an SM
Indirect string	1	Type code = X'82
	2	Unused
	3-8	SR pointing one byte before string terminated by an SM in Free Storage, Temporary Space, or Binary Object Code
	9-10	Free Storage buffer size
File BMS	1	Type code = X'04'
	2	Unused
	3-6	Base
	7-8	Modulo
	9-10	Separation
Subroutine address	1	Type code = X'40'
	2	Unused
	3-8	SR pointing to cataloged subroutine
	9-10	Unused

BLIST

Purpose

BLIST lists DATA/BASIC source code in a more usable format than a simple LIST statement.

Syntax

BLIST {DICT} file-name item-id {(options)}

Note: **File-name item-id** specifies the program you want listed. However, if you generate an item-list using one the possible ENGLISH verbs (e.g., SELECT, BSELECT, etc.), do <u>not</u> specify an item-id.

Options

The following options are included:

A Indent **A**ll comments. Indents any line beginning with an asterisk (*) to the middle of the page.

Note: BLIST does not recognize comment lines that begin with the REM statement. Use an * to specify comments if you want to indent them with BLIST.

B Blanks. Set number of blanks per indent level (default is 3).

Logical structures are automatically indented 3 spaces. Use the B option to change this default value. You will be prompted for a number from 1 to 5 before the listing begins.

C Keep **C**omments at left margin (default is to follow indenting).

D **D**ouble-space output.

E **E**xpand INCLUDEd sections into the listing (used with the 'M' option).

F Include source **F**ilename in heading.

I **I**ndent. Causes any comments beginning at the end of a normal statement line to be indented to the middle of the page.

K **K**ill. Suppress generation of line of asterisks (*) when the source line begins with an exclamation point (!).

BLIST (Continued)

L	Outline Logical structures. Prints a period (.) at each level of indenting. (See discussion of Logical Structures.)
M	Print DATA/BASIC line numbers for INCLUDEd items (without expand). (See discussion of INCLUDE Statements.)
N	NOPAGE. Listing scrolls without paging.
P	Output to system Printer (default is to terminal).
S	Suppress listing.
U	Update source code with logical indenting. Logical formatting can then be seen when you edit the program with the EDITOR. The new format does not affect compilation.
X	Print level number for each INCLUDE statement (used with 'M' and 'E' options). (See discussion of INCLUDE Statements.)
n{-m}	Begin printing with line **n** {through line **m**}. This option lets you restrict the program listing to certain lines and still retain the correct logical indenting.

Additional options include:

!	When used as the first nonblank character of a line, causes a row of asterisks (*) to be printed as the output line. (This may be suppressed using the K option.)
!!	When used as the first two characters of a line, causes a page eject after printing the line.

Line Numbers BLIST numbers all statement lines. The line numbers correspond to the line numbers supplied by the EDITOR when editing the source code. BLIST also indents all lines to allow labeled statements to be "outdented". This makes it much easier to modify the program and to locate labels quickly.

Logical Structures The L option performs logical indenting of CASE and multiline IF statements, and of LOOP and FOR-NEXT structures. This makes it easier to follow the flow of the program.

BLIST (Continued)

BLIST also flags any logical structure's ending statement that does not match the beginning statement. For instance, if a structure begins with a FOR statement, it must end with a NEXT statement. If it ends with any other statement (e.g., END or REPEAT), BLIST flags the invalid ending statement.

The L option prints a dot at each level of indenting, as in the following example:

```
IF A = 2 THEN
.  FOR I = 1 TO 10
.  .  PRINT I
.  NEXT I
END
```

INCLUDE Statements

When DATA/BASIC compiles programs that have INCLUDE statements, the line numbers output in error messages and by the debugger will differ from those in the source code. The M option lets you see both the EDITOR line numbers and the DATA/BASIC line numbers. The first column gives the attribute numbers corresponding to the source code item. The second column gives the line numbers DATA/BASIC uses when compiling and running the program.

If you specify the E option in conjunction with the M option, the INCLUDEd items are expanded in the listing. The same two columns are displayed.

If you use the X option with the E option, a number corresponding to the current level of the INCLUDE statement is displayed in the third column.

For example, in the following example, the programs T1, T2 and T3 contain the following statements:

```
T1                       T3
PRINT "LEVEL 1"          PRINT "LEVEL 3"
INCLUDE T2 FROM BP       END
END

T2
PRINT "LEVEL 2"
INCLUDE T3 FROM BP
END
```

BLIST (Continued)

The statement **:BLIST BP T1 (M,E,X)** displays the following information:

```
Edit  Bsc

001   001               PRINT "LEVEL 1"
002   002    0          *INCLUDE T2 FROM BP
      003    1          PRINT "LEVEL 2"
      004    1          *INCLUDE T3 FROM BP
      005    2          PRINT "LEVEL 3"
      006    2=======>END
      007    1=======>END
003   008               END
```

Examples

:BLIST FILE1 (N)

Displays FILE1 on the terminal without stopping at each page.

***

:BLIST FILE3 (F,U)

Displays the file-name (FILE3) in the heading and updates the source program with spaces to reflect logical indenting.

***

:BLIST TEST1 (P,100-150)

Outputs lines 100 through 150 to the printer.

***

The program TESTPROG contains the following code:

```
Q = 1
PRINT Q, * Prints the value of Q
FOR I = 1 TO 10
PRINT I*Q
NEXT I
* Executes loop 10 times
END
```

BLIST (Continued)

When you execute the following command:

```
:BLIST TESTPROG (A
```

the program now looks like this:

```
Q = 1
PRINT Q,                    * Prints the value of Q
FOR I = 1 TO 10
   PRINT I*Q
NEXT I
                            * Executes loop 10 times
END
```

BREF

Purpose **BREF** produces a sorted cross-reference listing of variables and labels in a DATA/BASIC program.

Syntax BREF {(P)}

Comments The **P** option sends the listing to the printer.

BREF gets its listing from the CSYM file. (If the CSYM file is not already present on your account, you must create it.) In order to place a cross-reference item in the CSYM file, the program must have been compiled using the X option.

Table 6-3 shows a sample of a BREF display. Each item contains the line number(s) on which that label or variable is referred to. An asterisk (*) is appended to line numbers where the variable's value may change, such as an assignment statement (with the exception of function and subroutine parameters).

Table 6-3. Sample BREF Display

```
16:30:03  18 FEB 1987                          PAGE 1
SYSTEM CROSS-REFERENCE

CSYM               TYPE  REFERENCES

E  . . . . . .           002  041

ST.NUM . . . .           004* 016* 016  040* 041  046

FLAG . . . . .           015* 018* 039

99 . . . . . .           017  *044

3  . . . . . .           021  035*

I  . . . . . .           020* 020* 021  022  023  024
                         025* 027* 027  028  030*
                         033  046* 046* 047

SP . . . . . .           038* 039* 041

BUF  . . . . .           001* 044

STRINGS  . . .           003  016* 041* 047
```

BVERIFY

Purpose

The **BVERIFY** verb verifies the object code of a cataloged program (provided that the $item is present in the original file).

Syntax

BVERIFY item-id {account-name} {(options)}

Comments

Item-id is the cataloged program's name.

If you want to verify a program in another account, specify the **account-name**.

The following **options** are available:

A	List **A**ll mismatches.
E	List only the **E**rrors found.
P	Send output to the **P**rinter.

If the verify is successful, the following message is displayed:

```
FILE      : BP
ITEM      : MATH
COMPILED  : 18 FEB 1987 AT 16:16:38
PRECISION : SEQUEL 5.2 / SPIRIT 2.2
VERIFIED.
```

Examples

:BVERIFY ITEM1 (A)

Verifies the object code of ITEM1 and lists all mismatches, if any.

***

:BVERIFY ACCT4 PAYROLL (P)

Verifies the object code of ACCT4 in the PAYROLL account and sends output to the printer.

CATALOG

Purpose

The **CATALOG** verb loads the object code of a DATA/BASIC program into system disk space, so it can be shared by several users at once. If more than one user needs the same frame of code, they can share the copy that is in main memory, thereby reducing the number of frame faults.

Syntax

```
CATALOG file-name item-id {item-id...}
```

Comments

File-name item-id specifies the compiled DATA/BASIC program to catalog. You can specify more than one item-id if you wish.

When the program has been cataloged, the following message is displayed:

```
[241] item-id CATALOGED; n FRAMES USED.
```

N is the size of the object code in frames (500 bytes each).

If it does not already exist, a TCL-II verb (with the same name as the item-id) is added to your Master Dictionary (M/D). The M/D entry for the verb has the following form:

```
001 P
002 10B4
003
004
005 account-name item-id
```

Account-name is the name of the account from which the catalog took place, and **item-id** is the name of the cataloged program.

Placing the name of the cataloged program in the M/D allows cataloged program pointers to be copied to new names without having to recatalog the original program under a different name.

If the item already exists in your M/D, but it is not in the above format, it will not be cataloged, and the following message displays:

```
[415] item-id EXISTS ON FILE
```

Some previously cataloged programs may show only the account-name in the M/D. These can be changed to the newer format by manually adding the item-id after the account-name in the M/D.

CATALOG (Continued)

Note: If a program is recompiled, it must be recataloged.

Examples

:CATALOG BP TEST1

Loads the object code of program BP TEST1 into system disk space and displays the following message:

```
[241] 'TEST1' CATALOGED; 1 FRAMES USED.
```

***

:CATALOG FILEA 1 2 3

CATALOGs the three DATA/BASIC programs with item-ids of 1, 2, and 3 in file FILEA. The following messages are displayed:

```
[241] '1' CATALOGED; 3 FRAMES USED.
[241] '2' CATALOGED; 1 FRAMES USED.
[241] '3' CATALOGED; 5 FRAMES USED.
```

DB

Purpose

The **DB** PROC helps you develop programs faster by providing a menu selection for editing, compiling and running DATA/BASIC PROGRAMS.

Syntax

DB

Comments

DB prompts you for the name of the file and program (item) you want to edit, compile, run or print.

When you enter the file and item names, the 'at' sign (@) is displayed, prompting you for input. The valid options are:

E	Edit program.
C	Compile program.
R	Run program.
P	Copy program to printer.
<RETURN>	Return to TCL.

A similar PROC, but one providing a larger menu with more operations, is listed in the PROC programming examples in the PROC Programming Reference Manual.

Example

In the following example, boldface text represents text that you type. The rest of the text is generated by the system.

```
:DB

FILE NAME = BP
ITEM NAME = TEST

@E
TOP
.
```

At this point, you can edit the TEST program in file BP. When you exit the EDITOR, DB returns you to the @ prompt.

DELETE-CATALOG

Purpose

DELETE-CATALOG "decatalogs" a DATA/BASIC program.

Syntax

DELETE-CATALOG item-id {account-name}

Comments

An item is maintained in the POINTER-FILE for each cataloged DATA/BASIC program. DELETE-CATALOG deletes the specified **item-id** from the POINTER-FILE, returns all the overflow space and deletes the verb from your M/D.

If you want to "decatalog" a program contained in another account, simply specify the **account-name**.

After the verb has been deleted from the M/D, if you try to execute it from any account, the following message displays:

```
item-id IS NOT A VERB
```

Examples

:DELETE-CATALOG TEST4 SYSPROG

If TEST4 is a valid cataloged program on the SYSPROG account, it is deleted and the following message displays:

```
[242] 'TEST4' DELETED.
```

***

:DELETE-CATALOG ITEMA

If ITEMA is not a cataloged program, it cannot be deleted. The system displays the message:

```
'ITEMA' NOT ON FILE
```

PRINT-CATALOG

Purpose

The **PRINT-CATALOG** verb prints the time and date that a specified cataloged DATA/BASIC program was compiled and the precision used in the program. (This is the same as the PRINT-HEADER verb, except that it is used for cataloged programs.)

Syntax

```
PRINT-CATALOG file-name item-id {(P)}
```

Comments

File-name item-id specifies the DATA/BASIC program.

Use the **P** option to send the output to the printer.

If you try to execute a PRINT-CATALOG on a program which has not been cataloged, the following message displays:

```
'account-name*C*item-id' NOT ON FILE.
```

Example

:PRINT-CATALOG BP PGM1

The following information is printed about the cataloged program PGM1:

```
FILE:     : BP
ITEM      : PGM1
COMPILED  : 19 FEB 1987 AT 08:20:32
PRECISION : SEQUEL 5.2 / SPIRIT 2.2
```

PRINT-HEADER

Purpose The **PRINT-HEADER** verb prints the time and date that a specified DATA/BASIC program was compiled and the precision used in the program.

Syntax PRINT-HEADER file-name item-id {(P)}

Comment **File-name item-id** specifies the DATA/BASIC program.

Use the **P** option to send the output to the printer.

Example **:PRINT-HEADER PROG A1 (P)**

Displays the following information about the DATA/BASIC program:

```
FILE:     : PROG
ITEM      : A1
COMPILED  : 19 FEB 1987 AT 11:06:42
PRECISION : SEQUEL 5.2 / SPIRIT 2.2
```

RUN

Purpose The **RUN** verb executes a compiled DATA/BASIC program.

Syntax RUN file-name item-id {(options)}

Options **File-name item-id** specifies the DATA/BASIC program to execute.

If any **options** are specified, they must be enclosed in parenthesis. The closing parenthesis is optional. Multiple options must be separated by commas. The following options are available:

D Run-time Debug. Breaks to the DATA/BASIC symbolic debugger before executing the first statement in the program and whenever a DEBUG statement is executed. (The @ command in the debugger inhibits breaks at DEBUG statements.)

F Treats warning messages as Fatal errors. Breaks to debugger to allow determination of error and possible recovery. You may continue program execution with the G command or by pressing the <LINE FEED> key.

G Garbage collection data. Used to keep track of byte and buffer use during program execution. Twelve counters are used to determine the following:

- Total number of bytes used for variable storage.
- Total number of bytes abandoned.
- Number of 50-byte buffers used, abandoned, and reused.
- Number of 150-byte buffers used, abandoned, and reused.
- Number of 250-(or multiples of 250) byte buffers used, abandoned, and reused.

A garbage collection/buffer use report is printed when the program terminates. (For more information, refer to "Appendix C. Variable Structure and Allocation".)

I Inhibits initialization of data area. (Refer to the explanation of the I option in the CHAIN statement, found in Chapter 4, "DATA/BASIC Statements".)

RUN (Continued)

N	No page. Does not wait for a carriage return after each page is output to the terminal.
P	Turns the Printer on.
S	Suppresses run-time warning messages.
T	Inhibits writing of Tape label when using the WRITET statement. (This allows for compatibility to releases prior to 3.0.)

Cataloged Programs

Cataloging programs minimizes the time needed to execute a program. This is because cataloged programs are executed by issuing the item-id of the program as a verb. The same options apply to cataloged programs.

Run-time Errors

If a run-time error occurs, an appropriate warning/error message displays and the program traps to the DATA/BASIC symbolic debugger. Fatal run-time errors cause the program to abort. (For more information about run-time error messages, refer to "Appendix B. DATA/BASIC Messages".)

Example

```
:RUN PROGRAMS TESTING
```

This command executes the program TESTING.

DATA/BASIC and PROC

Introduction

A **DATA/BASIC** program may be used in a **PROC**.

Example

The following DATA/BASIC program is called LISTIDS:

```
          OPEN 'BASIC/TEST' ELSE
             PRINT 'FILE MISSING'; STOP
          END
      10  N = 0
      20  READNEXT ID ELSE STOP
          PRINT ID 'L#################':
          N=N+1
          IF N >= 4 THEN PRINT; GOTO 10
          GOTO 20
          END
```

The following PROC, called LISTBT, contains the command to execute the program LISTIDS:

```
   PQN
   HSSELECT BASIC/TEST
   STON
   HRUN BASIC/TEST LISTIDS<
   P
```

To execute this PROC (and the DATA/BASIC program contained within it), simply type in the name of the PROC at the TCL prompt. For example:

```
   :LISTBT
```

LISTBT sort selects the item-ids contained in the file BASIC/TEST and invokes the DATA/BASIC program LISTIDS, which then lists the selected item-ids, four per line.

DATA/BASIC Symbolic Debugger

Overview

There are three ways to enter the DATA/BASIC Symbolic Debugger:

1. Press the <BREAK> key while the program is executing.
2. Specify the D option when you issue the RUN verb (or when you specify the program item-id for a cataloged program).
3. Execute a DEBUG statement from within a DATA/BASIC program (only valid if the D option has been specified with the RUN verb).

Once you enter the debugger, the system displays the line number about to be executed and prompts you with an asterisk (*). The * distinguishes the DATA/BASIC debugger from the system debugger and TCL. For example, the following display appears when you press the <BREAK> key while the TEST program is executing:

```
:RUN PROGRAM TEST

*I15
*
```

This display indicates that line 15 was about to be executed when you pressed the <BREAK> key. The debugger prompts you for input.

Symbol Table

In order to reference variables and arrays by name while in the debugger, a symbol table must be present for that program. You create the symbol table when the program is compiled, by specifying the M option.

The symbol table is formed as a new item in the file with an * concatenated to the front of the item-id. If the symbol table is not present when a variable is referenced by name, the following message is printed:

```
NO SYM TB
```

Sample Exercise

Figure 7-1 shows a sample DATA/BASIC program. Figure 7-2 shows a sample exercise using the DATA/BASIC Symbolic Debugger. Boldface text represents the input you would type on the terminal. The rest of the text is generated by the debugger. The debugger commands you see in this exercise are explained in detail on the following pages.

```
          TEST3

     001  A=123.456
     002  B="THIS IS A STRING"
     003  DIM X(3)
     004  X(1)=123
     005  X(2)="HELLO THERE"
     006  X(3)=0
     007  PRINT A,B
     008  PRINT X(1),X(2),X(3)
     009  END
```

Figure 7-1. Sample DATA/BASIC Program

Dialogue	Explanation
:BASIC BP TEST3 (M) <RETURN>	Compile program with 'M' option to create Symbol Table.
:RUN BP TEST3 (D) <RETURN>	Run program with 'D' option to break before first line is executed.
*E1	Indicates execution halted before line 1.
***/X <RETURN>** X(1)=0= **<RETURN>**	Display array X. Did not change any elements; value is zero because no lines have been executed. Variable unchanged.
UNASGN VAR	
X(2)=0= **<RETURN>**	
UNASGN VAR	
X(3)=0= **<RETURN>**	
UNASGN VAR	
***B$=5 <RETURN>** +	Break when line number is 5.
***G <RETURN>**	Go.
*B1 5	Indicates that break condition is satisfied; about to execute line 5.
***/X(1) <RETURN>** 123= **<RETURN>**	Display X(1). Leave unchanged.
***TX(2) <RETURN>** +	Trace variable X(2). Print at each break. Set single step. Break at each statement.
***E1 <RETURN>**	
***G <RETURN>**	Go.
*E6	Indicates execution break caused by E1; program is about to execute line 6.
X(2) HELLO THERE	X(2) automatically displayed by trace.
***G <RETURN>**	Go.
*E7	Indicates execution break caused by E1; program is about to execute line 7.
X(2) HELLO THERE	Trace value of X(2) displayed automatically. Set execution break to normal mode. E1 off. Find what line is about to be executed.
***E <RETURN>**	
***$ <RETURN>** 7	
***/A <RETURN>** 123.456=**<RETURN>**	Display variable A. Leave unchanged.
***P <RETURN>** OFF	Turn terminal print off.
***B$=10 <RETURN>** +	Break when line number is 10.
***D <RETURN>**	Display trace and break tables.
T1 X(2)	
T2	
T3	
T4	
T5	
T6	
B1 $=5	
B2 $=10	
B3	
B4	
***K1 <RETURN>** -	Kill first break condition ($=5).
***/A <RETURN>** 123.456=356.71 **<RETURN>**	Display variable A and change it to 356.71.

Figure 7-2. Sample Exercise Using the Debugger

Summary of Debugger Commands

Introduction This section contains a brief description of the DATA/BASIC Symbolic Debugger commands, organized by function. Following this summary is a complete description of each command in alphabetical order.

Breakpoint and Trace Tables	B	Causes an execution break when a specified condition is true.
	D	Displays the break and trace tables.
	K	Deletes (kills) breakpoint conditions from the Breakpoint table.
	T	Traces and prints the value of a specified variable at each execution break.
	U	Deletes a variable from the trace table or deletes the entire table.
Symbol Table	Z	Assigns the symbol table.
Execution Control	E	Specifies number of program lines to execute between execution breaks.
	G	Resumes normal program execution until another execution break occurs.
	N	Bypasses a specified number of breakpoints before returning to debugger control.
	@	Toggles the function of any DEBUG statements found in the program.
Special Commands	/	Displays or changes variables and arrays during program execution.
	$	Displays the number of program line that is about to be executed.
	?	Displays the name of the program currently running.
	LP	Forces all program output to the printer.
	P	Suppresses all output from the DATA/BASIC program to the terminal, so that only debugger output is displayed.
	PC	Forces printing of any data that is waiting to be output.

Summary of Debugger Commands (Continued)

Exiting the Debugger	END	Terminates the program.
	OFF	Terminates the program and logs you off the system.

$ Command

Purpose	The **$** command displays the number of the program line that is about to be executed.
Syntax	$
Example	*$ <RETURN> 7
	Indicates that line 007 of the DATA/BASIC program is about to be executed.

/ Command

Purpose

The / command lets you display the value of a variable and then modify it if you wish.

Syntax

```
/[variable-name] [*]
```

Comments

Variable-name can be the name of a simple variable, an array or an array element.

If an * is used in place of the variable-name, the values of all the variables in the program are displayed.

Any system delimiters contained in a displayed variable are changed to printable characters. All other control characters are converted to tildes (~). Because this conversion also takes place when a statement is executed by single-stepping with the debugger, the debugger display is not interrupted by program-generated cursor control.

If the variable does not exist, or if the wrong symbol table is assigned, the following message is displayed:

```
SYM NOT FND
```

It is easy to modify variables in the debugger. For example:

/CITY <RETURN> ROME=

This command displays the value of variable CITY. You can then change the value by typing the new value next to the old one. To leave the value unchanged, simply press <RETURN>.

If an entire array has been specified, each element is displayed individually and each can be changed in turn. To stop the display of array elements, press the <BREAK> key.

If the /* form of the command is used, the variables are displayed but you cannot modify them.

/ Command (Continued)

Examples

***/NAME(3)**

Displays the value of the third element in the array NAME.

***

***/NAME**

Displays the value of every element of array NAME.

***

/

Displays the values of all the variables in the program.

***

***/GRID(4,5)**

Displays the value in the fourth row, fifth column of the matrix GRID.

***

***HOURS**

Displays the value of the variable HOURS.

? Command

Purpose The ? command displays the name of the program that is currently executing.

Syntax ?

Example *? **<RETURN>** TEST3

Displays the name of the DATA/BASIC program you are currently running (in this case, TEST3).

@ Command

Purpose

The **@** command inhibits a break if a DEBUG statement is encountered.

Syntax

@

Comments

If a program contains one or more DEBUG statements and the D option was specified at run-time, an execution break occurs every time a DEBUG statement is encountered.

The @ command toggles the function of the DEBUG statement. Issuing the @ command one time inhibits breaking. Issuing it a second time turns it back on.

The words ON and OFF are printed next to the @ to indicate the current status of the @ command.

Examples

```
*@ <RETURN> ON
```

Indicates that any subsequent DEBUG statements will be ignored.

***

```
*@ <RETURN> OFF
```

Turns the @ command off, so a subsequent DEBUG statement will cause an execution break.

B command

Purpose

The **B** command causes a break in program execution when a specified condition is true.

Syntax

```
B variable-name operator expression {& condition2}
B $ operator line-number {& condition2}
```

Comments

Variable-name can be a simple variable or an array element. **Expression** can be a variable, constant or array. **Operator** can be any of the following logical operators:

<	less than
>	greater than
=	equal to
#	not equal to
&	logical AND

If you specify another condition (**condition2**) also, the two conditions must be separated by the logical AND (**&**). The break occurs only when both conditions are true.

The dollar sign (**$**) indicates a line number in the program, so it means "when the line number is equal to ...".

String constants must be enclosed in quotes, using the same rules that apply to DATA/BASIC literals.

If the variable does not exist, or if the wrong symbol table is assigned, the following message is displayed:

```
SYM NOT FND
```

A plus sign is printed next to the command if it is accepted. When the condition is met, an execution break occurs, and the debugger stops execution of the program and displays:

```
*Bt n
```

T is the breakpoint table entry number and **n** is the number of the program line that caused the break.

Examples

```
*BX>42
```

Sets a break condition to stop execution when variable X is greater than 42.

***

B command (Continued)

```
*BADDRESS=''
```

Breaks when variable ADDRESS is null.

```
*BDATE=INV.DATE&$=22
```

Breaks when variable DATE is equal to variable INV.DATE and the line number equals 22.

***

```
*BNAME="CATHY"&STATE="CA"
```

Break occurs when variable NAME is equal to CATHY and variable STATE is equal to CA.

***

```
*BPRICE(3)=24.98
```

Breaks when the third element of array PRICE is equal to 24.98.

D command

Purpose The D command displays the Break and Trace tables.

Syntax D

Example

```
*B$=4
*G
*B1 4
.
*TX(3)
.
.
*B$=11
*D
```

When the D command is issued, the contents of the Breakpoint and Trace tables are displayed as follows:

```
T1 X(3)
T2
T3
T4
T5
T6
B1 $=4
B2 $=11
B3
B4
```

E command

Purpose

The **E** command lets you specify the number of program lines to execute between execution breaks.

Syntax

En

Comments

N is the number of program lines that will execute before another execution break is taken. If n is omitted, the E command by itself turns the function off.

The E command will be overridden if a condition in the Breakpoint table is met.

Examples

```
*E5
```

Allows five lines of the program to execute before the next execution break.

***

```
*E1
```

Sets execution control so that only one line of the program is executed at a time.

***

```
*E
```

Turns the E function off.

***

```
*BX=34
*E6
```

If the condition of X=34 is met, a break occurs immediately; otherwise, executes six lines of program code before the next break.

END command

Purpose

The **END** command terminates the DATA/BASIC program and exits the debugger.

Syntax

```
END <RETURN>
END <LINE FEED>
```

Comments

If you press <RETURN> , the program terminates and control returns to TCL.

If you press the <LINE FEED> key after you type in END and the program was being executed from a PROC, the program terminates and control returns to the next statement in the PROC.

Examples

```
*B$=5 <RETURN>
*G <RETURN>
*B1 5
*END (RETURN>
```

Terminates program and returns to TCL.

***

```
*END <LINE FEED>
```

Terminates program and returns control to PROC that called it.

G command

Purpose

The **G** command resumes normal execution of a DATA/BASIC program until the next execution break is encountered.

Syntax

G{line-number}

Comments

If you specify a **line-number**, program execution continues with the DATA/BASIC statement on that line. If you specify a line-number that is greater than the number of lines in the program, the following message displays:

```
# > PROGRAM LENGTH
```

If you just press **<RETURN>** after typing G, program execution continues with the next line in the program.

Control returns to the debugger when another execution break occurs.

Examples

```
*B$=12 <RETURN>
*G <RETURN>
*B1 12
```

The G command resumes program execution until the condition in the Breakpoint table is satisfied (i.e., until line 12 is reached).

***

```
*G42 <RETURN>
```

Resumes program execution starting at line 42.

K command

Purpose The K command deletes (kills) one or all of the breakpoint conditions in the Breakpoint table.

Syntax K{n}

Comments If **n** is specified, it must be in the range 1 through 4. The specified breakpoint is deleted and the other breakpoints remain unchanged. If **n** is not specified, all breakpoint conditions are deleted.

Examples

```
*K2
```

Deletes the second breakpoint condition.

***

```
*K
```

Deletes all breakpoint conditions.

LP command

Purpose The **LP** command sends all program output to the printer.

Syntax LP

Comments Subsequent LP commands toggle the printer function. The current status of the LP command is displayed when you type LP. ON indicates that output will be sent to the printer, and OFF indicates that output will go to the terminal.

Example ***LP <RETURN>** ON

Turns the LP function on, so that any subsequent output is sent to the printer.

N command

Purpose	The **N** command lets you bypass a specified number of breakpoints.
Syntax	Nx
Comments	N lets you bypass **x**+1 breakpoints before control returns to the debugger. For example, N3 allows four breakpoints to pass before execution breaks again. However, variables being traced are still printed at each breakpoint. Press <RETURN> after typing N to reset the function so that no more breakpoints are bypassed.
Examples	***N4** Allows five (4+1) execution breaks to pass before returning control to the debugger. *** ***N** Returns the debugger to normal operation, stopping at every execution break.

OFF command

Purpose

The **OFF** command terminates the program and logs you off the system.

Syntax

```
OFF
```

Example

```
*B$=12
*G
*B3 12
*OFF
```

Logs you off the system.

P command

Purpose

The **P** command suppresses all output from the DATA/BASIC program to the terminal, so that only output from the debugger is displayed.

Syntax

P

Comments

Each time you issue the P command it toggles the status. The word ON or OFF is displayed next to the P command to let you know what the current status is. OFF indicates that program output is suppressed, while ON means that it displays.

Example

*P **<RETURN>** OFF

Suppresses program terminal output until another P command is issued.

PC command

Purpose The **PC** command forces printing of any data that is waiting to be output. Its function is similar to the DATA/BASIC PRINTER CLOSE statement.

Syntax PC

Comments Normally, printer output is held until the program finishes execution. PC lets you print out anything that is already in the print queue.

Example

```
*PC
```

Forces printing of data waiting to be output.

T command

Purpose

The **T** command creates a trace table to trace and print the values of specified variables at each execution break.

Syntax

T{variable-name}

Comments

Variable-name can be either a simple variable or an array element.

Note: Only individual array elements can be traced.

If the variable does not exist or the wrong symbol table is assigned, the following message displays:

```
SYM NOT FND
```

If the command is accepted, a plus sign (+) is printed next to the command.

The trace table may contain up to six variable names whose values are printed whenever an execution break occurs.

The trace can be turned on or off by specifying T without the variable-name. Simply type T and press **<RETURN>**. The word ON or OFF is displayed next to the T command to let you know the status of the trace function.

Examples

***TX**

Sets a trace for variable X.

***

***TPRICE(3)**

Sets a trace for the third element of array PRICE.

***

***T <RETURN>** OFF

Turns the trace off.

U command

Purpose

The **U** command deletes variables from the trace table.

Syntax

U{variable-name}

Comments

If you specify a **variable-name**, the U command deletes that variable name from the trace table. If you omit the variable-name, U deletes the entire trace table.

Examples

***USTATE**

Deletes the variable STATE from the trace table.

***

***UWAGES(3)**

Removes the trace for the third element in array WAGES.

***

***U**

Deletes all variables from the trace table.

Z command

Purpose

The **Z** command assigns the symbol table to the DATA/BASIC program being debugged.

Syntax

Z {DICT} file-name item-id

Comments

File-name item-id indicates the DATA/BASIC program you wish to debug.

If a symbol table is present in the same file from which the program was compiled, it is automatically assigned to the program. However, if the symbol table resides in a different file, the Z command must be used to assign it.

Examples

***Z BP FORMAT**

Assigns the symbol table FORMAT in the BP file to the current program.

***

***Z DICT BP MAPIT**

Assigns the symbol table MAPIT in the dictionary of the BP file to the current program.

Chapter 8 Programming Hints and Examples

Overview

This chapter contains a number of general coding techniques you should keep in mind when writing a DATA/BASIC program.

Using System Delimiters

The REALITY system uses standard attribute, value and subvalue delimiters. These should be defined once at the beginning of the program, then referenced by their symbol names. For example:

```
EQUATE AM TO CHAR(254)        Attribute Mark

EQUATE VM TO CHAR(253)        Value Mark

EQUATE SVM TO CHAR(252)       Subvalue Mark
```

The values are equated to symbols rather than assigned to variables (as AM=Char(254)) because you don't need to change them during the program and by equating them to a symbol they don't require accessing a variable location each time they are referenced.

Cursor Positioning

The REALITY system also uses standard Cursor Positioning Characters which also should be defined only once and then referenced by symbol name. For example:

```
EQU UP TO CHAR(26)
EQU DOWN TO CHAR(10)
EQU LEFT TO CHAR(21)
EQU RIGHT TO CHAR(6)
EQU BELL TO CHAR(7)
```

Of course, you can use extended cursor addressing (@(-10), etc.) for all of the above. However, the extended cursor addressing symbols can only be assigned to variables; they can not be equated to a symbol.

Opening Files

The OPEN statement is very time-consuming and should be executed as few times as possible. All files should be opened to file variables at the beginning of the program; access to the files can then be performed by referencing the file variables.

Repeating Operations

Operations should be predefined rather than repetitively performed. This operation, for example:

```
X=SPACE(9-LEN(OCONV(COST,'MD4'))):OCONV(COST,'MD4')
```

Repeating Operations (Continued)

should have been written:

```
E=OCONV(COST,'MD4')
S=SPACE(9-LEN(E))
X=S:E
```

The same is true for the following operation:

```
FOR I=1 TO X*Y+Z(20)
 .
 .
 .
NEXT I
```

should have been written:

```
TEMP=X*Y+Z(20)
FOR I=1 TO TEMP
 .
 .
 .
NEXT I
```

Unknown Number of Values

The following LOOP statement could be used to access an unknown number of values from an attribute (including null values):

```
EQU VM TO CHAR(253)
   READV ATTR FROM ID, ATTNO ELSE STOP
   VNO=0
   LOOP
      VNO=VNO+1
      VALUE=FIELD(ATTR,VM,VNO)
   WHILE COL2() #0 DO
      PRINT VALUE
   REPEAT
```

PYTHAG PROGRAM

The following sample program finds pythagorean triples.

```
    PRINT
    PRINT 'SOME PYTHAGOREAN TRIPLES ARE:'
    PRINT
    FOR A=1 TO 40
        FOR B=1 TO A-1
           CC=A*A+B*B
           GOSUB 50
           IF C = INT(C) THEN PRINT B,A,C
        NEXT B
     NEXT A
     STOP
*   SQUARE ROOT SUBROUTINE
50   C=CC/2
     FOR I=1 TO 20
        X=(C+CC/C)/2
        IF C = X THEN RETURN
        C=X
     NEXT I
     RETURN
END
```

GUESS PROGRAM

The following sample DATA/BASIC program is a game which asks you to guess a number between 0 and 100.

```
     HEADING "'N'"
     HISSCORE=0; YOURSCORE=0
     LOOP
        PAGE;* CLEAR SCREEN
        PRINT 'GUESS NUMBERS BETWEEN 0 AND 100'
        PRINT 'MACHINE:':HISSCORE:...
        ' ':'YOUR:':YOURSCORE
        PRINT
        NUM=RND(101)
        FLAG = 1
        FOR I=1 TO 6 WHILE FLAG
           PRINT 'GUESS ':I:' ':
           INPUT GUESS
           BEGIN CASE
              CASE GUESS<NUM
                 PRINT 'HIGHER'
              CASE GUESS>NUM
                 PRINT 'LOWER'
              CASE 1
                 FLAG = 0
           END CASE
        NEXT I
        PRINT
        IF FLAG THEN
           PRINT 'YOU LOST, YOU DUMMY; YOUR:...
           NUMBER WAS ':NUM
           HISSCORE=HISSCORE+1
        END ELSE
           PRINT "YOU WON! GREAT! FABULOUS!!"
           YOURSCORE=YOURSCORE+1
        END
        PRINT
        PRINT 'AGAIN?':
        INPUT X
     WHILE [1,1] # 'N' DO REPEAT
 END
```

INV-INQ PROGRAM

The following program queries an inventory file. It reads the dictionary of the INV file to get the attribute numbers of the part description (DESC) and the quantity-on-hand (QOH). It then prompts the user for a part number which is the item-id of an item in INV. It uses the attribute numbers to read and display the part description and quantity-on-hand. The program loops until a null part number is entered.

```
*---- Get attribute definitions from DICT INV
      OPEN 'DICT','INV' ELSE
        PRINT 'CANNOT OPEN "DICT INV"'; STOP
      END

      READV DESC.AMT FROM 'DESC',2 ELSE
        PRINT 'CANT READ "DESC" ATTR'; STOP
      END

      READV QOH.AMT  FROM 'QOH',2  ELSE
        PRINT 'CANT READ "QOH"  ATTR'; STOP
      END

*---- Open data portion of INV
      OPEN 'INV' ELSE
        PRINT 'CANNOT OPEN "INV"'; STOP
      END
*---- Prompt for part number
      LOOP
         PRINT
         PRINT 'PART-NUMBER ':
         INPUT PN
      WHILE PN # "" DO
         READ ITEM FROM PN THEN
*---- PRINT DESCRIPTION AND QUANTITY-ON-HAND
            PRINT 'DESCRIPTION - ': ITEM<DESC.AMT>
            PRINT 'QTY-ON-HAND - ': ITEM<QOH.AMT>
         END ELSE
            PRINT 'CANNOT FIND THAT PART ':PN
         END
      REPEAT
  END
```

AREA PROGRAM

The following DATA/BASIC program finds the area of various geometric figures.

```
*                            Display figure menu
*
    EQUATE ERASE TO CHAR(12)
    T=25; C=45
    PRINT ERASE
    PRINT @(-4):@(T-3,4):'THIS PROGRAM FINDS THE'
    PRINT @(T-3):'AREAS OF GEOMETRIC FIGURES.':
    PRINT @(T+3,7):'TYPE':@(C-2):'CODE'
    PRINT @(T):'-------------      ----'
    PRINT @(T):'RECTANGLE':@(C):'1'
    PRINT @(T):'CIRCLE':@(C):'2'
    PRINT @(T):'TRIANGLE':@(C):'3'
    PRINT @(T):'PARALLELOGRAM':@(C):'4'
    PRINT @(T):'RHOMBUS':@(C):'5'
    PRINT @(T):'TRAPEZOID':@(C):'6'
*                            Prompt for selection
    P=@(1,17):'ENTER '
    Q=@(1,18):'ENTER '
    R=@(1,19):'ENTER '
    AREA=@(1,20):'AREA = '
    PRINT @(1,16):'ENTER A FIGURE TYPE CODE':
10  PRINT @(26,16):'    ':@(25,16):; INPUT CODE
    BEGIN CASE
*
*                            Rectangle
    CASE CODE = 1
       PRINT P:'LENGTH':;  INPUT LENGTH
       PRINT Q:'WIDTH ':;   INPUT WIDTH
       PRINT AREA:LENGTH*WIDTH:
*
*                            Circle
    CASE CODE = 2
       PRINT P:'RADIUS':; INPUT RADIUS
       PRINT AREA:RADIUS*RADIUS*3.1416
*
*                            Triangle
    CASE CODE = 3
       PRINT P:'BASE':; INPUT BASE
       PRINT Q:'ALTITUDE':; INPUT ALTITUDE
       PRINT AREA:BASE*ALTITUDE/2
```

AREA PROGRAM (Continued)

```
*                               Parallelogram
    CASE CODE = 4
       PRINT P:'BASE':; INPUT BASE
       PRINT Q:'HEIGHT':; INPUT HEIGHT
       PRINT AREA:BASE*HEIGHT

*                               Rhombus
    CASE CODE = 5
       PRINT P:'LENGTH OF FIRST  DIAGONAL':...
       ; INPUT DIAG1
        PRINT Q:'LENGTH OF SECOND DIAGONAL':...
        ; INPUT DIAG2
       PRINT AREA:DIAG1*DIAG2/2

*                               Trapezoid
    CASE CODE = 6
       PRINT P:'LENGTH OF FIRST BASE':...
       ; INPUT BASE1
       PRINT Q:'LENGTH OF SECOND BASE':...
       ; INPUT BASE2
       PRINT R:'HEIGHT':; INPUT HEIGHT
       PRINT AREA:HEIGHT*(BASE1+BASE2)/2
       CASE CODE <1 OR CODE >6
       PRINT @(1,18):'ILLEGAL FIGURE CODE'
       END CASE
20     PRINT @(1,22): 'ENTER CR TO CONTINUE "...
       "OR X TO QUIT':; INPUT X
       IF X MATCHES '' THEN
       FOR I=20 TO 17 STEP -1
          PRINT @(0,I):EOL:
       NEXT I
       GOTO 10
    END
END
```

PROFITS PROGRAM

This program prints a profits report. A select must be performed before running this program.

```
      EQU AM TO CHAR(254), VM TO CHAR(253)
      BUDGET = 0; REV.BUDGET =
!
*  Input from file
*  Total attributes 5, 7, and 23
!
      OPEN 'PROJ' ELSE
        PRINT "CANT OPEN PROJ"; STOP
      END

      READ TOT.PRC FROM 'TOT.PRC' ELSE
        PRINT "CANT READ TOT.PRC"; STOP
      END

      READ REV.TOT.PRC FROM "REV.TOT.PRC" ELSE
         PRINT "CANT READ REV.TOT.PRC"
         STOP
      END

      READ NUM.UNITS FROM 'NUM.UNITS' ELSE
        PRINT "CANT READ NUM.UNITS"; STOP
      END

      LOOP WHILE READNEXT ID DO
         IF LEN(ID) # Y THEN PRINT
           "I.D. NOT 7 CHARACTERS"; STOP
         END
         READ ITEM FROM ID ELSE
           PRINT "CANT READ ":ID; STOP
         END
         BUDGET = BUDGET + ITEM<5>
         IF ITEM<7> > 0 THEN
            REV.BUDGET = REV.BUDGET + ITEM<7>
         END
         REV.BUDGET = REV.BUDGET + : ...
         SUMMATION(ITEM<23>)
      REPEAT
!
*  Convert to dollars and compute budget totals
!
      BUDGET = BUDGET/100
      REV.BUDGET = REV.BUDGET/100
      TOT.PRC = TOT.PRC/100
      REV.TOT.PRC=REV.TOT.PRC/100
      OVER.UNDER = BUDGET - REV.BUDGET
      TOT.PRC.O.U = TOT.PRC-REV.TOT.PRC
```

PROFITS PROGRAM (Continued)

```
!
*  Compute profits
!
     PROJ.PROF = TOT.PRC-BUDGET
     REV.PROJ.PROF = REV.TOT.PRC-REV.BUDGET
     PROJ.PROF.O.U.= PROJ.PROF-REV.PROJ.PROF


!
*  Compute per/unit profits and percentages
!
     PU1 = PROJ.PROF/NUM.UNITS
     PU2 = REV.PROJ.PROF/NUM.UNITS
     PU3 = PROJ.PROF.O.U./NUM.UNITS
     P1  = PROJ.PROF/(TOT.PRC/100)
     P2  = REV.PROJ.PROF/(REV.TOT.PRC/100)
!
*  Print
!
     FMT = 'R2,#14'
     PRINT
     PRINT "                           BUDGET:...
     REV-BUDGET":
     PRINT "   OVER/UNDER"
     PRINT "   TOTAL SALES PRICE  ":TOT.PRC:...
     FMT:REV.TOT.PRC FMT:TOT.PRC.O.UFMT
     PRINT "   TOTAL COST         ":BUDGET:...
     FMT:REV.BUDGET FMT:O.U FMT
     PRINT "   PROJECTED PROFIT   ":PROJ.PROF:...
     FMT:REV.PROJ.PROF FMT:PROJ.PROF.O.U FMT
     PRINT "   PER UNIT           ":...
     PU1 FMT:PU2 FMT:PU2 FMT
     PRINT "   AS % OF SALES      ":P1 FMT:...
     P2 FMT
END
```

DECIMAL	HEXADECIMAL	ASCII	DECIMAL	HEXADECIMAL	ASCII
000	00	NUL	046	2E	.
001	01	SOH	047	2F	/
002	02	STX	048	30	0
003	03	ETX	049	31	1
004	04	EOT	050	32	2
005	05	ENQ	051	33	3
006	06	ACK	052	34	4
007	07	BEL	053	35	5
008	08	BS	054	36	6
009	09	HT	055	37	7
010	0A	LF	056	38	8
011	0B	VT	057	39	9
012	0C	FF	058	3A	:
013	0D	CR	059	3B	;
014	0E	SO	060	3C	<
015	0F	SI	061	3D	=
016	10	DLE	062	3E	>
017	11	DC1	063	3F	?
018	12	DC2	064	40	@
019	13	DC3	065	41	A
020	14	DC4	066	42	B
021	15	NAK	067	43	C
022	16	SYN	068	44	D
023	17	ETB	069	45	E
024	18	CAN	070	46	F
025	19	EM	071	47	G
026	1A	SUB	072	48	H
027	1B	ESC	073	49	I
028	1C	FS	074	4A	J
029	1D	GS	075	4B	K
030	1E	RS	076	4C	L
031	1F	US	077	4D	M
032	20	SPACE	078	4E	N
033	21	!	079	4F	O
034	22	"	080	50	P
035	23	#	081	51	Q
036	24	$	082	52	R
037	25	%	083	53	S
038	26	&	084	54	T
039	27	'	085	55	U
040	28	(	086	56	V
041	29	)	087	57	W
042	2A	*	088	58	X
043	2B	+	089	59	Y
044	2C	,	090	5A	Z
045	2D	-	091	5B	[

DECIMAL	HEXADECIMAL	ASCII	DECIMAL	HEXADECIMAL	ASCII
092	5C	\	138	8A	
093	5D	]	139	8B	
094	5E	^	140	8C	
095	5F	_	141	8D	
096	60	`	142	8E	
097	61	a	143	8F	
098	62	b	144	90	
099	63	c	145	91	
100	64	d	146	92	
101	65	e	147	93	
102	66	f	148	94	
103	67	g	149	95	
104	68	h	150	96	
105	69	i	151	97	
106	6A	j	152	98	
107	6B	k	153	99	
108	6C	l	154	9A	
109	6D	m	155	9B	
110	6E	n	156	9C	
111	6F	o	157	9D	
112	70	p	158	9E	
113	71	q	159	9F	
114	72	r	160	A0	
115	73	s	161	A1	
116	74	t	162	A2	
117	75	u	163	A3	
118	76	v	164	A4	
119	77	w	165	A5	
120	78	x	166	A6	
121	79	y	167	A7	
122	7A	z	168	A8	
123	7B	{	169	A9	
124	7C	¦	170	AA	
125	7D	}	171	AB	
126	7E	~	172	AC	
127	7F	DEL	173	AD	
128	80		174	AE	
129	81		175	AF	
130	82		176	B0	
131	83		177	B1	
132	84		178	B2	
133	85		179	B3	
134	86		180	B4	
135	87		181	B5	
136	88		182	B6	
137	89		183	B7	

DECIMAL	HEXADECIMAL	ASCII	DECIMAL	HEXADECIMAL	ASCII
184	B8		220	DC	
185	B9		221	DD	
186	BA		222	DE	
187	BB		223	DF	
188	BC		224	E0	
189	BD		225	E1	
190	BE		226	E2	
191	BF		227	E3	
192	C0		228	E4	
193	C1		229	E5	
194	C2		230	E6	
195	C3		231	E7	
196	C4		232	E8	
197	C5		233	E9	
198	C6		234	EA	
199	C7		235	EB	
200	C8		236	EC	
201	C9		237	ED	
202	CA		238	EE	
203	CB		239	EF	
204	CC		240	F0	
205	CD		241	F1	
206	CE		242	F2	
207	CF		243	F3	
208	D0		244	F4	
209	D1		245	F5	
210	D2		246	F6	
211	D3		247	F7	
212	D4		248	F8	
213	D5		249	F9	
214	D6		250	FA	
215	D7		251	FB	SB
216	D8		252	FC	SVM
217	D9		253	FD	VM
218	DA		254	FE	AM
219	DB		255	FF	SM

Messages

This appendix presents a list of the messages which may occur when you compile or run your DATA/BASIC program. For a complete explanation of these messages, refer to the manual titled System Messages.

[83] FRAME IS PART OF BASIC RUNTIME - NOT UNLOCKED

[86] FILE REFERENCE ATTEMPTED ON FILE NOT PREVIOUSLY OPENED

[89] FATAL ATTEMPT TO UPDATE BY PROCESS THAT HAS GROUP READ LOCKED!

CATALOGED PROGRAM HAS xxx MISMATCHES (225)

[241] 'xxx' CATALOGED; xxx FRAMES USED

[310] ITEM IS LOCKED BY LINE xxx

[604] 'xxx' IS AN UNDEFINED LABEL REFERENCE

[960] 'xxx' READ LOCK CLEARED

[961] 'xxx' UPDATE LOCK CLEARED

[962] 'xxx' NOT UPDATE LOCKED BY THIS LINE

```
LI#  FID.... FID.... FID.... FID.... FID....
FID.... FID.... FID.... FID....      (963)
```

[964] NOT READ LOCKED BY THIS LINE

[965] CURRENT READ/UPDATE LOCK TABLE SIZE IS xxx DEFAULT READ/UPDATE LOCK TABLE SIZE IS xxx

[966] READ/UPDATE LOCK TABLE MUST BE EMPTY TO CHANGE THE SIZE

[967] SIZE MUST BE IN THE RANGE 1 TO xxx

[970] CURRENT ITEM LOCK TABLE SIZE IS xxx
DEFAULT ITEM LOCK TABLE SIZE IS xxx

[971] ITEM LOCK TABLE MUST BE EMPTY TO CHANGE THE SIZE

```
LI#  LEVEL LOCK.... LOCK.... LOCK.... LOCK....
LOCK.... LOCK.... LOCK....    (972)
```

[973] xxx LOCKS LISTED

Messages (Continued)

LI# FILEBASE ITEM ID.........................
.... LOCK#... BY#   (974)

[975]	ITEM LOCKED BY LINE xxx AT LEVEL xxx
[976]	'xxx' UNLOCKED
[B0]	COMPILATION COMPLETED
[B9]	WRITE, DELETE, OR CLEARFILE OPERATION ATTEMPTED ON READ ONLY FILE
[B10]	VARIABLE HAS NOT BEEN ASSIGNED A VALUE; ZERO USED!
[B11]	TAPE RECORD TRUNCATED TO x BYTES!
[B12]	FILE HAS NOT BEEN OPENED
[B13]	NULL CONVERSION CODE IS ILLEGAL; NO CONVERSION DONE!
[B14]	BAD STACK DESCRIPTOR
[B15]	ILLEGAL OPCODE: xx
[B16]	NONNUMERIC DATA WHEN NUMERIC REQUIRED; ZERO USED!
[B17]	ARRAY SUBSCRIPT OUT-OF-RANGE, ABORT!
[B18]	ATTRIBUTE NUMBER LESS THAN 1 IS ILLEGAL
[B19]	ILLEGAL PATTERN
[B20]	COL1 OR COL2 USED PRIOR TO EXECUTING A FIELD STMT; ZERO USED!
[B21]	MATREAD: NUMBER OF ATTRIBUTES EXCEEDS VECTOR SIZE
[B22]	BRANCH INDEX OF 'x' IS ILLEGAL; BRANCH TAKEN TO FIRST STATEMENT-LABEL!
[B23]	BRANCH INDEX OF 'x' EXCEEDS NUMBER OF STATEMENT-LABELS; BRANCH TAKEN TO LAST STATEMENT-LABEL!
[B24]	DIVIDE BY ZERO; RESULT ZERO!

Messages (Continued)

[B25]	PROGRAM 'x' HAS NOT BEEN CATALOGED
[B26]	'UNLOCK x' ATTEMPT BEFORE LOCK!
[B27]	RETURN EXECUTED WITH NO GOSUB
[B28]	NOT ENOUGH WORK SPACE
[B29]	CALLING PROGRAM MUST BE CATALOGED
[B30]	ARRAY SIZE MISMATCH
[B31]	STACK OVERFLOW
[B32]	PAGE HEADING EXCEEDS MAXIMUM OF 1400 CHARACTERS
[B33]	PRECISION DECLARED IN SUBPROGRAM 'x'
[B34]	INSUFFICIENT NUMBER OF PARAMETERS PASSED TO EXTERNAL SUBROUTINE; ABORT!
[B35]	'M/DICT' INVALID OBJECT OF 'CLEARFILE'; IGNORED!
[B36]	SYSTEM DICT ILLEGAL OBJECT OF "CLEARFILE"; ABORT!
[B37]	EXCESSIVE NUMBER OF PARAMETERS PASSED TO EXTERNAL SUBROUTINE; ABORT!
[B38]	MATWRITE INCREASED THE NUMBER OF ATTRIBUTES
[B39]	DIVISION OVERFLOW; RESULT IN DOUBT!
[B40]	PAGE FOOTING EXCEEDS MAXIMUM OF 400 CHARACTERS
[B41]	STRING EXCEEDED ALLOWABLE LENGTH.
[B50]	FUNCTION WITH ARGUMENT VALUE <= 0 UNDEFINED; ZERO RETURNED!
[B51]	NEGATIVE VALUE RAISED TO NON-INTEGER VALUE. ZERO RETURNED!
[B52]	STRING LENGTH GREATER THAN 32760. TRUNCATED!
[B54]	OVERFLOW ; RESULT SUSPECT!

Messages (Continued)

[B55] MAT READ/WRITE OPERATION MUST BE TO/FROM A VECTOR

[B60] PROCWRITE ATTEMPTED OUTSIDE OF PROC MODE

[B98] GARBAGE COLLECTION/BUFFER UTILIZATION REPORT

```
NUMBER OF TIMES GARBAGE COLLECTED: x
   TOTAL NUMBER OF BYTES BUFFER USED: x
   TOTAL NUMBER OF BYTES ABANDONED  : x

  ** BUFFER USAGE =*

BUFFERS/SIZE      50 BYTES      150 BYTES      250 BYTES
                  --------      ---------      ---------
USED              x             x              x
REUSED            x             x              x
ABANDONED         x             x              x

                 *--OR MULTIPLES THEREOF
```

[B99] MAXIMUM NUMBER OF NEW CONTEXT LEVELS EXCEEDED

[B100] COMPILATION ABORTED; NO OBJECT CODE PRODUCED.

[B101] MISSING "END", "NEXT", "WHILE", "UNTIL", "REPEAT" OR "ELSE"; COMPILATION ABORTED, NO OBJECT CODE PRODUCED.

[B102] BAD STATEMENT

[B103] LABEL 'x' IS MISSING

[B104] LABEL 'x' IS DOUBLY DEFINED

[B105] 'x' HAS NOT BEEN DIMENSIONED

[B106] 'x' HAS BEEN DIMENSIONED AND USED WITHOUT SUBSCRIPTS

[B107] "ELSE" CLAUSE MISSING

[B108] "NEXT" STATEMENT MISSING

[B109] VARIABLE MISSING IN "NEXT" STATEMENT

Messages (Continued)

[B110]	INVALID 'END' STATEMENT
[B111]	"UNTIL" OR "WHILE" MISSING IN "LOOP" STATEMENT
[B112]	"REPEAT" MISSING IN "LOOP" STATEMENT
[B113]	TERMINATOR MISSING
[B114]	MAXIMUM NUMBER OF VARIABLES EXCEEDED
[B115]	LABEL 'x' IS USED BEFORE THE EQUATE STMT
[B116]	LABEL 'x' IS USED BEFORE THE COMMON STMT.
[B117]	LABEL 'x' IS MISSING A SUBSCRIPT LIST
[B118]	LABEL 'x' IS THE OBJECT OF AN EQUATE STMT AND IS MISSING.
[B119]	WARNING - PRECISION VALUE OUT OF RANGE - IGNORED!
[B120]	WARNING - MULTIPLE PRECISION STATEMENTS - IGNORED!
[B121]	LABEL 'x' IS A CONSTANT AND CAN NOT BE WRITTEN INTO
[B122]	LABEL 'x' IS IMPROPER TYPE AS OBJECT OF EQUATE
[B123]	UNMATCHED "NEXT", "REPEAT" OR "END CASE"; COMPILATION ABORTED, NO OBJECT CODE PRODUCED!
[B124]	INVALID USE OF RESERVED WORD
[B125]	RESERVED WORD USED AS A LABEL. EXAMPLE: 'ELSE=5'
[B126]	ITEM-LIST 'x' HAS NOT BEEN CATALOGED WITH THE SHARE VERB
[B128]	SYMBOL 'x' EQUATED TO AN ARRAY ELEMENT WHICH IS OUT OF RANGE
[B199]	PRECISION GREATER THAN 6

Messages (Continued)

```
[B641]  HOLD FILE #x ADDED;

ATTEMPT TO READ FROM A NON-SELECT VARIABLE  (B900)
[B1000] *** 'LF' OR 'G' NOT ALLOWED AFTER FATAL
        MSG           ***

[B999]  FILE:  file-name
        ITEM:  item-id
        COMPILED:  date AT time
        PRECISION:  n
```

System Error Messages

If a DATA/BASIC program aborts with a System Error Message and enters the debugger (which prompts with an "!"), you may determine the EDITOR line number of the statement which was being executed when the error occurred by typing in:

```
!G155.1 <RETURN>
```

The system responds with:

```
LINE xx [B0] COMPILATION COMPLETED.
FILE:  file-name
ITEM:  item-id
COMPILED:  date AT time
PRECISION:  n
```

Variable Structure

Introduction

The data area used by a DATA/BASIC program consists of a descriptor table, free storage area, and a buffer size table.

Descriptor Structure

The descriptor table contains 'n' entries of 10 bytes each, where 'n' is the number of variables (including array elements) in the program. The current limit on the number of descriptors is 3224. A descriptor contains two bytes of type code which identify the type of the descriptor and one of the following:

Six byte binary number	For numeric variables.
Seven byte string terminated by an SM	For string values of seven characters or less.
Six byte pointer to the free space area	For string values with more than seven characters.
Base (4 bytes), modulo (2 bytes) separation (2 bytes)	For file-variables.
Six byte pointer to a cataloged subroutine	For cataloged program names.

For more information on descriptor structure, refer to the topic titled "BASIC Verb with Map Option" in Chapter 6, "DATA/BASIC Commands Entered at TCL".

Free Storage

The free storage area is made up of buffers of various sizes. These buffers are assigned to a variable if the string to be stored in the variable cannot fit in its descriptor (that is, the string has more than seven characters). A pointer to this area is stored in the descriptor.

Buffer Allocation

Strings longer than 7 bytes are placed in storage buffers located in the free storage space.

Buffers are 50 bytes, 150 bytes, or multiples of 250 bytes in length. When a string requires a new buffer, the store processor looks in a table of abandoned buffers for a buffer of the appropriate size. If one cannot be found, a buffer size is calculated, and a buffer of the same size is then allocated to the variable in question.

Variable Structure (Continued)

Allocating free storage this way makes the buffer larger than the string it will contain, so that larger strings can be stored in the same buffer. This is important because of the allocation procedure.

Initially, free storage is one contiguous block of space. Buffers are allocated from the beginning of the free storage area. When a string is assigned to a variable which exceeds the variable's current buffer size, the buffer is abandoned and a new buffer is allocated from the remaining contiguous portion of free storage.

If there is not enough contiguous space for the new buffer, a procedure called "garbage collection" takes place. Garbage collection collects the abandoned buffer space and forms a single block of contiguous space. If there still is not enough contiguous space, the program terminates with the message:

```
[B28] NOT ENOUGH WORK SPACE
```

Variable Allocation

Introduction

Variables are allocated descriptors in the following order:

1. Common variables
2. Simple variables
3. Dimensioned variables

Note: If the 'B' option is used with the BASIC verb, dimensioned variables are allocated descriptors before simple variables.

Passing Values: Subroutines

The arrangement of descriptors for a main program and a subroutine may be illustrated with the following figure:

Values passed through argument list			
DESCRIPTORS	COMMON	Variables	Variables
	Used by main program and subroutine	Used locally by main program only	Used locally by subroutine program program only

Figure C-1. Passing Values between Programs

Variables declared as COMMON share the same locations. There is a one-to-one correspondence between the variables in both COMMON statements.

When values are passed through the argument list by CALL and SUBROUTINE statements, the values are copied back and forth between the two local areas as indicated in Figure C-1 above.

Variable Allocation (Continued)

If subroutine calls are nested, the arrangement of descriptors is as follows:

Values passed through argument list			
COMMON	Variables	Variables	Variables
Used by main program and subroutine	Used locally by main program only	Used locally by subroutine1 program only	Used locally by second subroutine program only

Figure C-2. Passing Values Through Nested Subroutine Programs

Values passed through the argument list are copied as indicated in Figure C-2 above.

Note: You may not execute a CHAIN or ENTER statement from a subroutine, but you can CHAIN to or ENTER a program that calls a subroutine.

Passing Values: CHAINed and ENTERed Programs

Value passing is different when programs are CHAINed or ENTERed. No copying takes place, therefore values that are to be passed must be declared as COMMON variables.

Executing the RUN verb with the 'I' option guarantees that the common area will not be reinitialized when CHAINing.

Variable Allocation (Continued)

Figure C-3 illustrates the descriptor arrangement for CHAINed or ENTERed programs:

Program 1	COMMON	variables
	Used by both Programs.	Used locally by first program. These will be lost when CHAIN or ENTER occurs.
Program 2	COMMON	variables
	(Same locations as for first program.)	Used locally by second program. Overwrites variables in first program.

Figure C-3. Value Passing with CHAINed or ENTERed Programs

CAUTION

If the CHAINed or ENTERed programs have COMMON areas of different sizes, some COMMON variables may get overwritten.

For example, if the second program (i.e., the one that you CHAIN to or ENTER) has a smaller COMMON area than the first program, some COMMON variables may be lost.

Appendix D. User Exit Conversions

Introduction

The following user exits have been supplied to perform special processing when used in an ICONV or OCONV intrinsic function.

Conversion	Meaning and Usage
U307A	Causes terminal to "sleep" until the specified time (in 24-hour format) is reached. Example: DUMMY = ICONV("9:00","U307A") causes the process to "sleep" until 9:00 am.
U407A	Causes terminal to "sleep" until the specified number of seconds is reached. Example: DUMMY = ICONV(30,"U407A") causes the process to "sleep" for 30 seconds.
U50BB	Returns line number and account name. (Similar to TCL verb WHO). Example: WHO = OCONV(0,"U50BB") might assign the string "1 SYSPROG".
U10DD	Returns the system serial number as specified by the hardware. Example: SN = OCONV(0,"U10DD") might assign the value 1279.
U00E0	Returns a zero if executing a noncataloged program and a one if program is cataloged. Example: CATALOGED = OCONV(0,"U00E0")
U10E0	Returns options specified at run-time in input line in alphabetic order, commas removed. Example: OPTIONS=ICONV(0,"U10E0") assigns the string "DGT" if the options (G,T,D) are used with the RUN verb or with the cataloged program name.
U20E0	Returns TCL input statement (not valid if program run from a PROC). The verb, redundant blanks, and options are removed. Remaining blanks are replaced with attribute marks. Example: STMT = OCONV(0,"U20E0") returns the string "BP^TEST^PGM1" if the TCL input statement is "RUN BP TEST PGM1 (T)".

Conversion	Meaning and Usage
U30E0	Returns a one if a command generating a select list has previously been processed (either from TCL or within a PROC). Returns zero otherwise (including a DATA/BASIC SELECT). Example: SEL = OCONV(0,"U30E0").
U40E0	Toggles printing of warning messages during run-time (i.e., toggles the 'S' option of the RUN verb). Example: DUMMY = ICONV(0,"U40E0").
U60E0	Returns current setting of terminal page width. Example: TSIZE = OCONV(0,"U60E0").
U70E0	Sets terminal echo ON (similar to TCL verb NOHUSH). Example: DUMMY = OCONV(0,"U70E0").
U80E0	Sets terminal echo OFF (similar to TCL verb HUSH). Example: DUMMY = OCONV(0,"U80E0").
U90E0t	Returns the specified number concatenated to the character indicated by "t". Example: AMOUNT = OCONV(100,"U90E0$") returns the string "$100".

Appendix E. Mask Character Conversions

Introduction

The following mask character conversions are available for DATA/BASIC.

Conversion	**Meaning and Usage**
MCA	Returns only the alphabetic values from the input string.
MC/A	Returns only the nonalphabetic characters from the input string.
MCN	Returns only the numeric characters from the input string.
MC/N	Returns only the nonnumeric characters from the input string.
MCB	Returns just the alphabetic and numeric characters from the input string.
MC/B	Removes the alphabetic and numeric characters from the input string.
MCC;x;y	Changes all occurrences of string 'x' to string 'y' in the input string.
MCL	Converts all upper case characters to lower case.
MCU	Converts all lower case characters to upper case.
MCT	(Text). Converts all upper case characters to lower case starting with the second character in each word. This also forces the first character to upper case if necessary.
MCP	Converts all nonprintable characters (X'00'-X'1F', X'80'-X'FA') to tildes (~).

Conversion	Meaning and Usage
MCPN	Same as MCP, but whenever a nonprintable character is converted to a tilde, the tilde is followed by the two character hex representation of the overwritten character. For example, the string: ABCxDEFyGH where 'x' is a <CTRL>A and 'y' is a <CTRL>E is returned as: ABC~01DEF~05GH
MCDX	Converts the input hex value to its equivalent hex value.
MCXD	Converts the input hex value to its equivalent decimal value.

Note: MCDX and MCXD each perform the opposite function if called from DATA/BASIC with an ICONV rather than an OCONV.

Introduction

The following date conversions are available in DATA/BASIC. In each case, the conversion is applied to a date in internal format. These conversions are used with the OCONV intrinsic function as well as in output format strings.

Conversion Codes

Conversion	Meaning and Usage
D{n}{s}	The value **n** is an optional single digit specifying the number of digits to be printed in the year field. 0, 1, 2, 3, and 4 are acceptable values (4 is default). The value **s** is an optional nonnumeric character to be used as the separator between day, month and year on output. If **s** is not specified, the abbreaviation of the month is returned. If **s** is specified, the numeric value of the month is returned.

DI	Internal date. This is a special case of date conversion. It allows you to convert from external to internal format as an output conversion, the inverse of the normal D conversion. This is a way to handle dates which are stored in a file in external format.
	
	Because DI is used as an output conversion, it may be specified as a DATA/BASIC format string to convert a date to internal format. (Remember that conversions within DATA/BASIC format strings are always output conversions. For example:
	
	`INPUT input.date`
	`internal.date = input.date 'DI'`

DD	Returns the day of the month.
DJ	Returns the Julian day of the year as a number from 1 to 365 (366 in a leap year).
DM	Returns the month as a number from 1 to 12.
DMA	Returns the name of the month.

Conversion	Meaning and Usage
DQ	Returns the quarter as a number from 1 to 4.
DW	Returns the day of the week as a number from 1 to 7, where Monday is 1 and Sunday is 7.
DWA	Returns the name of the day of the week.
DY{n}	Returns the year. If the optional 'n' is present and in the range of 0 to 4, it returns the rightmost 'n' digits of the year. If 'n' is not present or is in the range of 5 to 9, the year defaults to 4 digits.

Examples

In all of the following examples, DAY = DATE() = 6940 (DECEMBER 31, 1986).

Conversion	Result
PRINT OCONV(DAY,'D')	31 DEC 1986
PRINT OCONV(DAY,'D2')	31 DEC 86
PRINT OCONV(DAY,'D2/')	12/31/86
PRINT OCONV(DAY,'DWA')	WEDNESDAY
PRINT DAY 'DY'	1986
PRINT DAY 'DM'	12

***

And the final example:

```
PRINT DAY 'DWA':", ":DAY 'DMA':
 " ":DAY 'DD':", ":DAY 'DY'
```

Returns:

```
WEDNESDAY, DECEMBER 31, 1986
```

Appendix G. Using Denationalization from DATA/BASIC

Introduction

This appendix explains how to denationalize your DATA/BASIC code for users who may need to use different languages with the same program. For more information on denationalization, refer to the Developer's Guide to Denationalization.

There are no utility programs that implement denationalization on your DATA/BASIC programs, so you must follow a number of steps to make your program messages work properly.

Step 1

1. Logon to the account called DENAT. The following menu appears:

```
                    MAIN MENU

          1. Language table maintenance
          2. Error message file  maintenance
          3. Character tables maintenance
          4. List file names
          5. Exit to TCL
          6. Logoff

               Enter selection:
```

Figure G-1. DENAT Main Menu

Step 2

2. Select option 5, Exit to TCL.

   Create a file in the DENAT account for the text strings output by your DATA/BASIC program.

   You should create at least one file for each language used on the system.

   The number of files per language should be equal, and each file should be named according to its language. For example, a system in Montreal might have files named ENGLISH.MSGS and FRENCH.MSGS.

   When you have created as many files as you need, type **MENU <RETURN>** to return to the Main Menu.

Step 3

3. Add the language files to the denationalization tables so your programs can find the files for the language being used by the invoking process.

Step 3 (Continued)

Select option 1, Language table maintenance, from the Main Menu.

The following menu appears:

```
              Language maintenance

   1. Add a message
   2. Replace a message
   3. Delete a message
   4. Display a message
   5. Create a new message class
   6. Translate a message
   7. Load a message
   8. Define a new language

        Enter selection:
```

Figure G-2. Language Maintenance Menu

Select option 5, Create a new message class.

When you are prompted for a source file name, type in your language file name. A separate source file name must be typed for each language to be used by the program, but only one can be input at a time.

When you are prompted for a class number, type in a number not used by the denationalization system software. Any integer between 40 and 79 is valid if it is not already in use. If the class number you choose is in use, the following message displays:

```
[415] 'CLASSxx' EXISTS ON FILE
```

When you are prompted for a description of the class, type in a brief description of the messages, for example, APPLICATION.MSGS.FILES. Press <RETURN>. The following text displays:

Step 3 (Continued)

```
This routine is for the input of messages. Messages are keyed in as
normal input. All characters are accepted as input except for the
escape key, which is used to delimit hex strings from character strings.
The following characters have special meaning when entered as the last
character on any input line:
% -- will be replaced by a X'FE'
: -- will be replace by a X'FD'
+ -- will be replaced by a X'FC'. This will supress a cr/lf when the
                                  message is preceded by the PRTMSG or
                                  CRLFMSG routine.

Input message or <RETURN> to process
:
```

Figure G-3. Entering a Message

Enter the name of the appropriate message file (e.g., ENGLISH.1). After you enter each message, the system displays:

```
Message number 'n' inserted
```

Write down the message numbers for each file and store them in a safe place for future reference. When you have entered all the message files, press <RETURN> to terminate input.

Repeat this process for each language.

Step 4

4. Select option 7, Load a message, from the Language maintenance menu.

   When you are prompted for a source file name and message class, type in the entries you made in the previous steps.

   When you are prompted for a language number or name, type in the name of the destination language or the number that defines it. Type a question mark (?) to see a list of all the available languages.

   The following message displays:

   ```
   [9100] 'CLASSxx' loaded.
   ```

   Repeat these procedures for each language file on the system.

Step 5

5. Insert code blocks to handle the messages into your programs at every point where text strings are output.

   Do not hard code your program's messages. Place them in the files you defined in steps 1 through 3.

   Note: Good programming practice dictates that you make the item-id of each item containing messages for a program the same as the item-id of the program itself. This makes message maintenance much easier and keeps the item-ids segregated from your code.

   Your message handler code blocks should look something like the following example from an actual denationalization code segment:

```
DIM PROG.MSGS(35) ; * define an array to store the messages
MAT PROG.MSGS = '' ; * clear the array
*
FILE.NAME = GETMSG(14,5) ;* get name of file with messages
PROG.NAME = SYSTEM(40) ; * get name of current program
OPEN FILE.NAME TO MSG.FILE ELSE STOP 201,FILE.NAME
MATREAD PROG.MSGS FROM MSG.FILE,PROG.NAME ELSE STOP 202,PROG.NAME
```

Notes: PROG.MSGS(35) defines the number of messages used by this program as 35. PROG.MSGS is an array that stores the messages used by this program. Each message should be its own attribute.

GETMSG(14,5) defines the message class to be accessed as 14 and the message number to be obtained as 5.

SYSTEM(40) returns the name of the program.

Example

The following DATA/BASIC program, TEST.PROG in file BP, has not been denationalized.

```
TEST.PROG

CLEAR
OPEN 'TEST' TO TEST.FILE ELSE STOP 201,'TEST'
CRT @(-1):'INPUT ITEM ID': ; INPUT ID
LOOP
  READ ITEM FROM TEST.FILE,ID THEN
    CRT 'ITEM FOUND!'
    FOR X=1 TO 10
      PRINT ITEM<X>
    NEXT X
    LOOP
      CRT ; CRT 'EVERYTHING OK?': ; INPUT ANS,1_
    UNTIL ANS='Y' DO
      CRT 'LINE NUMBER TO CHANGE': ; INPUT LINE.NO
      CRT 'NEW DATA': ; INPUT LINE
      ITEM<LINE.NO>=LINE
    REPEAT
    WRITE ITEM ON TEST.FILE,ID
  END ELSE
    CRT 'ITEM NOT ON FILE' ; RQM
  END
REPEAT
END
```

To make this program usable to speakers of any language, you must take a number of steps to create the appropriate files and items.

The steps to be followed, in the order given, are:

1. Create a new message class (40, for example) in the DENAT account that contains only one message. The message is ENGLISH.APPL.MSGS.

   Note that this is message 0 of class 40.

2. Create a file called ENGLISH.APPL.MSGS in the BP account.

3. Create an item called TEST.PROG in the ENGLISH.APPL.MSGS file in the BP account. Put all the messages used by the DATA/BASIC program TEST.PROG into the TEST.PROG item.

   Note that the names of the DATA/BASIC program and the item are identical. The identical names are not required, but they do make it easier to maintain the system.

Example (Continued)

TEST.PROG would look like this:

```
     TEST.PROG
001  INPUT ITEM ID
002  ITEM FOUND
003  EVERYTHING OK?
004  LINE NUMBER TO CHANGE
005  NEW DATA
006  ITEM NOT ON FILE
007  Y
```

Each attribute of the item corresponds to one of the messages output by the DATA/BASIC program called TEST.PROG.

4. To make the DATA/BASIC program TEST.PROG work with the denationalized system, insert the following code segment into the program following the CLEAR statement:

```
DIM PROG.MSGS(10)
MSG.FILE=GETMSG(40,0)
OPEN MSG.FILE TO MESSAGES ELSE STOP 201,MSG.FILE
PROG.NAME=SYSTEM(40)
MATREAD PROG.MSGS FROM MESSAGES,PROG.NAME ELSE
 STOP 202,PROG.MSG
```

This code reads the messages from the item TEST.PROG in the ENGLISH.APPL.MSGS file and stores them in the variable PROG.MSGS. You can then retrieve each message as needed from PROG.MSGS and display it on the terminal.

Note that the parameter set to 40 in the GETMSG function defines the number of the class being accessed and the parameter set to 0 defines the message number within that class. In this case, 40 is the message class called CLASS40 and 0 is the message ENGLISH.APPL.MSGS within that message class.

The fully denationalized version of the TEST.PROG program appears below.

Example (Continued)

```
TEST.PROG

CLEAR
DIM PROG.MSGS(10)
MSG.FILE=GETMSG(40,0)
OPEN MSG.FILE TO MESSAGES ELSE STOP 201,MSG.FILE
PROG.NAME=SYSTEM(40)
MATREAD PROG.MSGS FROM MESSAGES,PROG.NAME ELSE
 STOP 202,PROG.MSG
OPEN 'TEST' TO TEST.FILE ELSE STOP 201,'TEST'
CRT @(-1):PROG.MSGS(1): ; INPUT ID
LOOP
  READ ITEM FROM TEST.FILE,ID THEN
    CRT PROG.MSGS(2)
    FOR X=1 TO 10
      PRINT ITEM<X>
    NEXT X
    LOOP
      CRT ; CRT PROG.MSGS(3): ; INPUT ANS,1_
    UNTIL ANS=PROG.MSGS(7) DO
      CRT PROG.MSGS(4): ; INPUT LINE.NO
      CRT PROG.MSGS(5): ; INPUT LINE
      ITEM<LINE.NO>=LINE
    REPEAT
    WRITE ITEM ON TEST.FILE,ID
  END ELSE
    CRT PROG.MSGS(6) ; RQM
  END
REPEAT
END
```

DOCUMENTATION COMMENTS

Title:______________________________ Publication Number:______________

Did you find any errors? If so, please be specific and indicate the page number on which the error is found.

____

____

____

Did you find this document understandable, usable and well organized? Please make suggestions for its improvement.

____

____

____

Is any material missing? If so, please describe and indicate where it should be placed.

____

____

____

Name:____ Date:_______________

Position:____________________ Organization:__________________________

Street:____ Phone:________________

City:________________________________ State:________ Zip:_____________

This form is for document comments and corrections only. Any system problems are to be reported to your appropriate support group.

Please use the back of this form or attach additional sheets if you have further comments.

Please return this form to McDonnell Douglas Computer Systems Company, P.O. Box 19501, Irvine, CA 92713 Attn: End User Documentation, Mail Stop: RYN