Ultimate
THE ULTIMATE CORP.

BASIC Language Reference Guide

Ultimate

THE ULTIMATE CORP.

BASIC Language Reference Guide

The Ultimate Corp.
East Hanover, NJ

Version 3

Ultimate BASIC Language Reference Guide
Version 3.0

How to order this guide:

The Ultimate BASIC Reference Guide is included with the system documentation set.

To obtain additional copies, please call your dealer.

Publication Information

Operating System Release 10, Revision 210

Document No. 6929-3

Contents

Appendices

Figures

Tables

Notes

How to Use This Manual

This manual is intended as a reference for programmers using the Ultimate BASIC programming language. It covers all aspects of Ultimate BASIC through revision 210 of the Ultimate operating system.

BASIC is a simple yet versatile programming language that was first developed at Dartmouth College in 1963 and is suitable for expressing a wide range of problems. The Ultimate version has been extensively modified to support the unique features of the Ultimate data base structure and operating system.

How the Manual is Organized

Chapter 1 gives an overview of programming with Ultimate BASIC. It covers the program file structure, components of a program, compiler options, and methods of executing programs.

Chapter 2 discusses how data can be represented in a BASIC program: as constants (literals), variables, or arrays. It also covers the use of expressions (arithmetic, logical, string, and relational), and the standard vs. extended arithmetic (floating point and string) operations.

Chapter 3 lists all statements and functions in alphabetical order. Each statement and function is detailed in a single-topic unit.

Chapter 4 explains each command in the BASIC debugger and gives an example of the use of the debugger.

Chapter 5 reviews the Ultimate data file structure and gives some recommended coding techniques. The chapter also contains several sample programs for reference.; these programs illustrate the use of Ultimate BASIC for file updating, job control, and other applications.

The appendices list compiler and runtime messages, debugger messages, ASCII codes, standard user exits from BASIC, the USERMSG file, and features introduced in revision 200.

Conventions

This manual presents the general syntax for each BASIC statement and function. In presenting and explaining the syntax, the following conventions apply:

Convention	Description
UPPER CASE	Characters printed in upper case are required and must appear exactly as shown.
lower case	Characters or words printed in lower case are parameters to be supplied by the user (for example, line number, data, etc.).
{ }	Braces surrounding a parameter indicate that the parameter is optional and may be included or omitted at the user's option.
bold	Boldface type is used for section and unit headings. It is also used in examples to indicate user input as opposed to system displayed data.
RETURN	The RETURN symbol indicates a physical carriage return pressed at the keyboard. A RETURN is required to complete a command line, and signals the system to begin processing the command.
<key>	Angle brackets are used to indicate a key other than letters or numbers; for example <ESC>.
enter	The word enter is used to mean "type in the required text, then press RETURN."
X'nn'	This form is used to define a hexadecimal number where 'nn' is the hex value; for example, X'0B', X'41', X'FF'.
RND(expr)	All functions require a set of parentheses, which usually enclose a parameter. No space is allowed between the function name and the left parenthesis.
`Enter option`	This typeface is used for messages and prompts displayed by the system.

1 Introduction

This manual describes the Ultimate BASIC programming language, which is an extended version of Dartmouth BASIC.

Ultimate BASIC includes the following features:

- Compiled object code
- Optional alphanumeric or numeric statement labels of any length
- Multiple statements on one line
- Single statements on multiple lines
- String handling with variable length strings
- String and numeric format masking
- Shared source code between programs
- Linked programs
- Computed GOTO and GOSUB statements
- Complex and multi-line IF statements
- CASE statement selection
- External subroutine calls
- Magnetic tape input and output
- Fixed point, floating point, and string arithmetic
- Data conversion capabilities
- Ultimate file access and update capabilities
- File level or group level lock capabilities
- Pattern matching
- Dynamic arrays
- Job control capabilities
- Debugging language
- Variably dimensioned arrays

The File Structure of BASIC Programs

A BASIC source program is stored as an item in the data section of a file. The program name is its item.id. Each individual line within the BASIC program is stored as an attribute in the item.

When a program is successfully compiled, the compiler generates a pointer to the object code and stores this pointer in the dictionary section of the file, using the program name as the pointer name. Thus, in order to compile programs, the data and dictionary sections must be distinct files.

Object pointer items have a format similar to that of POINTER-FILE save-list items:

Attribute	Description
item.id	program name
01	CC
02	starting frame number of object code
03	number of frames of object code
04	null
05	time and date of compilation

Attributes 0 through 4 are protected by the system against alterations by the Editor or any other file-updating program.

Note: *Frame number is also referred to as the* frame identifier *or* FID.

Stored along with the object code of each program (unless suppressed at compile time) is a symbol table for use with the BASIC debugger. The symbol table contains all variable names defined in the program. (For details on the BASIC debugger, refer to Chapter 4, BASIC Debugger.)

When object pointer items are saved on tape as part of a FILE-SAVE or ACCOUNT-SAVE, the associated object code is also saved. Individual object programs may also be saved on tape by T-DUMPing specified pointers in a file dictionary. Programs may be restored from FILE-SAVE and ACCOUNT-SAVE tapes using ACCOUNT-RESTORE or SEL-RESTORE (specifying a file dictionary). Object programs may be T-LOADed into file dictionaries from T-DUMP tapes.

The Components of a BASIC Program

A BASIC program consists of a sequence of BASIC statements. Each BASIC statement tells the system to perform a specific program operation. A statement may include one or more data values, expressions, and/or intrinsic functions. (Please refer to Chapter 2 for details on representing data and expressions. Refer to Chapter 3 for an alphabetical listing and discussion of each BASIC statement and intrinsic function.)

Multi-Statement Lines

More than one statement may appear on the same program line, separated by semicolons. For example:

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

Multi-Line Statements

Certain statements which take an indefinite number of arguments may be continued on several lines; each line except the last must end with a comma. For example:

```
CALL A.BIG.SUBROUTINE(LONGPARAMETERNAME1,
  LONGPARAMETERNAME2,
  EVEN.LONGER.PARAMETERNAME3)
```

The continued lines may be indented to improve program clarity, but this is not required by the BASIC Compiler. Statements with the multi-line option are noted in their individual discussions.

Labels

Any BASIC statement may begin with an optional statement label that can be either numeric or alphanumeric.

Numeric statement labels may be any constant number. The following INPUT statement, for example, has a statement label of 100:

```
100 INPUT X
```

Alphanumeric statement labels may contain letters, numbers, dollar signs, and periods, but the first character must be a letter. An alphanumeric label, when it is defined, must be followed by a colon.

When an alphanumeric label is referenced, the colon is not used. The colon is optional in defining numeric labels.

The following routine defines the statement label INPUTLOOP and references itself and two other labels:

```
INPUTLOOP:  GOSUB GETINPUT
   GOSUB DOIT
   GOTO INPUTLOOP
```

A label can be the only text on a line, in which case it labels the next non-blank non-null line. For example:

```
DOIT:
   GOSUB DOITAGAIN
```

Compiler Directives

A BASIC program can also include compiler directives. Directives look similar to BASIC statements, but they affect the way a program is compiled, not the way it runs. The following compiler directives are available:

$*	inserts comments into object code
$CHAIN	links program file items
$COMPATIBILITY	compiles according to alternative standards
$INCLUDE	shares source code between programs
$INSERT	equivalent to $INCLUDE
$NODEBUG	omits source line references and symbol table, which limits debugging capabilities
INCLUDE	equivalent to $INCLUDE

The compiler directives are described in Chapter 3, BASIC Statements and Functions.

Use of Blanks

Except for situations explicitly called out in the following sections, blank spaces appearing in the program line and that are not part of a string are ignored. All-blank lines and null lines (containing no text and no blanks) are also ignored. Thus, blanks and null lines may be used freely within the program for purposes of clarity and readibility.

Remarks

A helpful feature to use when writing a BASIC program is the REMark statement. A REMark statement is used to explain or document the program. It allows the programmer to place comments anywhere in the program without affecting program execution. (The REMark statement, which can be written as REM, !, or *, is described in Chapter 3.)

Figure 1-1 uses a simple BASIC program to show overall program format. Figure 1-2 illustrates the same program with a number of REMark statements and a null line added for clarity.

End of Program

An Ultimate BASIC program does not require any special end of program command; however, an END or STOP statement can be used if desired. The compiler always places a STOP command after the last line in the program.

```
   I = 1
5  PRINT I
   IF I = 10 THEN STOP
   I = I + 1
   GO TO 5
   END
```

Figure 1-1. Sample BASIC Program

```
REM  PROGRAM TO PRINT THE
*    NUMBERS FROM ONE TO TEN
*
   I = 1                  ;* start with one
5  PRINT I                ;* print the value
   IF I = 10 THEN STOP    ;* stop if done
   I = I + 1              ;* increment I
   GOTO 5                 ;* continue
END
```

Figure 1-2. BASIC Program With Remark Statements

Creating BASIC Programs

BASIC programs are created and edited using one of the system editors. To invoke an editor, issue one of the following commands at TCL:

ED{IT} file.name program.name
SE file.name program.name
EEDIT file.name program.name

The EDIT verb calls the line editor. The SE verb calls the full screen editor. The EEDIT verb performs the same function as EDIT, but compresses the storage space used by eliminating all spaces when the item is filed.

Program listings are easier to follow when you indent statements within a loop or routine. You may set tab stops at TCL or within either editor. Figure 1-3 shows the commands in the line editor. (For details about using either editor, see the *Editor/Runoff User Guide*.)

The program is stored in the file specified by filename using the program name as the item.id.

```
:TABS I 4,8,12 <CR>                    User sets input tabs at TCL

:ED BP COUNT <CR>                      User edits item 'COUNT' in file
New Item                               'BP' (Basic Programs)
Top
I <CR>                                 User enters input mode
001+* PROGRAM COUNTS FROM 1-10 <CR>
002+   FOR I = 1 TO 10 <CR>            tab once
003+      PRINT I <CR>                 tab twice
004+   NEXT I <CR>                     tab once
005+END <CR>
006<CR>
Top
.FI <CR>                               User files item
'COUNT' filed.
```

Figure 1-3. Creating BASIC Program

Compiling BASIC Programs

After the BASIC program has been filed, it can be compiled. Compiling a program creates object code, which can then be executed using the RUN verb, or the program can be cataloged, then executed directly from TCL. The symbol table is also included with the object code (unless suppressed by the S option or $NODEBUG directive).

If either EDIT or SE was used to create the program, two TCL verbs are available to compile programs and create the object code: COMPILE and BASIC. Either may be used since they perform the same operation. If the EEDIT verb was used to create the program, the EBASIC form of BASIC must be used to compile the program. EBASIC expands the item to include any spaces that were compressed by EEDIT.

Syntax

BASIC file.name item.list {(options)}
COMPILE file.name item.list {(options)}
EBASIC file.name item.list {(options)}

item.list may contain one or more explicit item.ids (program names) separated by one or more blanks, or may be an asterisk (*) to indicate all programs in the file.

options if used, options must be enclosed in parentheses; multiple options used in a single command should be separated by commas. The valid options are

C suppress end-of-line (EOL) opcodes from object code. This eliminates one byte of run-time object code for every line of source code. The EOL opcodes are used to count lines for error messages. This option is designed to be used with debugged cataloged programs; any run time error message that occurs in a program compiled with this option specify a line number of 1.

F used with the M option to list internal variables and labels, including those created by IF/THEN and FOR/ NEXT loops; internal variables and labels are displayed preceded by an asterisk.

I if L option is specified, also list lines from $INCLUDEd programs.

L list BASIC program; generates a line by line listing of the program during compilation. Error lines with associated error messages are indicated.

M list map of BASIC program; generates a variable map and a statement map that show where the program data will be stored in the user's workspace. The variable map lists the offset from the beginning of the descriptor table of every BASIC variable in the program. The display is similar to the following:

```
Symbol table is 2% full
Last variable is at 210
   ---------- V A R I A B L E S ----------
   30 REPLY   40 FEXISTS  50 FTYPE   60 TIME
   70 HH      80 MM      100 N      150 MODE
  -------------- L A B E L S --------------
   55 PRINTID 59 ERRORPR
  ------------ E Q U A T E S -------------
 BELL=CHAR(7)   CR=CHAR(13)   ESC=CHAR(27)
```

The variable locations are given as offsets from the beginning of the descriptor table. The gaps in the table are either because a variable is a dimensioned array or because there are CALLs to subroutines between two definitions. The location of the last variable shown above the variables may be greater than the last location shown in the table for the same reasons. In addition, offsets 10 and 20 are never displayed; offset 10 is used for the internal default file variable and offset 20 is used for the internal default select variable. (The descriptors used for subroutines and for internal variables will be displayed if the F option is also specified.) Descriptors are ten bytes in length.

The number preceding each label is the line number where the label is defined. If the program is compiled with the C option, the line number is always 1; if there is a

$NODEBUG directive in the program itself, the line number is always the line number of the $NODEBUG statement.

N no page; inhibits automatic paging on terminal when using the L and M options.

P print compilation output on line printer

S suppress generation of symbol table; suppresses saving the symbol table generated during compilation. The symbol table is used exclusively by the BASIC debugger for reference; therefore it must be kept only if the user wishes to use the debugger to display or manipulate variables.

X cross-reference all labels and variables used in a BASIC program and stores this information in the BSYM file. (***Note**: A BSYM file must be created prior to using this option.*)

The X option first clears the information in the BSYM file, then creates an item for every variable and label used in the program. The variable or label name is used as the item.id. Each line number where the variable or label is referenced is placed as a value in attribute 1. An asterisk precedes the line number where a label is defined or where the value of a variable is changed.

The output is not displayed by this option; use RECALL to sort the file. To format the listing, create an attribute definition item in the dictionary of the BSYM file for attribute 1, called something such as line-number, then use a SORT command to create a cross reference listing of the program to be generated:

```
:SORT BSYM BY LINE-NUMBER LINE-NUMBER
```

Description

The BASIC compiler displays a message when an error is encountered; the compiler also indicates where on the line it was scanning when it noted the error. For example, if the THEN/ELSE clause is missing in an OPEN statement, the compiler displays an error message similar to the following:

```
004 OPEN 'BP' TO BP
***                  ^ THEN or ELSE clause missing
```

After the program is compiled, the system no longer needs the source program, which can then be deleted, if desired.

Note: The compile process does not create an item in the Master Dictionary (MD); to create an item in MD from the compiled program, use the CATALOG command. The compile-and-go format can be used to place a BASIC source program in the MD (for information on compile-and-go, see the section Executing BASIC Source Programs.)

The BASIC compiler stores a compiler version number in each program's object code. At run-time, before running a program, the system checks the program's compiler version number to see if it is compatible with the current compiler version. If it is not, the program is not allowed to run and the system issues an error message, which indicates that the program must be recompiled before it can be run.

The maximum BASIC object code size is 57,534 bytes.

The BASIC, COMPILE, and EBASIC commands are also discussed in the *Ultimate System Commands Reference Guide*.

Example	Description

`:COMPILE BP COUNT <CR>`
`*****`
`[B241] Line 5, 'COUNT' successfully compiled; 1 frames used.`
	compile command

Cataloging BASIC Programs

The CATALOG verb is used to catalog compiled BASIC programs into the user's master dictionary; after the program is cataloged, its program name can be used as a command at TCL.

Syntax

CATALOG file.name item-list {(L{)}}

file.name file containing programs to be cataloged

item-list one or more program names (item.ids), or "*" to indicate all programs in the file

L indicates that the program is not to be automatically executed at logon time, if the name of the program is the same as the name of the account in which the program is cataloged. If the L option is not present, a cataloged program with the same name as the current account is automatically executed whenever a user logs on to the account. (For details about executing programs at log on, please refer to the section, Executing BASIC Programs.)

Description

A program must be compiled before it can be cataloged.

For each program successfully cataloged, the system responds with

```
[244]  'item.id' cataloged.
```

The CATALOG verb adds the program to the MD as an item with the following form:

Attribute	Description
item.id	program name
001	PC
002	E6
003	
004	
005	file.name item.id

If the program was cataloged using the L option, attribute one of the verb definition is P rather than PC.

After a program has been cataloged, it can be executed by entering its name at the TCL prompt, using the following general format:

```
:programname {argument list}
```

The programname must be entered exactly as the program name is stored in the user's Master Dictionary. The optional argument list contains any parameters that need to be passed to the program.

The external subroutines used with the BASIC CALL statement may also be cataloged, although it is unnecessary if both the subroutine and the calling routine are in the same program file. The CALL statement first searches the Master Dictionary for a cataloged verb; if no verb is found, CALL then looks in the dictionary of the program file for the calling routine.

The program to be cataloged cannot have the same name as an existing item in the user's Master Dictionary unless that item is also a cataloged program. If a conflicting item exists in the user's Master Dictionary, a message similar to the following is displayed and the program is not cataloged:

```
[415] 'item.id' exists on file
```

Decataloging BASIC Programs

The DECATALOG verb deletes the Master Dictionary reference to the program and removes the object code from the system.

Syntax

DECATALOG file.name item-list

file.name file containing programs to be decataloged

item-list one or more program names (item.ids), or "*" to indicate all programs in the file

Description

DECATALOG removes the object programs by deleting the appropriate pointer items from the dictionary of the file; the associated frames containing the object code are returned to the system's pool of available frames (overflow). DECATALOG also deletes the verbs for cataloged programs from the Master Dictionary, but a program does not have to be cataloged before it is decataloged.

The CATALOG and DECATALOG commands are also discussed in the *Ultimate System Commands Reference Guide*.

Executing BASIC Programs

BASIC programs can be executed in the following ways:

- a cataloged BASIC program can be executed by issuing the program name at TCL
- the RUN verb issued at TCL can be used to execute a compiled BASIC program
- a cataloged BASIC program with the same name as an account name can be automatically executed at logon time
- a source program that is stored in the master dictionary and that has a PROGRAM statement as the first line can be compiled and executed by issuing only the program name at TCL
- programs can be executed as part of another BASIC program or as part of a PROC

RUN Command

The RUN verb is used to execute programs that have already been compiled.

Syntax

RUN filename item.id {argument list} {(options)}

filename	file containing program to be executed
item.id	program to be executed
argument list	parameters that must be passed to the program
options	if used, the options must be enclosed in parentheses; multiple options may be separated by commas. Valid options are as follows:

A abort option; inhibits entry to the BASIC debugger under all error conditions; instead, if an error occurs, the program prints the error message and terminates execution

D run-time debug option; causes the BASIC debugger to be entered before the start of program execution. Note that the BASIC debugger may also be called at any time while the program is executing, by pressing the BREAK key on the terminal

E errors option; forces the program to enter the BASIC debugger whenever an error condition occurs. The use of this option forces the operator either to accept the error by using the debugger, or to exit to TCL

I inhibit initialization of data area (refer to the description of the BASIC CHAIN statement)

N nopage option; cancels the default wait at the end of each page of output when that output has been routed to the terminal by a program using the HEADING, FOOTING, and/or PAGE statements

P printer on (has same effect as issuing a BASIC PRINTER ON statement). Directs all program output to the Spooler

S suppress run-time warning messages.

Executing Programs at Logon Time

When a user logs on, the system attempts to execute an item in the user's Master Dictionary with the same name as the logon account name. This item can be a cataloged BASIC program, a compile and-go BASIC program, or a PROC. This feature is useful to run a standard job control sequence or present a custom-tailored menu of choices to the user.

However, you may need to catalog a BASIC program with the same name as the name of the account, but you do not want it to run automatically at logon time. To avoid automatic execution, the program should be cataloged with the L option.

For details on cataloging programs, refer to the section, "Cataloging BASIC Programs".

Executing BASIC Source Programs (Compile and Go)

BASIC source programs can be stored as items in Master Dictionaries and can be executed from TCL without previous compilation. This option, called "compile-and-go", requires only that the source program have a PROGRAM statement beginning at the first character (no leading blanks) of line one. The PROGRAM statement can be abbreviated as PROG. For example:

```
  HELLO
001 PROG
002   PRINT "HELLO"
003 END
```

The general format for running the program is:

program.name {argument list}

The effect of compile-and-go is similar to that of writing a PROC, but with BASIC's more powerful run-time and debugging features. Compile-and-go programs can be executed at logon time if the program name is the same as an account name.

Note: When a compile-and-go program has been established in a user's Master Dictionary, that name cannot be used as the name of another program when it is cataloged.

Using BASIC for Job Control Tasks

BASIC programs can be used for job control tasks by executing BASIC programs, PROCs, and TCL verbs within a controlling BASIC program. The controlling program can use EXECUTE statements, as well as other supporting statements (PUT, GET, SEEK) and functions (EOF) to implement the job control tasks.

The BASIC program can control error processing by using TRAP ON THEN CALL statement. This statement can trap program terminations, error conditions, pressing of the BREAK key, and commands entered from the BASIC and system debuggers.

For details on using these statements, please refer to the appropriate statement name, listed alphabetically in Chapter 3 of this manual.

2 Working with Data

This section describes the features of BASIC that are available for working with data. It also describes the way in which the system allocates variables. The following features are discussed:

- reserved words
- numbers and numeric data
- string data
- arrays
- arithmetic expressions
- string expressions
- concatenation
- format strings: numeric mask and format mask codes
- relational expressions
- logical expressions
- summary of expression evaluation
- limited expressions
- variable data area
- variable allocation

Reserved Words

Figure 2-1 is a list of BASIC reserved words. These words cannot be used as simple variable names, array variable names, or labels.

AND	GOTO	OUT.
ARG.	GO	PASSLIST
CAPTURING	GT	REPEAT
CASE	IN.	RETURNING
CAT	LE	RTNLIST
DO	LOCKED	SELECT.
ELSE	LT	STACKING
END	MATCH	STEP
EQ	MATCHES	THEN
FROM	NE	TO
GE	NEXT	UNTIL
GLE	ON	WHILE
GOSUB	OR	

Figure 2-1. BASIC Reserved Words

Figure 2-2 is a list of BASIC functions; the names of these functions cannot be used as array or matrix variable names. Ultimate strongly advises against using these names as simple variables and labels, although the compiler allows such use.

@	FIELD	RND
ABS	FMT	SADD
ALPHA	FMUL	SCMP
ASCII	FSUB	SDIV
CHAR	ICONV	SEQ
COL1	INDEX	SIN
COL2	INDEXINFO*	SMUL
COS	INMAT	SORT
COUNT	INSERT	SOUNDEX
DATE	INT	SPACE
DCOUNT	LEN	SQRT
DELETE	LN	SSUB
EBCDIC	MATCHFIELD	STR
EOF	MAXIMUM*	SUM
ERROR	MINIMUM*	SYSTEM
EXP	MOD	TAN
EXTRACT	NOT	TIME
ERRTEXT	NUM	TIMEDATE
FADD	OCONV	TRIM
FCMP	PWR	TRIMB
FDIV	REM	TRIMF
FFIX	REPLACE	USERTEXT
FFLT	REUSE	

*Reserved for future use

Figure 2-2. BASIC Functions

Numbers and Numeric Data

Numbers may be represented in Ultimate BASIC in three formats:

- fixed
- floating point
- string

Each format has its own arithmetic operators. For both floating point arithmetic and string arithmetic, the standard operations of add, subtract, multiply, divide, and compare have been implemented as functions within BASIC.

Fixed Point Numbers

A fixed point number may contain any number of digits to the left of the decimal point and can have a maximum of nine digits to the right of the decimal point. The actual number of digits is determined by the PRECISION statement; the default number is four. (For details, see the description of the PRECISION statement in Chapter 3.)

The unary minus sign is used to specify negative numbers. For example:

```
-17000000
-14.3375
```

The fixed point arithmetic operators are

^	exponentiation
*	multiplication
/	division
+	addition
–	subtraction

Floating Point Numbers

Floating point numbers have a different format from fixed point numbers. A floating point number consists of a mantissa and an exponent. Ultimate BASIC floating point uses an integer mantissa and a base-10 exponent. The mantissa may contain from 1 to 13 digits and may be either positive or negative. A negative mantissa uses a minus sign in front of it; a positive mantissa is unsigned. The exponent may be in a range of -255 to 255. Like the mantissa, a negative exponent uses a minus sign; a positive exponent is unsigned. An E is used to separate the mantissa from the exponent.

Values to be used as floating point numbers must be specially formatted strings. Functions are provided that convert fixed point numeric or string numeric values to floating point format. Another set of functions may be used after floating point operations to convert the results back to fixed point numeric or string values.

The following examples show the floating point representation of various numbers:

Floating Point Representation	Expanded Number
0E0	0
1E0	1
1E3	1000
1E-20	.00000000000000000001
-1234567890123E-5	-12345678.90123
9876543210987E-13	.9876543210987
-28855E-2	-288.55

The following functions are available for arithmetic operations on floating point numbers:

Operation	Floating Point Function
Addition	FADD
Subtraction	FSUB
Multiplication	FMUL
Division	FDIV
Comparison	FCMP
Convert to floating	FFLT

Convert to fixed	FFIX

For details, see the description of each function in Chapter 3.

String Numbers

A string number, that is, a number that is enclosed in string delimiters, can have any magnitude and any precision.

The following examples show various string numbers:

```
"1"
"-300.23"
"500000000000000000000000"
".00000000000000000000000023"
```

String numbers can use any of the numeric operators shown with the fixed point numbers. However, these operators restrict the number of decimal places in the result of the arithmetic operation to the current precision.

The following set of functions have been defined for string arithmetic and, except for SDIV, do not restrict the number of decimal places. The divisor in SDIV is restricted to 13 significant digits and the quotient is restricted to 14 significant digits.

Operation	String Function
Addition	SADD
Subtraction	SSUB
Multiplication	SMUL
Division	SDIV
Comparison	SCMP

For details, see the description of each function in Chapter 3.

String Data

A string may contain any number of characters. A string is defined by a set of characters enclosed in single quotes('), double quotes ("), or backslashes (\); these characters are described as string delimiters.

The following are examples of strings:

```
"THIS IS A STRING"
'ABCD1234#*'
\3A\
```

If a string value contains a character that can also be a used as a string delimiter, then another delimiter must be used to delimit that string.

```
"THIS IS A 'STRING' EXAMPLE"
'THIS IS A "STRING" EXAMPLE'
```

Internally, a string is delimited by a segment mark (SM), which is a character having a decimal value of 255. A string may not include a segment mark.

A string may include data delimited by system delimiters (attribute marks, value marks, and subvalue marks). Such strings are called "dynamic arrays" and are described in the section "Arrays", starting on page 2-12.

Constants and Variables

Numeric and string data values may be represented as either constants or variables.

Constants

A constant, as its name implies, has the same value throughout the execution of a program. A constant may be a literal value such as the number 2 or string "HELLO", or it may be a named value. In this case, a symbolic name is equated with a constant value; for example, the name "AM" could be equated to CHAR(254); and the name can be used instead of the value in BASIC statements.

Variables

A variable has both a name and a value. The value of a variable may be either numeric or string, and may change dynamically during the execution of the program.

Storage space for variables is allocated in the order that the variables appear in a program. No special statements are needed to allocate space for simple variables (except COMMON variables), but the size of each dimensioned array must be specified in a DIM or COMMON statement to allocate its space.

The maximum number of variables in a program is 3223. If an array is dimensioned to a literal number of elements, each element counts as one variable. An array that is dimensioned to a variable number of elements counts as only one variable, regardless of the value of the variable. For more information, see the section Arrays, which starts on page 2-12.

Variable Names

Variables are identified by a variable name; the name remains the same throughout program execution. Variable names consist of an alphabetic character followed by zero or more letters, numerals, periods, or dollar signs. Variable names may be of any length.

The following terms are valid variable names:

```
X
QUANTITY
DATA.LENGTH
B$..$
```

BASIC reserved words may not be used as variable names (the BASIC reserved words are listed at the beginning of this chapter).

Although a BASIC variable name may end with a period (.), it is recommended that programmers not use names in this format for their own variables in order to distinguish the variables predefined by the Ultimate operating system. Since variable names in this format may or may not be treated as names of predefined variables in all cases, depending on the operating system release, The Ultimate Corp. strongly suggests programmers rewrite their software, if necessary, to avoid possible conflict.

Values of Variables

The value of a variable may change during the execution of the program. The variable X, for example, may be assigned the value 100 at the start of a program, and may later be assigned the value "THIS IS A STRING".

A program can retrieve the value of a variable by specifying the variable name. For example, the following program lines assign the value 12 to A, then print the value of A:

```
A = "12"
PRINT A
```

Predefined Symbols

The following symbols have been preassigned values and can be used in place of variables:

@FM field mark; this has the value CHAR(254)

@VM value mark; this has the value CHAR(253)

@SM sub-value mark; this is has the value CHAR(252)

System Variables

The following symbols return information based on the current status of the system:

@EXECLEVEL — returns current EXECUTE level; equivalent to SYSTEM(21)

@HOLDFILE — number of last hold file created by PRINT statement in current BASIC program; if no hold file has been assigned, returns zero; equivalent to SYSTEM(22)

@LANGUAGE — returns two-digit language code of the language assigned to current port; equivalent to SYSTEM(27)

@PRIVILEGE — returns 0, 1, or 2 to indicate system privilege level of current user; equivalent to SYSTEM(23)

@SELECT — returns 1 if external select list is active, else returns 0; equivalent to SYSTEM(25)

@SENTENCE — returns TCL statement that invoked current program; statement is formatted as dynamic array; equivalent to SYSTEM(18). Elements in the statement are separated by attribute marks. If an element is enclosed in delimiters, the delimiters are removed. For example, if a program is invoked using the following command:

```
RUN BP PGM1 A 'B,C' (D)
```

the following is returned in @SENTENCE within PGM1:

```
RUN^BP^PGM1^A^B,C^(D)
```

@SPOOLOPTS — returns current spooler assignment status; equivalent to SYSTEM(24)

@USERNO — returns current port number; equivalent to SYSTEM(19)

@WHO	name of current user; does not return name of any CHARGE-TO account; equivalent to SYSTEM(26)

File Variables

A file variable is the variable to which a file is opened and contains information the system needs to locate the file. The file variable can be used in a BASIC PRINT statement or BASIC debugger / (list) instruction to display the base frame number (FID) of the file.

In addition, the file variable can be tested to see if any file has been opened to it. A non-zero value indicates the file is opened.

If the file variable is included in a COMMON statement, the file information assigned to it can be passed to subsequent programs.

If the file variable is changed in any way by the BASIC program, it is no longer considered a file variable.

```
OPEN 'TEST.FILE' TO TF ELSE STOP
   .
   .
PRINT 'Base frame ID:  ':TF
```

Prints base frame ID as given in the file identification item in the file dictionary.

```
OPEN 'TEST.FILE' TO TF ELSE TF = 0
   .
   .
IF TF THEN
   READ ITEM FROM TF,'T1' ELSE ITEM = ' '
   END ELSE
   ITEM = 'No test file available'
END
PRINT ITEM<1>
```

Arrays

Arrays are variables with multiple elements. Ultimate BASIC supports two types of arrays: dimensioned and dynamic.

A dimensioned array is defined by a DIM or COMMON statement. The exact number of elements can be fixed in the defining statement, or the number can be specified in a variable and determined at run time. A dynamic array is a string that contains elements delimited by attribute marks, value marks, and subvalue marks.

An array is associated with multiple storage locations, each of which has a separate value and which can function as a simple variable. A particular location (or element) within an array is specified by following the array name with subscripts (numbers or other arithmetic expressions).

Elements in dimensioned arrays are referred to with subscripts in parentheses. For example, if A defines a dimensioned array, A(10) refers to the tenth element of the array. Elements in dynamic arrays are referred to with subscripts in angle brackets. The first subscript specifies the attribute, the second subscript specifies the value, and the third subscript specifies the subvalue. For example, if X is a dynamic array, X<3> refers to the third attribute of the dynamic array; X<3,1,2> refers to the second subvalue in the first value in the third attribute of the dynamic array.

A dynamic array can be an element of a dimensioned array. An element within the dynamic array is referred to by placing the dynamic array subscript after the dimensioned array subscript. For example, if A defines a dimensioned array, A(10)<3> refers to the third attribute of the dynamic array in the tenth element of the dimensioned array.

Dynamic arrays, which are strings, should not be confused with dimensioned arrays, which are sets of storage locations. Unlike dimensioned array elements, the individual attributes, values, and subvalues of a dynamic array are not directly addressable, and are searched for on each reference since they may move as the dynamic array changes.

These two array types are described in detail in the following two sections.

Dimensioned Arrays

A BASIC program can address any element of a dimensioned array as a separate variable and can assign values to the individual elements or to the entire array.

A dimensioned array contains one value per element. Any array element may be accessed by specifying its position in the array as a subscript following the array name. For example, if array A has been dimensioned as A(4) and assigned values, it might look similar to the following:

3	A(1) has value 3
8	A(2) has value 8
-20.3	A(3) has value -20.3
ABC	A(4) has string value "ABC"

The above example illustrates a one-dimensional array. A two-dimensional array is characterized by having rows and columns. For example, if array Z has been dimensioned as Z(3,4) and assigned values, it might look similar to the following:

3	XYZ	A	-8.2	Z(1,2) has string value 'XYZ'
8	ABC	500	.333	Z(2,2) has value 'ABC'
2	XYZ	Q12	84	Z(3,2) has value 'XYZ'

The MATREAD{U} statement can be used to assign each attribute of an item to an individual array element. Conversely, the MATWRITE{U} statement can be used to write an item from an array. The MATPARSE statement can be used to assign values in a dynamic array to corresponding elements in a dimensioned array. (For details, see the appropriate statement listed alphabetically in Chapter 3.)

The maximum number of elements to which an array can be dimensioned is 3223. An array can be dimensioned with a literal or with a variable. An array that is dimensioned with a literal can be accessed more quickly than an array that is dimensioned with a variable. However, each element in an array with a literal dimension counts toward the total number of variables in a program. An array dimensioned with a variable counts as only a single variable.

Dynamic Arrays

A string that has elements delimited by system delimiters is called a dynamic array. A dynamic array does not have a fixed number of elements nor is it dimensioned. It is an array in that its component data elements can be referenced using subscripts. It is dynamic in that individual elements may be added, changed, or deleted within the string, causing the relative positions of the elements to be subject to change.

A dynamic array consists of one or more attributes; multiple attributes are delimited by attribute marks. An attribute mark has an ASCII equivalent of 254, shown as ^ by the editor and ~ by BASIC.

An attribute, in turn, may consist of one or more values; multiple values in an attribute are delimited by value marks. A value mark has an ASCII equivalent of 253, shown as] by the editor and } by BASIC.

Finally, a value may consist of one or more subvalues; multiple subvalues in a value are delimited by subvalue marks. A subvalue mark has an ASCII equivalent of 252, shown as \ by the editor and | by BASIC.

Note: *This manual displays the delimiters as shown by the editor.*

An example of a dynamic array is as follows:

```
Jones^Alice^244^temporary
```

Jones, Alice, 244, and temporary are attributes.

The following illustrates a more complex dynamic array:

```
Jones^Alice^2364 E. Main]Apt 206^English\s\r\w]
Spanish\s
```

Jones, Alice, 2364 E. Main]Apt 206, and English\s\r\w]Spanish\s are attributes. 2364 E. Main, Apt 206, English\s\r\w, and Spanish\s are values. English, s, r, w, Spanish, and s are subvalues.

Each element of the dynamic array can be addressed by specifying its position within angle brackets (< >); the first subscript specifies the attribute, the second subscript, if present, specifies the value within the selected attribute, and the third subscript, if present, specifies the subvalue within the selected value.

For example, if X represents the first example dynamic array above, then X<2> denotes attribute two of the string, which is "Alice". If Y represents the second dynamic array above, then Y<3,2> = "Apt 206" and Y<4,2,1> = "Spanish".

If the specified element is not in the array, a null value is returned; a missing dynamic array element is not considered an error. For example, if Y represents the second dynamic array, Y<2,2> = "".

Dynamic arrays are significant in Ultimate BASIC because items in files are in dynamic array format; thus, dynamic arrays may be used to represent data in disk files. Special constructs are available for manipulating dynamic arrays, thus making it easier to access and update files.

The maximum number of attributes in a dynamic array is 32,767.

The following BASIC functions and statements are used to reference dynamic arrays:

- EXTRACT
- DELETE
- INSERT
- LOCATE
- REMOVE
- REPLACE
- REUSE

For details, please refer to the appropriate function or statement, listed alphabetically in Chapter 3.

Arithmetic Expressions

Expressions are formed by combining operators with variables, constants, or BASIC functions. Arithmetic expressions are formed by using arithmetic operators.

When an expression is encountered as part of a BASIC program statement, it is evaluated by performing the operations specified by each of the operators on the adjacent operands.

The simplest arithmetic expression is a single unsigned numeric constant, variable, or intrinsic function. 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.

Order of Operations

When more than one operator appears in an expression, certain rules are followed to determine which operation is to be performed first. Each operator has a precedence rating. In any given expression the highest precedence operation is performed first.

The arithmetic operators have the following precedence:

Operator	Operation	Precedence
^	exponentiation	1
*	multiplication	2
/	division	2
+	addition or identity	3
–	subtraction or negation	3

If there are two or more operators with the same precedence, or an operator appears more than once, the leftmost operation is performed first.

For example, consider this expression: -50/5+3*2. The division and multiplication operators have the same precedence and it is higher than the precedence of the other operators. Since the division operator is leftmost, it is evaluated first: 50/5 = 10. The expression then becomes -(10)+3*2. The multiplication operation is performed next: 3*2 = 6. The expression then becomes: -(10)+(6). The negation is the leftmost

operator, so it is applied to the 10. The addition is then performed, yielding the final result, -4.

Any sub-expression may be enclosed in parentheses. The parenthesized sub-expression as a whole has highest precedence and is evaluated first. However, within the parentheses, the rules of precedence apply. For example, the following expression is evaluated as follows:

```
10+2*3-1 = 15
```

However, parentheses can change the order of operation:

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

Parentheses may be used anywhere to clarify the order of evaluation, even if they do not change the order. For example,

```
10+(2*3)-1 = 15
```

Arithmetic operators may not be adjacent to each other. For example, 2*-3 is not a valid expression, although 2*(-3) is.

Example	Description
`2+6+8/2+6`	evaluates to 18
`12/2*3`	evaluates to 18
`12/(2*3)`	evaluates to 2
`-5^2`	evaluates to -25
`(-5)^2`	evaluates to 25
`8*(-2)`	evaluates to -16
`5 * "3"`	evaluates to 15

Processing Numeric and String Data

In BASIC, data may be stored as a numeric value or a string value (which may or may not consist entirely of numbers). Arithmetic operations process these data types differently .

Internally in the Ultimate operating system, a numeric value is stored as a six-byte binary number, which is expressed in hexadecimal or converted to decimal. The maximum value possible is:

140737488355327 = X'7FFFFFFFFFFF'

Thus, when the PRECISION is set to 4 (the default), the maximum decimal value is 14,073,748,835.5327.

The PRECISION statement allows a program to preset the number of decimal places returned by standard arithmetic performed in that program; the range is 0 (only integer values returned) to 9 (returned values may have up to nine decimal places). Thus, a program's PRECISION affects the range of numeric values that are valid in that program. However, a PRECISION statement is ignored by explicitly coded string and floating point arithmetic operations, since these functions are designed to deal with larger (string) numbers, and by the functions EXP, LN, and PWR.

A string value is stored as a series of ASCII characters. String numbers may be of any length; hence, there is no limit on the magnitude or precision.

In BASIC, arithmetic may be performed via expressions that contain arithmetic operators and via certain functions, such as PWR. The arithmetic operators performs binary arithmetic, if possible, on numeric or string values by converting them to binary for the operation.

If the values exceed the range that binary arithmetic can handle within its six-byte maximum, string arithmetic is automatically invoked by the system, without programmer or user intervention. Both operations are considered "standard arithmetic". Again, if the result of an arithmetic operation is too large to be stored in a six-byte binary number, string arithmetic is automatically used by the system. In this case, the program's PRECISION is in effect .

If a string value containing only numeric characters is used in an arithmetic expression, it is considered as a decimal number. For example, 123 + "456" evaluates to 579.

If a string value containing non-numeric characters is used in an arithmetic expression, a warning message is printed when the program is executed and zero is assumed for the string value. For more information, see Appendix B, BASIC Run-Time Error Messages.

The following expression, for example, when executed, generates a warning message and evaluates to 123:

```
123 + "ABC"
```

Arithmetic Operators and Dynamic Arrays

The variables used in arithmetic expressions can contain dynamic arrays. The specified operation is automatically performed on corresponding array elements. If the arrays do not have the same number of elements, the system assumes a value of zero (0) for the missing elements for addition, subtraction, multiplication, and dividends in division. It assumes a value of one (1) for missing divisors in division.

Note: *The function REUSE allows you to use the previous value instead of zero when the number of elements differ. For more information on REUSE, see Chapter 3.*

```
ARRAY1 =  1:AM: 2:VM: 2:VM: 2:AM: 3
ARRAY2 = 10:AM:20:VM:20:VM:20:AM:30
ARRAY3 = ARRAY1 + ARRAY2
```

result:

```
ARRAY3 = 11:AM:22:VM:22:VM:22:AM:33
```

The elements of ARRAY3 are composed of the sums of the five elements in ARRAY1 and ARRAY2.

```
ARRAY1 = 1:VM:1:AM:2:VM:2:AM:3
ARRAY2 = 1:      AM:2       AM:3
ARRAY3 = ARRAY1 + ARRAY2
```

result:

```
ARRAY3 = 2:VM:1:AM:4:VM:2:AM:6
```

ARRAY3 is built as follows:

1. The first two values in attribute 1 are added.

2. ARRAY2 does not have a second value in attribute 1, so 0 is added to the second value in ARRAY1.

3. The first value in attribute 1 of ARRAY1 is added to the first value in attribute 2 of ARRAY2.

4. ARRAY2 does not have a second value in attribute 2, so 0 is added to the second value in ARRAY1.

5. The values in attribute 3 are added.

Rules for Standard Arithmetic

For each arithmetic operation, the system performs as follows:

1. The system first attempts to convert all values to binary numbers (if they are not already).

2. If all values can be converted to binary, binary arithmetic is performed. If the resulting binary number can be stored in six bytes, it is stored, using the program's PRECISION to truncate to the proper number of decimal places if needed. The operation is then considered complete.

   If the result would overflow a 6-byte binary storage area, the system automatically cancels the operation and prepares for automatic string math (see 3, below).

3. If all values cannot be converted to binary or the result would overflow, then the system attempts to convert the original values to string numbers.

4. If all values can be converted to strings, string arithmetic is performed. The resulting value is stored as a string, using the program's PRECISION to truncate to the proper number of decimal places if needed. The operation is then considered complete.

5. If the values cannot be converted to either binary or string numbers, then an error message is generated, and the operation is performed, with zero being used for the unconverted values. The result is stored as a string, and the PRECISION is applied.

Extended Arithmetic Functions

In addition to standard arithmetic, functions are available that can be used in expressions to perform mathematical operations. These functions allow the programmer to explicitly code string arithmetic or floating point arithmetic operations into a program. These functions are considered "extended arithmetic".

When a program requires calculations beyond the precision or magnitude of the standard arithmetic, either the string or floating point arithmetic may be used. It is usually best to select one of the two types and do all calculations in that mode. This minimizes confusion and also reduces the number of conversions which must be performed.

String arithmetic can handle virtually any operation and it requires the least conversion since all standard numbers are automatically string numbers as well. One might decide to always use string arithmetic except for speed considerations.

The speed of floating point operations and string operations are essentially the same except in multiplication. Floating point multiplication is considerably faster, depending on the number of digits involved. For example, it is four times faster to multiply 12345678909.87 by 1.00327 in floating point than in string and it is seven times faster to multiply two 13-digit numbers together in floating point.

For each string arithmetic operation:

1. A specific intrinsic function is used in an expression (SADD, SSUB, SMUL, SDIV).

2. The system attempts to convert all original values to string numbers (if they are not already).

3. If all values can be converted to strings, string arithmetic is performed. The resulting value is stored as a string. The program's PRECISION is ignored and the full resulting value is always stored. The operation is then considered complete.

4. If the values cannot be converted to string numbers, an error message is generated, and the operation is performed with zero being used for the unconverted values. The result is stored as a string.

For each floating point arithmetic operation:

1. The original values must have been converted to floating point values via the FFLT function.

2. A specific intrinsic function is used in an expression (FADD, FSUB, FMUL, FDIV).

3. Floating point arithmetic is performed. The resulting value is stored as a floating point number.

4. The resulting value may be used in other floating point functions or converted back to a string number by the FFIX function.

After any arithmetic operation, the resulting value has the same data type as the values used; that is, binary arithmetic produces numeric values, string arithmetic produces string values, and floating point arithmetic produces floating point string values.

Arithmetic Values and Comparison Statements

The results of an arithmetic operation may be used in a comparison statement in the BASIC program. Each comparison statement (IF, FCMP, and SCMP) follows certain processing rules in making the comparison.

Rules for IF Comparisons

1. The system first attempts to convert all values to binary numbers.

2. If all values can be converted to binary numbers, the binary values are compared as numeric entities, using the program's PRECISION to determine the proper number of decimal places. The result of comparison is either "true" (1) or "false" (0).

3. If all values cannot be converted to binary, the system attempts to convert the original values to string numbers.

4. If all values can be converted to numeric strings, a numeric comparison is made using string arithmetic. The program's PRECISION is not considered. The result of the comparison is either "true" or "false" and depends on the specific operators used in the expression.

5. If either operand is not numeric, then a pure string comparison is done (see Numeric vs. String Comparisons below).

Rules for SCMP (String) Comparisons

1. The system attempts to convert all values to string numbers. If they cannot be converted successfully, an error message is generated, and zero (0) is used for the value of each unconverted value.

2. The converted values are compared as ASCII numeric strings and the result specifies if they are equal in value or if the first is less than or greater than the second. The resulting value (0, -1, or 1) is returned.

Rules for FCMP (Floating Point) Comparisons:

1. The system attempts to compare both values as floating point string numbers. The result specifies if they are equal in value or if the first is less than or greater than the second. The resulting value (0, -1, or 1) is returned.

Numeric vs String Comparisons

The type of comparison used depends on whether the values are being compared as numbers (if binary or numeric strings) or ASCII characters (if any string operand is non-numeric).

The two methods are summarized as follows:

1. **Numeric comparison** - A numeric comparison is attempted first, and is made whenever both values can be converted to binary or numeric strings. If both values have an equivalent numeric value, then they are considered to be 'equal'. If they are unequal in value, the value with the larger numeric value is considered 'greater than' the other. If either or both values contains any non-numeric characters, the character pair comparison is made on the non-numeric character pairs (as in 2 below). For example:

   100 is equal to 0100
   A1 is greater than 99
   1A is less than 99

2. **String (ASCII) comparison** - Character pairs are compared one at a time from left to right. If no unequal character pairs are found, then the strings are considered to be 'equal'. If an unequal pair of characters is found, the characters are ranked according to their ASCII code numeric equivalent. The character with the higher numeric ASCII equivalent is considered to be greater than the other. If the two strings are not the same length, and the shorter string is otherwise identical to the beginning of the longer string, the longer string is considered greater than the shorter string. For example:

   WORDS is less than Words
   XXX is greater than XX

If the string has only numbers and includes a decimal point, the decimal point is used to determine the magnitude of the number.

For example:

'12345.1 ' is less than '123451'
'12345.1' is equal to '12345.10'

For a list of ASCII code equivalents, see Appendix D.

String Expressions

A string is a set of characters enclosed in single or double quotes or backslashes. A string expression may be any of the following:

- string constant
- variable with a string value
- a substring
- concatenation of string expressions

String expressions may be combined with arithmetic expressions. If numeric values are used in a string expression, the system converts them into equivalent string values before performing the operation.

Substrings

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

Substrings are defined by specifying the starting character position and the number of characters, separated by a comma and enclosed in brackets:

```
string[start.pos{,no.char}]
```

If the starting position specification is past the end of the string, an empty substring value is returned; for example, if A has a value of 'XYZ', A[4,1] has a value of "". If the starting position specification is less than one, one is used; for example, if X has a value of 'JOHN', X[-5,1] has a value of 'J'.

If the number of characters specification exceeds the remaining number of characters in the string, the remaining string is selected; for example, if B has a value of '123ABC', B[5,10] has a value of 'BC'. If the number of characters specification is less than one, an empty substring is returned; for example, B[1,-2] has a value of "". If number of characters is not specified, 1 is assumed.

Concatenation

Two strings are concatenated by appending the characters of the second string onto the end of the first. Concatenation is specified by a colon (:) or CAT operator. A space must precede and follow the CAT operator. Spaces are not required for the colon.

The following examples both return the same value:

```
'Good ' CAT 'Morning'
'Good ':'Morning'
```

The result is

```
'Good Morning'
```

Precedence

The precedence of the concatenation operator is lower than any of the arithmetic operators. So, if the concatenation operator appears in an expression with an arithmetic operator, the concatenation operation is performed last. Multiple concatenation operations are performed from left to right. Parenthesized sub-expressions are evaluated first.

The precedence of the substring operator is higher than that of the arithmetic operators. So in an expression such as A+B[7,3], the substring of B is extracted, converted to a numeric value, then added to the value of A.

In the following examples, assume

```
A = ABC123
Z = EXAMPLE
```

Expression	Description
`Z[1,4]`	Evaluates to "EXAM"
`A : Z[1,1]`	Evaluates to "ABC123E"
`Z[1,1] CAT A[4,3]`	Evaluates to "E123"
`3*3:3`	3*3 is evaluated first and results in the number 9. 9:3 is then evaluated and results in the string value 93.
`A[6,1] +5`	Evaluates to 8.
`Z CAT A : Z`	Evaluates to "EXAMPLEABC123EXAMPLE"
`Z CAT " ONE"`	Evaluates to "EXAMPLE ONE"

Format Strings: Numeric Mask and Format Mask Codes

Both numeric and non-numeric values may be formatted by the use of format strings. A format string immediately following a variable name or expression specifies that the value is to be formatted as specified by the characters within the format string. (You can also use the FMT function to format values; see the description of FMT listed alphabetically in Chapter 3.)

A format string may contain a numeric mask and/or a format mask.

Syntax

```
"{just}{num.mask}{(format.mask)}"
"D{d}"
```

just — justification; may be R for right justification or L for left justification; for input, may be V, which specifies exact match. Default justification is left.

num.mask — numeric mask, in the following format:

```
{n{m}}{$}{,}{N}{Z}{c}
```

- n — single numeric digit that specifies the number of digits to display following the decimal point; the displayed value is rounded, if necessary (the actual value is not altered). If n = 0, the decimal point is not output following the value.
- m — single numeric digit that specifies scaling factor; causes the converted number to be descaled (divided) by a factor equal to 10 raised to the power (m minus PRECISION value). For example, to descale a number by 10 if PRECISION is 4, m should be set to 5; to descale a number by 100 if PRECISION is 0, m should be set to 2. If m is used, n must precede it.
- $ — places a dollar sign immediately to the left of the value
- , — inserts commas between every thousands position of value
- N — causes the minus sign of negative values to be suppressed
- Z — specifies suppression of leading zeros; if value is zero, null is displayed

c credit indicators; may be any one of the following:

C causes the letters 'CR' to follow negative values and causes two blanks to follow positive or zero values

D causes the letters 'DB' to follow positive values; two blanks to follow negative or zero values

M causes a minus sign to follow negative values; a blank follows positive or zero values

E causes negative values to be enclosed in angle brackets (<...>); a blank precedes and follows positive or zero value

(format.mask) can be any of the following:

#n specifies that the data is to be placed in a field of n blanks

*n specifies that the data is to be placed in a field of n asterisks

%n specifies that the data is to be placed in a field of n zeros

x any other characters, including parentheses and dollar signs, are displayed exactly as specified. Each character adds one to the number of characters displayed in the result. See the examples.

D{d} format as date; d may be any one of the following:

n{s} number of digits to display for year; may be any value between 0 and 4; s is separator, may be any character to use as separator. If s is used, date is displayed in dd/mm/yy format; if s not used, date is displayed in dd mon yy format

D day of the month

J julian date

M month as numeric value

Q quarter

Y year

Description

The format string may be a literal or it may be assigned to a variable and immediately follows the expression it is to format. The entire format string is enclosed in single or double quotes or backslashes when it is used as a literal. If the format mask is used, it should be enclosed in parentheses. (Some format masks, such as #, function correctly without the parentheses, but it is recommended that all format masks be enclosed in parentheses.)

If a dollar sign is specified in a numeric mask, it is output just preceding the value. If a dollar sign is used within the format mask, it is output in the position indicated in the mask. See the mask examples.

The resulting formatted value may be used anywhere an expression is permitted, including in an assignment statement, which stores the formatted value, and in PRINT statements of the following form:

```
PRINT X "format string".
```

A format string can be used in an INPUT statement, in which case the input is verified according to the format specifications and redisplayed in formatted form. For information on INPUT verification, see the description of the INPUT statement in chapter 3.

Characters are placed in the format mask starting with the rightmost character if right justification is specified and starting with the left in all other cases. If the number of characters in the value is greater than the number of characters in the format mask, the extra characters are truncated.

Precedence

Formatting has higher precedence than concatenation, but lower than substring and arithmetic operations.

The following examples assume the PRECISION is 4.

Unconverted String	Format String	Result
X = 1000	V = X "R26"	10.00
X = 38.16	V = X "1"	38.2
X = 1234588	V = X "R27,"	1,234.59
X = -12345888	V = X "R27,E$"	$<1234.59>
X = -1234	V = X "R25$,M(*10)"	**$123.40-
X = -1234	V = X "R25,M($*10)"	$***123.40-
X = 072458699	V = X "L(###-##-####)"	072-45-8699
X = 072458699	V = X "L(#3-#2-#4)"	072-45-8699
X = Smith, John	V = X "L((#12))"	(Smith, John)
X = 12.25	Y = "1"; PRINT X Y	12.3
X = 12345	PRINT X "R2,"	12,345.00
X = 345	PRINT 12:X "R2,"	12345.00
X = 1	INPUT @(2,4):X "R(%%)"	01
X = Smith	PRINT X '(NAME: #10)'	NAME: Smith
A = ""	INPUT @(3,5):A "V(%%%)"	000 (will only accept three numeric digits as input)
A = 8100	PRINT A "D"	05 MAR 1990

Relational Expressions

Relational expressions are the result of applying a relational operator to a pair of arithmetic or string expressions.

The following relational operators are available:

Operator Symbol	Operation
=	Equal to
< LT	Less than
> GT	Greater than
<= =< LE	Less than or equal to
>= => GE	Greater than or equal to
# <> >< NE	Not equal to
MATCH MATCHES	Pattern matching

Precedence

Relational operators have lower precedence than all arithmetic and string operators; therefore, relational operators are only evaluated after all arithmetic and string operations have been evaluated.

Evaluation

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

For purposes of evaluation, relational expressions are divided into arithmetic and string relations. An arithmetic relation is a pair of arithmetic expressions separated by any one of the relational operators. A string relation is a pair of string expressions separated by any one of the relational operators. A string relation may also be a string expression and an arithmetic expression separated by a relational operator; if a relational operator encounters one numeric operand and one string operand, it treats both operands as strings.

If the two strings are not the same length, and the shorter string is otherwise identical to the beginning of the longer string, the longer string is considered "greater" than the shorter string.

Example	Description
`4 < 5`	Evaluates to 1 (true).
`"D" EQ "A"`	Evaluates to 0 (false).
`"D" > "A"`	ASCII equivalent of D (X'44') is greater than ASCII equivalent of A (X'41'); expression evaluates to 1.
`"Q" LT 5`	ASCII equivalent of Q (X'51') is not less than ASCII equivalent of 5 (X'35'); expression evaluates to 0.
`6+5 = 11`	Evaluates to 1.
`Q EQ 5`	Evaluates to 1, if current value of variable Q is 5; otherwise, evaluates to 0.
`"ABC" GE "ABB"`	Evaluates to 1 since C is greater than B
`"XXX" LE "XX"`	Evaluates to 0.

Pattern Matching

BASIC pattern matching allows the comparison of a string value to a predefined pattern. Pattern matching is specified by the MATCH or MATCHES relational operator, which compares the string value of the expression to the predefined pattern (which is also a string value) and causes the relation to evaluate to 1 (true) or 0 (false).

Syntax

expression MATCH{ES} {~}pattern

expression — any valid string expression

~ — indicates negation of pattern that follows; valid for nN and nA patterns only

pattern — may consist of any combination of the following:

- nN — tests for n numeric characters
- nA — tests for n alphabetic characters
- nX — tests for n characters of any type
- 'string' — tests for specified literal string of characters; if the literal pattern contains numeric characters, they must be enclosed within delimiters other than the delimiters enclosing the string

Description

The number of characters specified by n must match the number of characters in the string to be compared.

The ~ (tilde) negates the pattern match. The negation is true only if no characters in the expression match the type (N or A).

If the integer number used in the pattern is 0, the relation evaluates to 1 if all the characters in the string match the type, regardless of the number of characters in the string.

Example	Description
`A = 'ABC123'`	
`A MATCHES '3A3N'`	Evaluates to 1.
`A MATCHES '~3N~3A'`	Evaluates to 1.
`A MATCHES 'ABC"123"'`	Evaluates to 1.
`A MATCHES '~6N'`	Evaluates to 0 because there are not 6 non-numeric characters.
`Q MATCHES "0N"`	Evaluates to 1 if current value of Q is any unsigned integer; otherwise, evaluates to 0.
`B MATCH '3N"-"2N"-"4N'`	Evaluates to 1 if current value of B is, for example, any social security number; otherwise, evaluates to 0.
`A MATCHES "0N'.'0N"`	Evaluates to 1 if current value of A is any number containing a decimal point; otherwise, evaluates to 0.
`X MATCHES ""`	Evaluates to 1 if current value of X is the empty string; otherwise, evaluates to 0.

Logical Expressions

Logical expressions (also called Boolean expressions) are the result of applying logical (Boolean) operators to relational or arithmetic expressions.

The following logical operators are available:

Operator Symbol	Operation
AND &	Logical AND operation
OR !	Logical OR operation

Precedence

Logical operators operate on the true or false results of relational or arithmetic expressions. Logical operators have the lowest precedence and are only evaluated after all other operations have been evaluated. If two or more logical operators appear in an expression, the leftmost is performed first.

Evaluation

A OR B is true (evaluates to 1) if A is true or B is true; it is false (evaluates to 0) only when A and B are both false.

A AND B is true (evaluates to 1) only if both A and B are true; it is false (evaluates to 0) if A is false or B is false or both are false.

Example	Description
`1 AND A`	Evaluates to 1 if current value of variable A is non-zero; evaluates to 0 if current value of A is 0.
`8-2*4 OR Q5-3`	Evaluates to 1 if current value of Q5-3 is non-zero; evaluates to 0 if current value of Q5-3 is 0.
`A>5 OR A<0`	Evaluates to 1 if the current value of variable A is greater than 5 or is negative; otherwise, to 0.
`1 AND (0 OR 1)`	Evaluates to 1.
`J EQ 7 AND I EQ 5*2`	Evaluates to 1 if the current value of variable J is 7 and the current value of variable I is 10; otherwise, evaluates to 0.
`X1 AND X2 AND X3`	Evaluates to 1 if the current value of each variable (X1, X2, and X3) is non-zero; evaluates to 0 if the current value of any or all variables is 0.

Summary of Expression Evaluation

Expressions may consist of constants, variables, function references, and operators. Each operator has a precedence which determines the order in which operations within an expression are performed.

The operands of an expression may be constants, variables, function references, and other expressions enclosed in parentheses. All expressions, whether in parentheses or not, are evaluated according to the same rules of operator precedence:

- expressions in parentheses are evaluated before the results are used as operands in other expressions
- operators with higher precedence are processed first
- a series of operators with equal precedence is processed left to right.

Operators and their precedence are given in Table 2-1.

Table 2-1. BASIC Operators

Operator Symbol	Operation	Precedence
<...>	Dynamic array extraction	1 (high)
[...]	Substring specification	1 (high)
^	Exponentiation	2
*	Multiplication	3
/	Division	3
+	Addition or Identity	4
–	Subtraction or Negation	4
expression	Formatting	5
: or CAT	Concatenation	6
< or LT	Less than	7
> or GT	Greater than	7
<= or =< or LE	Less than or equal to	7
= or EQ	Equal to	7
# or <> or >< or NE	Not equal to	7
>= or => or GE	Greater than or equal to	7
MATCH or MATCHES	Pattern Matching	7
AND or &	Logical AND	8 (low)
OR or !	Logical OR	8 (low)

Figure 2-3 illustrates the way an expression with several levels of precedence is evaluated.

Expression to be evaluated:

```
A + B : C[D,(E^F*G)] H MATCH I AND J
```

Evaluation: (r1...r8 are results of prior operations):

```
1.  A + B : C[D,(E^F*G)] H MATCH I AND J
2.  A + B : C[D,((r1)*G)] H MATCH I AND J
3.  A + B : C[D,(r2)] H MATCH I AND J
4.  A + B : (r3) H MATCH I AND J
5.  (r4) : (r3) H MATCH I AND J
6.  (r4) : (r5) MATCH I AND J
7.  (r6) MATCH I AND J
8.  (r7) AND J
9.  (r8)
```

Figure 2-3. Precedence

Limited Expressions

Certain instructions cannot use expressions that contain operators with a precedence level 5 or above. Expressions of this type are considered to be "limited expressions" since they may specify only the following operations:

<...>	Dynamic array extraction
[...]	Substring specification
^	Exponentiation
*	Multiplication
/	Division
+	Addition
–	Subtraction

For identification in the documentation, limited expressions are referred to as X4-expr in the statements that use them. The following statements contain one or more parameters which may be of the X4-expr type only:

Assignment (=) statement
DELETE statement
EXTRACT statement
INPUT statement
INS statement
INSERT statement
LOCATE statement
MATREAD{U} statement
MATWRITE statement
PRINT ON statement
REPLACE statement
REUSE statement

For details, please see the appropriate function or statement, listed alphabetically in Chapter 3.

Note: *Whenever an expression with a level 5-8 operator is needed for an X4-expr parameter, the complex expression may be enclosed in parentheses and is then considered valid. For example,*

```
INS (A:B) BEFORE ...
```

Variable Data Area

The variable data area used by a BASIC program is composed of a descriptor table, free storage area, and a buffer size table.

Descriptor Table Structure

The type of information in each variable in a program is kept in the descriptor table. The descriptor table contains one 10-byte entry for each variable (including array elements) in the program. The number of descriptors, and hence, the number of variables in a program, is limited to 3223.

A descriptor contains a code byte which identifies the type of the descriptor as one of the following:

Content of Descriptor	Usage
6-byte binary number	for numeric values
8-byte string plus a terminating segment mark	for string values of eight characters or less
6-byte pointer to a string in the free storage area	for string values with more than eight characters
8-byte reference to file access information (base, modulo, separation)	for file variables
6-byte pointer to external subroutine code	for external subroutines
2-byte mode-id	for assembly code routines

Free Storage Area

The free storage area is made up of buffers of various sizes. One of these buffers is assigned to a variable if the string to be stored in the variable cannot fit in its descriptor (more than eight characters). A pointer to this area is stored in the descriptor.

Buffer Table

Strings longer than eight bytes are placed in storage buffers located in the free storage space. These buffers are by default 50 bytes, 150 bytes, or multiples of 250 bytes in length. There is overhead involved; the BASIC run-time package reserves 7 bytes per buffer for internal usage. The maximum length for strings in 50-byte buffers, then, is 43 bytes.

A program can change the default buffer sizes of 50 bytes, 150 bytes, and multiples of 250 bytes, by executing a STORAGE statement. (Please refer to the STORAGE statement, listed alphabetically in Chapter 3.)

When a string requires a new buffer, the system looks in the table of abandoned buffers for a buffer of the appropriate size. If one cannot be found, a buffer that is somewhat larger than the string it will contain is allocated from free storage. This allows the string to grow.

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, after garbage collection takes place, there is still not enough contiguous space (which should happen very rarely), the program is aborted with the message:

```
NOT ENOUGH WORK SPACE
```

At this point, the programmer can attempt to reduce the number of characters in variables, and set to null all variables that are no longer needed. For example, if a very large item has been retrieved and only one attribute from it is required, the attribute can be extracted and the variable for the item set to null.

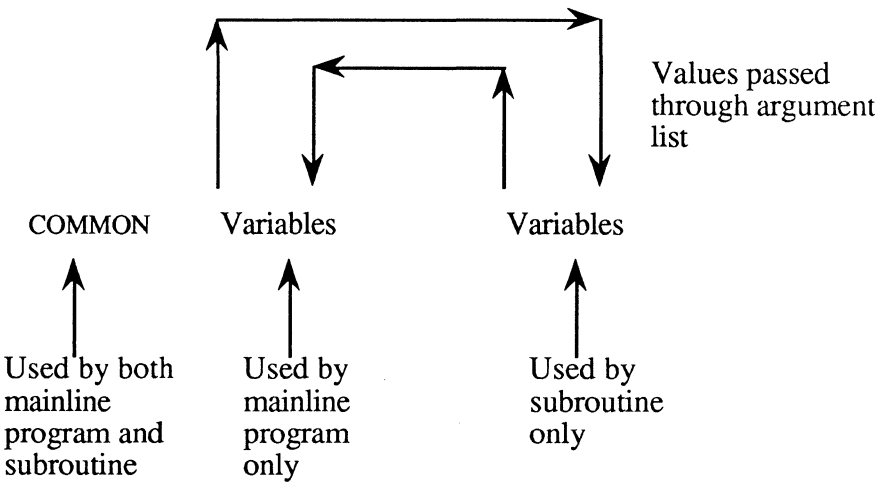
Variable Allocation

Variables are allocated descriptors in the order in which they are declared in a program. For this reason, it is important that COMMON variables be declared before any other processing takes place. To ensure that variables in the main program and its subroutines match, it is recommended that the COMMON statements be placed in a separate program that is $INCLUDEd by all programs using that COMMON.

Note: *For details on COMMON and $INCLUDE, refer to Chapter 3, Statements and Functions.*

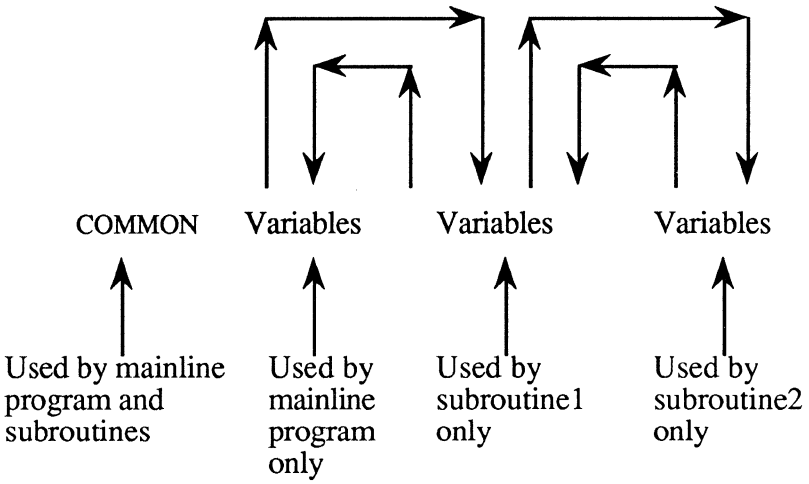
Program Descriptors

The arrangement of descriptors for a main program and an external subroutine is illustrated as follows:



The programs should be written so that variables declared as COMMON in both the main program and the subroutine are first. The COMMON variables then refer to the same locations and there is a one-to-one correspondence between the variables in both COMMON statements. However, when values are passed through the argument list on the CALL and SUBROUTINE statements, the values are copied back and forth between the two local areas as indicated above.

If subroutine calls are nested, the arrangement of descriptors is:



Values passed through the argument list are copied as indicated above.

Each of these statements are described in detail in Chapter 3, Statements and Functions.

CHAIN and ENTER

The ENTER statement may be used to transfer control to a new BASIC program which inherits the values of variables from the old program. The CHAIN statement may be used in a similar way when invoking the RUN verb with the I option to run a new program without initializing variables. (For details, please refer to the CHAIN and ENTER statements, listed alphabetically in Chapter 3.)

It is permissible to CHAIN or ENTER a program that calls a subroutine, but it is illegal to CHAIN or ENTER from a subroutine.

3 BASIC Statements and Functions

A BASIC statement performs a complete operation. Statements may appear anywhere in a program. All statements must be formatted with a space separating the statement name from any parameters that follow; for example:

```
GOTO 10
```

A BASIC function performs a function within a statement operation. Functions may appear anywhere that expressions can be used in a statement. All functions must be formatted with a left parenthesis following the function name, any parameters, and a right parenthesis; for example:

```
ALPHA(N)
COL1()
```

Organization of Chapter

Each statement and function is described in detail in its own separate topic. The topics are presented in alphabetical order by the statement or function name.

Each topic about a statement or function begins on a new page. Topics may be presented on one or more pages, as necessary. The statements and functions identified by symbols, such as the = (assignment) statement and the @ (cursor control) function, are listed before the statements and functions with alphabetical names:

! and * statement
$compiler directives
= (assignment) statement
@ function
ABORT statement
ABS function
.
.
WRITEV statement

For a description of the structure and components of a BASIC program, see Chapter 1. Chapter 1 also describes how programs are written, compiled, and executed.

A Summary of the Statements and Functions

Figure 3-1 lists the BASIC statements. Figure 3-2 lists the BASIC intrinsic functions. Figure 3-3 lists the BASIC compiler directives.

!	GOSUB	PROMPT
*	GOTO (GO TO)	PUT
=	HEADING	READ
ABORT	IF	READNEXT
BEGIN CASE	INPUT	READT
BREAK KEY	INPUTCLEAR	READU
CALL	INPUTCONTROL	READV
CASE	INS	READVU
CHAIN	LET	RELEASE
CLEAR	LOCATE	REM
CLEARDATA	LOCK	REMOVE
CLEARFILE	LOOP	REPEAT
CLEARSELECT	MAT =	RETURN (TO)
CLOSE	MATPARSE	REWIND
COMMON	MATREAD	RQM
CONVERT	MATREADU	SEEK
CRT	MATWRITE	SELECT
DATA	MATWRITEU	SLEEP
DEL	NEXT	STOP
DELETE	NULL	STORAGE
DIM	OFF	SUBROUTINE
DISPLAY	ON GOSUB	TRAP ON THEN
ECHO	ON GOTO	CALL
END	OPEN	UNLOCK
END CASE	PAGE	UNTIL
ENTER	PRECISION	WEOF
EQUATE	PRINT	WHILE
EXECUTE	PRINTER	WRITE
EXIT	PRINTERR	WRITET
FOOTING	PROCREAD	WRITEU
FOR	PROCWRITE	WRITEV
GET	PROGRAM	WRITEVU

Figure 3-1. BASIC Statements

@	FFLT	SADD
ABS	FIELD	SCMP
ALPHA	FMT	SDIV
ASCII	FMUL	SEQ
CHAR	FSUB	SIN
COL1	ICONV	SMUL
COL2	INDEX	SORT
COS	INMAT	SOUNDEX
COUNT	INSERT	SPACE
DATE	INT	SQRT
DCOUNT	LEN	SSUB
DELETE	LN	STR
EBCDIC	MATCHFIELD	SUM
EOF	MOD	SYSTEM
ERROR	NOT	TAN
EXP	NUM	TIME
EXTRACT	OCONV	TIMEDATE
ERRTEXT	PWR	TRIM
FADD	REM	TRIMB
FCMP	REPLACE	TRIMF
FDIV	REUSE	USERTEXT
FFIX	RND	

Figure 3-2. BASIC Functions

$*	$INCLUDE	INCLUDE
$CHAIN	$INSERT	
$COMPATIBILITY	$NODEBUG	

Figure 3-3. BASIC Compiler Directives

! and * Statements

The "!" and "*" statements are alternative forms of the REM (remark) statement. Remarks can identify a function or section of program code, as well as explain its purpose and method.

Syntax

! text ...
* text ...

text any arbitrary characters, up to the end of the line.

Description

A remark statement can be specified in one of three ways: by the REM statement, by an asterisk (*), or by an exclamation point (!).

REM, !, or * must be placed at the beginning of the statement, but may appear after another statement on the same line; a semicolon must be used to separate a remark statement from any other BASIC statement on the same line. A remark statement does not affect program execution.

Comments can be included in lines that end in a comma and continue on to the next line in the following statements:

CALL
COMMON
DIM{ENSION}
EQUATE

A semicolon must follow the comma and the comments **must** start with an asterisk (*):

```
010 COMMON FIRST,     ; *This is the first comment
011        CTR,       ; *This is another comment
012        LAST       ; *This is the last comment
```

Remarks are useful to summarize, introduce, explain, or document the program instructions and routines.

```
REM PROGRAM TO PRINT THE
*   NUMBERS FROM ONE TO TEN

     I = 1                    ;* START WITH ONE
BEG: PRINT I                  ;* PRINT THE VALUE
     IF I = 10 THEN STOP      ;* STOP IF DONE
     I = I + 1                ;* INCREMENT I
     GOTO BEG                 ;* BEGIN LOOP AGAIN
     END
```

$* Directive

The $* directive is used to embed comments (such as a copyright notice) in a program's object code.

Syntax

$* text

text comments to be included with the object code

Description

Any text that is specified after the asterisk (*) is assumed to be comments. The text appears in the object code in a code sequence not generated by any BASIC statement.

`*Program copyrighted`	Comments in source code
`$*Program copyrighted`	Comments in object code

$CHAIN Directive

The $CHAIN directive can be used to link program items together at compilation.

Syntax

$CHAIN {file.name} prog.name

file.name — name of file that contains program; if omitted, the file is assumed to be the one containing the program currently being compiled

prog.name — name of program to link together with current program

Description

The $CHAIN directive continues compilation with the specified program. Any source code in the current program appearing after the $CHAIN directive is ignored; therefore, the directive should be the last line in the source code.

If the final object code size cannot exceed 57,534 bytes.

```
001 * Long Program
  .
  .
999 $CHAIN MOD2          Continue compilation with next
                         program module
```

$COMPATIBILITY Directive

The $COMPATIBILITY directive is used to specify the compiler implementation to be used when compiling a BASIC program. This directive alters certain instructions to work according to a standard other than the Ultimate standard if there is a conflict.

Syntax

$COMPATIBILITY imp

imp implementation standards to use when compiling the program; currently, only R83PC, which generates instructions according to PICK R83 PC standards, can be specified

Description

The $COMPATIBILITY R83PC directive can be used in cases where an Ultimate release is not compatible with the PICK R83 PC standard. Currently, $COMPATIBILITY R83PC affects the use of data stacks and select lists used by EXECUTE statement.

If $COMPATIBILITY R83PC is in effect and there is an active data stack, the data stack is passed to the next EXECUTE statement with no IN. or STACKING clause. The data stack is also cleared on return from the EXECUTE statement. If $COMPATIBILITY R83PC is not in effect, the data stack is not passed, nor is it cleared.

Also, if $COMPATIBILITY R83PC is in effect and there is an active select list, the select list is passed to the next EXECUTE statement. If $COMPATIBILITY R83PC is in not effect, a select list is not passed unless the SELECT.< or RTNLIST parameter is specified.

Program with $COMPATIBILITY

```
$COMPATIBILITY R83PC
DATA 1 2 3
EXECUTE 'RUN BP NEXT'
INPUT A
```

Result:

The data stack is passed to the program NEXT and is cleared when control returns to this program. The INPUT statement requires an operator response.

Program without $COMPATIBILITY

```
DATA 1 2 3
EXECUTE 'RUN BP NEXT'
INPUT A
```

Result:

The data stack is not passed to the program NEXT. The INPUT statement takes the first value from the data stack.

Program with $COMPATIBILITY

```
$COMPATIBILITY R83PC
EXECUTE 'SSELECT MD = "UPD]"'
EXECUTE 'LIST ONLY MD'
```

Result:

```
MD...........
UPD-DEF
UPD-VALIDATE
UPD.LANGUAGES
  .
  .
7 items listed.
```

Program without $COMPATIBILITY

```
EXECUTE 'SSELECT MD = "UPD]"'
EXECUTE 'LIST ONLY MD'
```

Result:

```
MD...........
U/MAX
T-STATUS
U/HIGHAMC
  .
  .
355 items listed.
```

$INCLUDE Directive

The $INCLUDE directive may be used to include source code stored in one program item as part of another. $INSERT and INCLUDE may be used in place of $INCLUDE.

Syntax

$INCLUDE {file.name} prog.name
$INSERT {file.name} prog.name
INCLUDE {file.name} prog.name

file.name — name of file that contains program; if omitted, the file is assumed to be the one containing the program currently being compiled

prog.name — name of program to include in compilation of current program

Description

$INCLUDE directives may be nested up to three levels deep. Users should note that the object code of any BASIC program or external subroutine, whether or not it contains $INCLUDE directives, cannot exceed 57,534 bytes in size.

A typical use for the $INCLUDE directive is with a set of related BASIC programs using variables in COMMON. The COMMON statements can be placed in a single item which is included in each program by the $INCLUDE directive. This has the advantages of saving space, making changes easier, and reducing the chance of declarations in one program mismatching those in another.

```
** Start program
$INCLUDE COM.CODES          Include program that defines
  .                         COMMON variables
  .
```

$NODEBUG Directive

The $NODEBUG directive causes the compiler to not save the end-of-line (EOL) opcodes and the symbol table as part of the object code.

Syntax

$NODEBUG

Description

The $NODEBUG directive has the same effect as specifying the C (suppress EOL opcodes) and S (suppress generation of symbol table) options with the COMPILE or BASIC verb.

The $NODEBUG directive should be used only after a program has been debugged, because when it is specified, all runtime errors are reported as occurring on line 1 and the BASIC debugger cannot display variables and other symbols.

= (Assignment) Statement

The = (assignment) statement is used to assign a value to a simple variable, a dimensioned array element, an element of a dynamic array, or a substring. In addition, the assignment statement may be used to add, subtract, or concatenate an expression to a simple variable.

Syntax

```
variable = expression
variable += expression
variable –= expression
variable := expression
variable(row{,col}) = expression
variable <attrib.no{,val.no{,subval.no}}> = expression
variable[start.char,no.char] = expression                    (overlay)
variable[delimiter,start.sub,no.subs] = expression           (replace)
```

variable	name of element to receive assignment
expression	any valid BASIC expression
+=	plus-equals; adds an expression to a variable and returns the results to the variable; this is equivalent to var = var + expression
–=	minus-equals; subtracts an expression from a variable and returns the results to the variable; this is equivalent to var = var - expression
:=	concatenate-equals; concatenates an expression with a variable and returns the results to the variable; this is equivalent to var = var : expression
row	row parameter for dimensioned array element
col	column parameter for dimensioned array element
attrib.no	attribute number of dynamic array element; if attrib.no has a value of -1, the expression is inserted after the last attribute, or if last attribute is null, replaces last attribute

val.no	value number of dynamic array element; if val.no has a value of -1, the expression is inserted after the last value in the attribute specified by attrib.no, or if last value is null, replaces last value
subval.no	subvalue number of dynamic array element; if subval.no has a value of -1, the expression is inserted after the last subvalue in the value specified by val.no, or if last subvalue is null, replaces last subvalue
start.char	starting character position in the variable; if start.char evaluates to 0 or less, 1 is used as the value. If start.char evaluates to greater than the number of characters in the string, no characters are overlaid. (For a complete description of overlaying substrings, see the section Overlaying a Substring, which starts on page 3-16.)
no.chars	number of characters to be overlaid; if no.chars evaluates to 0 or less, no characters are overlaid
delimiter	substring delimiting character; if the delimiter evaluates to more than one character, only the first character is used as the delimiter; if the delimiter evaluates to a null, no characters are replaced
start.sub	first substring to be changed; if start.sub is 0 or less, 1 is used as the value. If start.sub is greater than the number of delimited substrings in the original string, the required number of null delimited substrings are appended to the string. (For a complete description of changing substrings, see the section Replacing Delimited Substrings, which starts on page 3-17.)
no.subs	number of substrings to be changed; the actual change is determined as follows: • if no.subs is greater than 0, this number of delimited substrings is replaced • if no.subs is 0, expression is inserted at the location specified by start.sub

- if no.subs is less than 0, then starting at the substring specified by start.sub, the absolute value of no.subs substrings are deleted from the existing string, then expression is inserted

All parameters can be literals or expressions.

Description

The value of the expression becomes the current value of the variable on the left side of the equality sign. The expression may be any legal BASIC expression.

The value of the variable does not change until the entire right side of the statement has been evaluated.

The LET statement may optionally be prefixed to an assignment statement, as in LET X = 12.

An equated symbol may not be used in place of a variable in an assignment statement if the symbol has already been assigned a constant (literal) value in the program. For more information, please see the EQUATE statement listed alphabetically in this chapter.

The elements in a dimensioned array can be assigned values by the MAT = assignment statement. For more information, please see the MAT = statement listed alphabetically in this chapter.

`X=5`	Assigns 5 to X.
`X +=1`	Increments X by 1.
`ST="STRING"`	Assigns the character string to ST.
`ST1=ST[3,1]`	If ST = "STRING", assigns substring "R" to ST1.
`TABLE(I,J)=A(3)`	Assigns element from array TABLE to element.in array A.
`A = (B = 0)`	Assigns 1 to A if "B=0" is true, assigns 0 to A if "B=0" is false.
`A<2>=0`	Assigns 0 to attribute 2 of dynamic array A
`EXECUTE 'SELECT F1', RTNLIST A` `B=A`	Copies select-list from A to B.

Overlaying a Substring

A substring can be overlaid by a string by using an assignment statement:

variable[start.char,no.chars] = expression

This form of the assignment statement does not change the length of the string variable.

If the number of characters in the replacement expression is less than the number of characters specified in no.char, blanks are added to the end of expression. If the number of characters in the replacement expression is greater than no.char, the excess characters in expression are not assigned.

If no.chars is greater than the number of characters remaining in the original string, only the number of characters remaining are overlaid. Any extra characters are ignored.

```
A = 'ABCDEFGHI'
A[4,3] = '***'
```

result:

```
A = 'ABC***GHI'
```

The substring starting at the fourth character position and containing three characters (DEF) is replaced by the specified three characters (***).

```
M = 'ABCDEFGHI'
M[2,10]  = 'XXXXX'
```

result:

```
M = 'AXXXXX   '
```

The substring 'BCDEFGHI', which starts at the second character and extends to the end of the string (since there are fewer than ten characters left in the string), is replaced by the specified 5-character substring plus 3 spaces.

Replacing Delimited Substrings

One or more delimited substrings can be added, deleted, or replaced in a string. This form of the assignment statement can change the length of the string.

variable[delimiter,start.sub,no.subs] = expression

The expression to be inserted in place of the delimited substrings is assumed to contain substrings delimited by the same value as the original string. The first substring has no initial delimiter; the last substring has no final delimiter.

If the delimiter is a system delimiter (attribute mark, value mark, or sub-value mark), the substring is terminated by a second delimiter of the same level and ignores any higher level delimiter. For example, if sub-value mark is the delimiter, the substring does not stop at a value mark or attribute mark, only at the next sub-value mark.

If the specified delimiter is null, no characters are replaced.

If no.subs is greater than the number of substrings in expression, the required number of delimited null substrings are added to expression.

If no.subs is non-zero and is less than the number of substrings in expression, the extra substrings in expression are ignored. If no.subs is zero, the entire expression is inserted as a delimited substring preceding the substring specified by start.sub.

If no.subs is greater than the number of substrings remaining in the original string, the required number of delimited null substrings are appended to the original string.

If no.subs is less than 0, then starting at the substring specified by start.char, no.subs substrings are deleted from the existing string, then the expression is inserted at that location.

```
A = '1*2*3*4*5'
A['*',2,3] = 'A*B*C*D'
```

result:

```
A = '1*A*B*C*5'
```

The substring delimiter is an asterisk (*); the substring to replace starts at the second delimited substring and contains three substrings (2*3*4); it is replaced by the specified number of substrings and their delimiters (A*B*C).

```
A = '1':VM:'2':AM:'3':VM:'4':VM:'5'
A[VM,2,2] = 'A':VM:'B'
```

result:

```
A = '1':VM:'A':VM:'B':VM:'5'
```

The substring delimiter is a value mark; the substring to replace starts at the second delimited substring and contains two substrings ('2':AM:'3':VM:'4':); it is replaced by the specified two substrings and their delimiter ('A':VM:'B'). The attribute mark that separates the value marks is ignored by this form of the assignment statement.

```
A = '1*2*3*4'
A['*',-3,5] = 'A*B'
```

result:

```
A = 'A*B***'
```

The assignment starts at the first substring (-3 defaults to 1). String A contains fewer than five substrings, the number of substrings specified, so one null substring is appended to A. The replacement expression also contains fewer than five substrings, so three substrings are appended to it. Finally, the expression overlays the specified substrings in A.

```
A = '1*2*3*4'
A['*',6,2] = 'A*B'
```

result:

```
A = '1*2*3*4**A*B'
```

The assignment starts at the sixth substring. However, string A contains only four substrings, so two null substrings are appended to it. The expression is then appended to A, starting at the sixth substring.

```
A = '1*2*3*4'
A['*',3,0] = 'A*B'
```

result:

```
A = '1*2*A*B*3*4'
```

Since the number of substrings is zero, the two substrings in expression are inserted at the third substring position and no substrings are deleted.

```
A = '1*2*3*4'
A['*',3,-2] = 'Z'
```

result:

```
A = '1*2*Z'
```

Two delimited substrings are deleted starting at the third delimited substring and the new substring is inserted.

@ Function

The @ ("at" sign) function generates a string of control characters used for cursor positioning or other terminal or printer control features. The terminal or printer is affected when the string is later output to it with a CRT, DISPLAY, or PRINT statement.

Syntax

@(col{,row})

col — column to which the cursor is to be positioned; if it is negative, the @ function returns a terminal control string as described in Table 3-1; if the value of the @ function is less than -100, it affects Ultimate-supported letter-quality printers as shown in Table 3-2.

row — row to which cursor is to be positioned; if not specified, and col is positive, the cursor is assumed to remain on the current line. However, if the terminal on which the statement is executed does not support column-only cursor positioning, the results are unpredictable.

Description

In general, the @ function, other than @(-100) and lower, is not meant for statements that are to be directed to the printer and may cause unexpected results.

Columns and rows are numbered starting with zero (0), left to right and top to bottom on the screen. When positioning the cursor, the values of expressions used in the @ function should be within the column and row limits of the screen; otherwise, the results are unpredictable.

The @ function generates values based on the current terminal or printer type for the port (line) on which the BASIC program is run. The terminal type is determined by the most recent TERM command executed for the port, or by a terminal type logon parameter set up with the TERMINAL command, or by the system's default terminal type, which may be changed with the SET-TERM command. These commands set up the terminal using parameters in the TERMDEF item for the specified terminal type. The printer type is shown and changed with the PRINTER

command. For more information on these commands, please refer to the *Ultimate System Commands Reference Guide.*

Not all terminals or printers attached to terminal auxiliary ports respond to all control codes listed here. The documentation for each terminal or printer must be consulted for information about which features are supported. If a non-supported feature is used, a null string is returned. If a non-supported terminal is used, all cursor control characters return a CR/LF.

```
X = 7
Y = 3
PRINT @(X,Y): Z
```

Prints the current value of variable Z at column position 7 of row 3.

```
Q = @(3): "HI"
PRINT Q
```

Prints "HI" at column position 3 of current row.

```
A = 5
PRINT @(A,A+5):A
```

Prints the value 5 at column position 5 of row 10.

```
PRINT @(-1)
```

Clears the screen and positions the cursor at 'home' position.

```
F = @(-46)
```

Returns default values of function keys.

```
CONVERT CHAR(251):CHAR(250) TO CHAR(254):CHAR(253) IN F
```

Puts values into dynamic array format

```
PRINT F<1>:F<2>:F<3,13>:'OFF':CHAR(13):F<5>
```

Sets function key 13 to log user off when pressed.

```
NW.CORNER = @(-49)[1,1]
NE.CORNER = @(-49)[3,1]
HORIZONTAL = STR(@(-49)[2,1],10)
TOP = @(-50):NW.CORNER:HORIZONTAL:NE.CORNER:@(-51)
PRINT @(3,3):TOP
```

Upper left corner is first position in string; upper right corner is third.

Prints the top of a box with a corner at each end.

Table 3-1. Cursor Control Values (1 of 8)

Code	Description
@(-1)	Clear the screen and positions the cursor at 'home' (upper left corner of the screen).
@(-2)	Position the cursor at 'home' (upper left corner).
@(-3)	Clear from cursor position to the end of the screen.
@(-4)	Clear from cursor position to the end of the line.
@(-5)	Start blink.
@(-6)	Stop blink.
@(-7)	Start protected field.
@(-8)	Stop protected field.
@(-9)	Backspace the cursor one character.
@(-10)	Move the cursor up one line.
@(-11)	Move the cursor down one line.
@(-12)	Move the cursor right one column.
@(-13)	Enable auxiliary (slave) port.
@(-14)	Disable auxiliary (slave) port.
@(-15)	Enable auxiliary (slave) port in transparent mode.
@(-16)	Initiate slave local print.
@(-17)	Start underline.
@(-18)	Stop underline.
@(-19)	Start reverse video.
@(-20)	Stop reverse video.
@(-21)	Delete line.
@(-22)	Insert line.

Table 3-1. Cursor Control Values (2 of 8)

Code	Description
@(-23)	Scroll screen display up one line.
@(-24)	Start boldface type.
@(-25)	Stop boldface type.
@(-26)	Delete one character.
@(-27)	Insert one blank character.
@(-28)	Start insert character mode.
@(-29)	Stop insert character mode.
@(-30,c)	Set foreground and background color: **c** **background** **foreground** 1 black cyan 2 black red 3 black blue 4 black green 5 black magenta 6 black yellow 7 black white 8 blue red 9 blue green 10 blue white 11 blue yellow 12 blue red 13 blue cyan 14 blue magenta 15 white red 16 white green 17 white blue 18 white cyan 19 white magenta 20 white black 21 red white 22 red green

Table 3-1. Cursor Control Values (3 of 8)

Code	Description
@(-31,f)	Set foreground color: **f** **foreground** 1 brown (may vary on some terminals) 2 white 3 red 4 magenta 5 yellow 6 green 7 cyan 8 blue
@(-32,b)	Set background color: **b** **background** 1 brown 2 white 3 black 4 red 5 blue 6 cyan 7 magenta
@(-33)	Set 80 columns.
@(-34)	Set 132 columns.
@(-35)	Set 24 rows.
@(-36)	Set 44 rows.
@(-37)– @(-45)	Reserved

Table 3-1. Cursor Control Values (4 of 8)

Code	Description
@(-46)	Returns function key default values as a string in the following format: sFBfFBx1FA...xnFBy1FA...ynFBeFB s character sequence needed to set the overall characteristics of the function line; typically, this is null FB CHAR(252)[1] f lead-in sequence used to load function keys xn value for function key n FA CHAR(251) yn value for shifted function key n e terminator for key text
@(-47)	Returns character sequence needed to set the overall characteristics for the label line (bottom line of terminal). The following information is returned: sFBfFBxFByFBeFBr s character sequence needed to set the overall characteristics of the label line FB CHAR(252) f lead-in sequence used for label line x lead-in sequence for unshifted label line y lead-in sequence for shifted label line e terminator for text r reset label line (turn off)

[1] After the string is returned, the CONVERT function can be used to change the delimiters to attribute marks (CHAR 254) and value marks (CHAR 253) if desired. (Doing this converts the string to a dynamic array.)

Table 3-1. Cursor Control Values (5 of 8)

Code	Description
@(-48)	Returns character sequence needed to set the overall characteristics for the status line (top line of terminal). The following information is returned: sFBfFBxFByFBeFBr s character sequence needed to set the overall characteristics of the status line FB CHAR(252) f lead-in sequence used for status line x lead-in sequence for unshifted status line y lead-in sequence for shifted status line e terminator for text r reset status line (turn off)
@(-49)	Returns string that defines the graphics characters codes for the current terminal; the exact characters that will be displayed depend on the terminal type. Before the code is printed, the terminal's graphic capability must be turned on by an @(-50) statement. After the graphics have been printed, the graphic capability must be turned off by an @(-51) statement. The codes in @(-49) are single digits whose meanings are determined by the position of the code in the string. The first eleven positions in the string define the following single line graphics characters:

Table 3-1. Cursor Control Values (6 of 8)

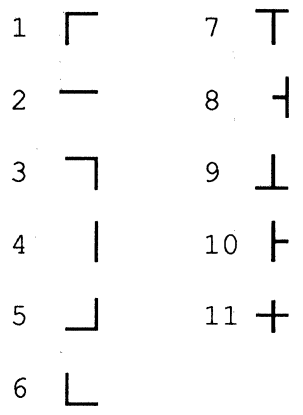
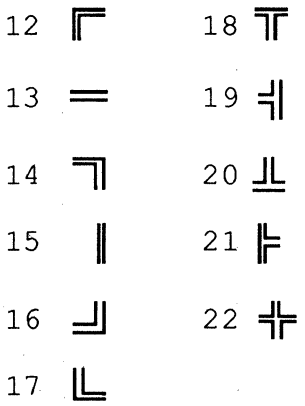
Code	Description
	 The second set of eleven positions define the following double line graphics characters:  The 23rd through 26th positions define other graphic characters, depending on the terminal type.
@(-50)	Start graphics.
@(-51)	Stop graphics.
@(-52)	Start blink.
@(-53)	Stop blink.
@(-54)	Start reverse video.

Table 3-1. Cursor Control Values (7 of 8)

Code	Description
@(-55)	Stop reverse video.
@(-56)	Start reverse video and blink.
@(-57)	Stop reverse video and blink.
@(-58)	Start underline.
@(-59)	Stop underline.
@(-60)	Start underline and blink.
@(-61)	Stop underline and blink.
@(-62)	Start underline and reverse video.
@(-63)	Stop underline and reverse video.
@(-64)	Start underline, reverse video, and blink.
@(-65)	Stop underline, reverse video, and blink.
@(-66)	Start dim.
@(-67)	Stop dim.
@(-68)	Start dim and blink.
@(-69)	Stop dim and blink.
@(-70)	Start dim and reverse video.
@(-71)	Stop dim and reverse video.
@(-72)	Start dim, reverse video, and blink.
@(-73)	Stop dim, reverse video, and blink.
@(-74)	Start dim and underline.
@(-75)	Stop dim and underline.
@(-76)	Start dim, underline, and blink.

Table 3-1. Cursor Control Values (8 of 8)

Code	Description
@(-77)	Stop dim, underline, and blink.
@(-78)	Start dim, reverse video, and underline.
@(-79)	Stop dim, reverse video, and underline.
@(-80)	Set 80 columns
@(-81)	Reserved
@(-82)	Set 132 columns

Table 3-2. Letter-Quality Printer Control Values

Code	Description
@(-101,p)	Set VMI (Vertical Motion Index) to p.
@(-102,h)	Set HMI (Horizontal Motion Index) to h.
@(-103)	Set alternate font.
@(-104)	Set standard font.
@(-105)	Generate a half line-feed.
@(-106)	Generate a negative half line-feed.
@(-107)	Generate a negative line-feed.
@(-108)	Print black ink.
@(-109)	Print red ink.
@(-110)	Load cut sheet feeder.
@(-111)	Select feeder1.
@(-112)	Select feeder2.
@(-113)	Select standard thimble.
@(-114)	Select proportional space thimble.
@(-115)	Start automatic boldfacing.
@(-116)	Stop automatic boldfacing.
@(-117)	Start automatic underlining.
@(-118)	Stop automatic underlining.

ABORT Statement

The ABORT statement terminates program execution. If the program was run from a PROC, the PROC is terminated as well.

Syntax

ABORT {errnum{,param, param, ...}}

errnum error message number (item.id) in the ERRMSG file

param parameters to be used within the error message format; must be separated by commas; may be variables or literals

Description

An ABORT statement may be placed anywhere within the BASIC program.

The ABORT statement displays the following message before terminating the program:

```
[B1] Run-time abort at line n
```

Line n is the program line number that contains the ABORT statement.

The STOP statement can also be used for program termination; STOP does not terminate a PROC. (Refer to the STOP statement, listed alphabetically in this chapter.)

```
PRINT 'PLEASE ENTER FILE NAME':
INPUT FN
OPEN FN TO FFN ELSE ABORT 201, FN
```

This program requests a file name from the user and attempts to open the file. If an incorrect file name is entered, the standard system error message "[201] 'xxx' IS NOT A FILE" is printed, followed by the BASIC run-time message "[B1] Run-time abort at line n". The program is then terminated.

ABS Function

The ABS function returns an absolute value.

Syntax

ABS(expr)

expr any numeric expression; if expression is non-numeric or null, zero is assumed

```
A = 100
B = 25
C = ABS(B-A)          The value 75 is assigned to C.

Z = ""
A = ABS(Z)            The value 0 is assigned to A.
```

ALPHA Function

The ALPHA function evaluates a specified expression for alphabetic characters.

Syntax

ALPHA(expr)

expr contains characters to test

Description

Alphabetic characters are the 26 letters of the alphabet, in upper or lower case. The null string ("") is not considered to be an alphabetic string.

The ALPHA function returns a value of true (1) if all characters in the given expression evaluate are alphabetic; if not, it returns a value of false (0).

```
IF ALPHA(I CAT J) THEN GOTO 5
```

Transfers control to statement label 5 if current value of both variables I and J are alphabetic strings.

```
PRINT ALPHA(N) OR ALPHA(M)
```

Prints a value of 1 if the current value of either M or N is an alphabetic string.

ASCII Function

The ASCII function returns the ASCII value of an EBCDIC string.

Syntax

ASCII(expression)

expression string value to be converted from EBCDIC to ASCII

Description

The inverse function, EBCDIC, is discussed as a separate function. (Please refer to the EBCDIC function, listed alphabetically in this chapter.)

For a list of ASCII values, refer to Appendix D.

```
READT X ELSE STOP
Y = ASCII(X)
```

Reads a record from tape and assigns value to variable X. Assigns ASCII value of record to variable Y.

BEGIN CASE Statement

The BEGIN CASE statement is the first statement in the CASE statement sequence.

Please refer to the CASE statement for information about the entire CASE statement sequence.

BREAK Statement

The BREAK statement controls the BREAK key on the terminal through a BASIC program.

Syntax

BREAK {KEY} OFF
BREAK {KEY} ON
BREAK {KEY} expr

expr determines setting; must evaluate to a numeric value; a value of zero (0) is equivalent to OFF, and all other values are equivalent to ON.

Description

The BREAK OFF statement disables the BREAK key on the terminal. When disabled, the BREAK key cannot be used to stop a program from executing. This is useful when the BREAK key must not be operative during critical processes such as file updates.

The BREAK ON statement enables the BREAK key on the terminal. When enabled, the BREAK key is set to its normal state.

Setting the BREAK key is cumulative. That is, each time a BREAK statement is encountered, the system increments or decrements by one, as appropriate, a counter called the BREAK inhibit counter. For example, if three BREAK OFF statements are encountered, three BREAK ON statements must be encountered before the BREAK key is enabled. Therefore, an equal number of BREAK ONs and BREAK OFFs must be executed to restore a breakable status.

The expression form of BREAK KEY increments or decrements the BREAK inhibit counter by one, as appropriate.

```
BREAK OFF                    Disable BREAK key
GOSUB UPD.FILES
BREAK ON                     Enable key after file update

F = 0
BREAK F                      Disable BREAK key

F = 1
BREAK KEY F                  Enable BREAK key
```

CALL Statement

The CALL statement provides external subroutine capabilities for a BASIC program. An external subroutine can be called directly or indirectly.

Syntax

CALL {@}subr.name {(argument list)}

@	specifies an indirect call; subroutine name has been assigned to a variable
subr.name	item name of a program; if @ is not used, the name cannot have any characters other than letters, numbers, and periods in it. If the @ is present, subr.name is a variable containing the name of the external subroutine to be called
argument list	one or more expressions, including literal values, separated by commas, that represent actual values passed to the subroutine. The argument list can pass an array to a subroutine by preceding the array argument with the word MAT. An argument list may continue on multiple lines; each line except the last must conclude with a comma and comments that start with an asterisk (*) may be included on each continuation line. The comments must be separated from argument list by a semicolon (;).

Description

An external subroutine is a subroutine that is compiled separately from the program or programs that call it.

The CALL statement first looks for the subroutine as a cataloged entry in the account's master dictionary; if the subroutine is not there, the CALL statement then looks for a compiled program in the file that contains the mainline program that is being executed.

Subroutines may be opened to a variable by the OPEN statement, then used in an indirect call. This greatly enhances the performance of indirect subroutine calls. For details, refer to the OPEN statement listed alphabetically in this chapter.

The CALL statement with no @ is a direct call, and transfers control to the external subroutine.

There is no correspondence between variable names in the calling program and variable names in the subroutine. The only information passed between the calling program and the subroutine are the values of the arguments; the values correspond in order of the variables in the argument list.

Variables may also be declared in COMMON and named COMMON areas and passed between the main program and its subroutines. For details, refer to the COMMON statement listed alphabetically in this chapter.

External subroutines may call other external subroutines, including themselves.

The SUBROUTINE statement must be used in conjunction with CALL. The called external subroutine must begin with a SUBROUTINE statement and must contain a RETURN statement. For details, refer to the SUBROUTINE statement listed alphabetically in this chapter.

The CALL statement checks to see that the appropriate number of arguments has been passed to the subroutine by the calling program. If not, CALL prints an error message and aborts to the BASIC Debugger.

If the correct number of arguments has been passed, the CALL statement evaluates the arguments and assigns their values to the corresponding variables named in the subroutine's SUBROUTINE statement. These variables may subsequently be assigned new values by the subroutine.

When the RETURN statement in the subroutine is executed, control is returned to the CALLing program and variables used as subroutine arguments are updated to reflect the most recent values of the corresponding variables in the subroutine. Constants and literals used as subroutine arguments are not affected.

Care should be taken not to update the same variable referenced by more than one name in an external subroutine. This can occur, for example, if a variable in COMMON is also passed as a subroutine argument.

If the execution of the subroutine is terminated before the RETURN is executed (such as by executing a STOP statement), control never returns to the calling program.

`CALL REVERSE (A,B)`	Subroutine REVERSE has two arguments.
`CALL REPORT`	Subroutine REPORT has no arguments.
`CALL VENDOR (NAME, ; * Comments` `ADDRESS, NUMBER)`	The arguments for VENDOR are continued on to the next line; comments can be included on multiple lines
`CALL DISPLAY (A,B,C)`	Subroutine DISPLAY has three argument.

Passing Arrays

Dimensioned arrays can be passed as parameters to the external subroutines by preceding the array name with MAT:

CALL subr.name (MAT array.name)

The array must be dimensioned in both the calling program and the subroutine. Array dimensions may be different, as long as the total number of elements matches.

Arrays are copied in row major order; that is, all columns in row 1 are copied before the first column in row 2.

Note: *An element in an array can be passed or the entire array can be passed; however, they should not be passed in the same CALL statement. If both an element from an array and the array itself are passed, the results are unpredictable.*

Calling Program

```
DIM A(4,10),B(10,5)
CALL REV (MAT A,MAT B)
```

Subroutine

```
SUBROUTINE REV (MAT C,MAT B)
  DIM C(4,10), B(50)
```

Subroutine REV accepts two input array variables, one of size 40 and one of size 50 elements.

```
DIM X(4,5)
CALL COPY (MAT X)
END
```

```
SUBROUTINE COPY (MAT A)
  DIM A(10,2)
  PRINT A(8,1)
  RETURN
END
```

In this subroutine the parameter passing facility is used to copy array X specified in the CALL statement of the calling program into array A of the subroutine. Printing A(8,1) in the subroutine is equivalent to printing X(3,5) in the calling program.

CASE Statement

The CASE statement provides conditional selection of a sequence of BASIC statements.

Syntax

```
BEGIN CASE
    CASE expression
        statements
    CASE expression
        statements
      .
      .
END CASE
```

The indentations are for clarity and are not required.

Description

If the logical value of the expression is true (non-zero), the statements that immediately follow, up to the next CASE or END CASE, are executed, then control passes to the statement following END CASE. If the expression is false (zero), control passes to the next CASE expression.

The expression CASE 1 is always true and can be used to force control to a series of statements.

`BEGIN CASE`
`  CASE Y=B`
`  Y=Y+1`
`END CASE`
	Increment Y if Y is equal to B. This is equivalent to the statement
	`IF Y=B THEN Y=Y+1.`

`BEGIN CASE`
`  CASE A=0; GOTO 10`
`  CASE A<0; GOTO 20`
`  CASE 1; GOTO 30`
`END CASE`
	Program control branches to the statement with label 10 if the value of A is zero; to 20 if A is negative; or to 30 in all other cases (CASE 1 is always true).

CHAIN Statement

The CHAIN statement terminates program execution and passes control to a specified TCL command. Control is not returned to the BASIC program that invokes the CHAIN statement.

Syntax

CHAIN "TCL.command"

TCL.command — any valid verb or PROC name in the user's Master Dictionary

Description

The TCL command may be used to initiate another BASIC program using values from the first program. The variables in one program that are to be passed to another program must be in the same location. (Variables are allocated in the order in which they first appear in a program, except that arrays are allocated in the order of their DIM statements after all other variables are allocated.) The variable names do not need to correspond; only the order is significant.

In order to use the variables from the first program in the CHAINed-to program, the program must be executed with the RUN verb with the I option. (The I option specifies that the variables are not to be initialized.) This causes the variables to take on values from variables in the first program, since variable data is always stored beginning at the same location in a user's workspace.

Caution! *The workspace areas used for variable storage are also used by other system software. Their contents cannot be guaranteed when CHAINing from one BASIC program to another if there is any intermediate processing. For example, CHAINing to a PROC that performs a Recall SELECT statement before it invokes a BASIC program with the I option, causes the contents of the BASIC program's variables to be unpredictable.*

It is illegal to CHAIN from an external subroutine, but legal to CHAIN to a program that calls a subroutine.

CHAIN "RUN FN1 LAX (I)"	Executes program LAX in file FN1. I option specifies that data area is not to be initialized; the program invoking the CHAIN statement passes values to program LAX.
CHAIN "RUN BP ABC"	Invokes the program ABC in file BP. Because the I option is not used, values are not passed to program ABC.

Program ABC
`A=500`
`B=1;C=2`
`CHAIN "RUN BP XYZ (I"`
`END`
	Executes program XYZ. The I option specifies that the variable data area is not to be initialized; thus, program ABC passes the values "500", "1", and "2" to program XYZ.

Program XYZ:
`PRINT X`
`PRINT Y`
`PRINT Z`
`END`
	Program XYZ, in turn, prints the values "500", "1", and "2" since they were allocated and passed in that order.

`CHAIN "LISTU"`	Invokes the LISTU PROC.
`CHAIN "LIST CUSTOMERS"`	Invokes the LIST Recall Verb.

CHAR Function

The CHAR function converts a numeric value to its corresponding ASCII character value.

Syntax

CHAR(expression)

expression — numeric value to be converted to ASCII character string value; if the value of expression is greater than 255, then CHAR(expression) = CHAR(expression MOD 256).

Description

CHAR always returns one character:

The inverse function, SEQ, is discussed as a separate function. For details, please refer to the SEQ function, listed alphabetically in this chapter.

For a complete list of ASCII codes, refer to Appendix D of this manual.

`VM = CHAR(253)`	Assigns the string value for a value mark to variable VM.
`X = 252` `SVM = CHAR(X)`	Assigns the string value for a subvalue mark to variable SVM.

`FOR I = 65 TO 90`
`  PRINT CHAR(I)`
`NEXT I`
	Prints upper case letters of the alphabet.

CLEAR Statement

The CLEAR statement is used to initialize variables to a value of zero.

Syntax

CLEAR

Description

The CLEAR statement assigns the value 0 to all simple variables and array variables. The CLEAR statement does not initialize COMMON or named COMMON variables.

The CLEAR statement may appear anywhere in a program.

```
X = 7
  .
  .
CLEAR                    Assigns zero to all variables.
```

CLEARDATA Statement

The CLEARDATA statement removes all data that has been pushed on to the stack with the DATA statement.

Syntax

CLEARDATA

Description

The CLEARDATA statement is useful to ensure that the stack is empty so that terminal input can be requested.

All data previously placed on the stack and not yet used is removed, even when that data stack has been established by a PROC.

After CLEARDATA has been executed, subsequent INPUT statements will request terminal input unless another DATA statement is executed between the CLEARDATA and the INPUT statements.

For information on clearing the type-ahead buffer, see the INPUTCLEAR statement, listed alphabetically in this chapter.

```
DATA '123'
DATA '456'
DATA '789'
INPUT A;PRINT 'A = ':A
INPUT B;PRINT 'B = ':B
CLEARDATA
INPUT C;PRINT 'C = ':C
```

The first two stacked DATA elements are used to satisfy the first two INPUT statements. The third stacked DATA element ('789') is removed from the stack, allowing input to be requested from the terminal to use as the value of variable C.

result:

```
A = 123
B = 456
?    (waiting for input from the terminal)
```

CLEARFILE Statement

The CLEARFILE statement clears all data from a specified file.

Syntax

CLEARFILE {file.var} {ON ERROR statements}

file.var — the variable to which the file which was previously assigned via the OPEN statement; if the file.var is omitted, the internal default file variable is used (thus specifying the file most recently opened without a file variable)

statements — statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be cleared due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when clearing local files.

The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.

Description

CLEARFILE deletes the data in the file, but not the file itself.

The BASIC program aborts with the appropriate error message if the specified file has not been opened prior to the execution of the CLEARFILE statement. (For a description of run-time messages, see Appendix B.)

UltiNet Considerations

The ON ERROR clause allows the program to retrieve the UltiNet error number and take appropriate action. Such action could, for instance, include printing the associated message text via a PUT statement or STOP statement, and resuming or terminating program execution.

If a remote file cannot be cleared due to network errors and no ON ERROR clause is present, the program terminates with an error message.

```
OPEN 'FILEA' TO A ELSE STOP
OPEN 'DICT FILEB' TO D.B ELSE STOP
CLEARFILE A
CLEARFILE D.B
```

Clears the data section of file FILEA and the dictionary section of FILEB.

```
OPEN 'ABC' ELSE STOP 201, 'ABC'
READ Q FROM 'IB3' ELSE STOP 780, 'IB3'
IF Q<5>='TEST' THEN CLEARFILE
```

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

CLEARSELECT Statement

The CLEARSELECT statement cancels a specified select list.

Syntax

CLEARSELECT {select.variable}

select.variable — select list to clear; if omitted, the program's internal default select variable is used

Description

Once the list is cleared, any subsequent READNEXT using the cleared select variable executes its ELSE statements (if any).

For more information on select lists, see the READNEXT and SELECT statements, list alphabetically in this chapter.

```
    SELECT MASTER.FILE TO MASTER.LIST
      .
      .
100 READNEXT CUST FROM MASTER.LIST ELSE GOTO 999
      .
      .
    CLEARSELECT MASTER.LIST
    GOTO 100
```

A select list is created in the variable MASTER.LIST. The list is later cleared. If the READNEXT statement is executed after the CLEARSELECT statement, the program transfers to the routine at 999.

CLOSE Statement

The CLOSE statement closes a file by breaking the connection between that file and a file variable. The file must have been previously connected to the file variable via an OPEN statement.

Syntax

CLOSE {file.var} {ON ERROR stmts}

file.var	file variable to use in closing the file; if file.var is omitted, the internal default file variable is assumed
ON ERROR stmts	statements to be executed if the file is a remote file accessed via UltiNet and it cannot be closed due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when local files are being closed. The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.

Description

When an Ultimate data file is opened, file items are always read into and written from a file variable. The OPEN statement establishes a connection between the file and the BASIC file variable. The file variable may be explicitly named in the OPEN statement. If no file variable is named, the internal default file variable is used.

The CLOSE statement closes the file indicated by the file variable, or by the internal default file variable if no file variable is specified. In the latter case, it would close the file most recently opened by an OPEN statement without a file variable. If the file is not currently connected to the file variable, an error message is generated and the program aborts to the BASIC debugger. For more information about opening files, refer to the OPEN statement, listed alphabetically in this chapter.

Closing a file breaks the connection between a file and the specified file variable. In order to use the specified closed file variable again in statements such as READ or WRITE, the variable must be reconnected to a file by another OPEN statement.

If the file is opened to more than one file variable, only the file variable explicitly stated in the CLOSE statement is disconnected; the file remains connected to all other file variables.

Local files do not need to be closed with a CLOSE statement. A file is implicitly closed whenever a file variable (including the internal default file variable) is assigned a new value, such as in an OPEN statement or Assignment statement. That is, if a file has been opened to a file variable, it is not necessary to CLOSE the file variable before assigning it a different value. Also, all open files are automatically closed when a program terminates execution.

When working with remote files, however, the advantage of closing a file when it is no longer needed in a program is that the corresponding remote open-file table entry is freed. Since the number of entries in this table is limited, freeing unused connections could allow greater use of the network. On the other hand, excessive opening and closing of remote files would merely increase network traffic and decrease program efficiency.

UltiNet Considerations

The purpose of the ON ERROR clause is to allow the program to retrieve the UltiNet error number and take appropriate action. Such action could, for instance, include printing the associated message text via a PUT statement or STOP statement, and resuming or terminating program execution. For more information, see the PUT and STOP statements, listed alphabetically in this chapter.

If a remote file cannot be closed due to network errors and no ON ERROR clause is present, the program terminates with an error message.

`CLOSE`	Closes file most recently opened without a file variable.
`CLOSE F`	Closes file OPENed to file variable F.

`CLOSE F ON ERROR`
`  ERRNUM=SYSTEM(0)`
`  GOSUB PROCESSERR`
`  GOTO TOP`
`END`
	Closes file opened to F, or retrieves error number and performs local subroutine on UltiNet error number.

COL1 and COL2 Functions

The COL1() and COL2() functions return the numeric values of the column positions immediately preceding and immediately following the substring selected by the most recent FIELD function.

Syntax

COL1()
COL2()

Description

The COL functions are used in conjunction with the FIELD function. COL1() returns the numeric value of the column position immediately **preceding** the substring selected by the most recent FIELD function. COL1() returns zero if the substring is not found.

COL2() returns the numeric value of the column position immediately **following** the substring selected by the most recent FIELD function. COL2() returns zero if the substring is not found.

```
B = FIELD("XXX.YYY.ZZZ.555",".",2)
BEFORE = COL1()
```

Assigns the numeric value 4 to variable BEFORE; the value "YYY", which is returned by the FIELD function, is preceded in the original string by column position 4.

```
B = FIELD("XXX.YYY.ZZZ.555",".",2)
AFTER = COL2()
```

Assigns the numeric value 8 to the variable AFTER; the value "YYY", which is returned by the FIELD function, is followed in the original string by column position 8.

```
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.

COMMON Statement

The COMMON statement is used to pass values between programs.

Syntax

COM{MON} {/name/} var1,var2,...

name	name to be given to common area; must be enclosed within slashes (/)
var1,var2,...	names of variables to be included in COMMON area

Description

The COMMON statement allows one or more variables to be shared by a main program and its external subroutines without having to pass the variables as parameters on each subroutine call. The list of variables may be continued on several lines; each line except the last must end with a comma.

File variables may be declared in COMMON; files that are opened in one program or subroutine are shared by all.

Fixed dimensioned arrays that are to be used in common must have their dimensions specified in a COMMON statement rather than in a DIM statement. This is accomplished by specifying the dimensions in parentheses after the array name, as in the DIM statement; for example:

```
COMMON A(10)
```

A variably dimensioned array may be specified in a COMMON statement by using 0 (zero) as the size of the array. However, the array **must** then also be specified in a DIM statement. Variably dimensioned arrays cannot be specified in named COMMON statements.

The values in the COMMON areas can be examined and modified within a program as necessary.

All variables, including COMMON variables, are allocated space in the order in which they appear. Therefore, to ensure that they have the expected values, COMMON variables should be declared before any

other variables. For more information, see the section Variable Allocation in Chapter 2.

COMMON variables differ from subroutine arguments used with the CALL statement in that the actual storage locations of COMMON variables are shared, whereas subroutine arguments are copied to local variables on entry to a subroutine and copied back to the calling program on exit. COMMON variables, then, may be used to increase program efficiency.

COMMON variables (including arrays and file variables) may be referred to by different names in different routines since they are accessed by their relative position in the COMMON area, rather than by name. However, it is not recommended to use different names.

```
MAINPROG                        SUBR
  COMMON X, Y, Z(5)               COMMON Q, R, S(5)
```

Variable X in MAINPROG above refers to the same location as variable Q in SUBR; Y in MAINPROG refers to the same location as R in SUBR; and array Z in MAINPROG refers to the same set of locations as array S in SUBR.

In MAINPROG:

```
COMMON A,B,C(10)
A = "NUMBER"
B = "SQUARE ROOT"
FOR I = 1 TO 10
  C(I) = SQRT(I)
NEXT I
CALL SUBPROG
PRINT "DONE"
```

Variables A, B, and array C are allocated space before any other variables.

Subroutine call to program SUBPROG.

In SUBPROG:

```
COMMON X(2),Y(10)
PRINT X(1), X(2)
FOR J = 1 TO 10
  PRINT J, Y(J)
NEXT J
RETURN
```

The 2 elements of array X contain respectively, the values of A and B from the main-line program. The array Y contains the values of C from the main-line program.

Returns to main-line program

```
COMMON X(0)
DIM X(0)
```

Variable dimensioned array in COMMON must also be defined in a DIM statement.

Named COMMON Areas

Variables within named COMMON areas are associated with the named area; the order in which the COMMON **areas** are defined is not important. However, within each named COMMON area, the order in which the COMMON **variables** are defined is important as the variables are allocated space in the order in which they appear.

Variables in named COMMON areas are initialized to nulls.

The named COMMON statements must be executed; the areas are assigned at runtime and if the statements are branched around or in any other way not executed, the named COMMON areas are not assigned.

Named COMMON areas are associated with a port; each port on the system has its own set of named COMMON areas. The TCL command LIST-NAMED-COMMON can be used to list the set of named COMMON areas for the current port.

Named COMMON areas remain until the port is logged off or until the area is canceled by the RELEASE /name/ statement. Values of variables in a named COMMON area retain their value as long as the COMMON area remains.

Variables in named COMMON areas are available to all programs that are executed, not just subroutines that are CALLed. Two different programs can be executed with the same named COMMON area; the value of the variables are passed from one program to the other. If you nest EXECUTE statements within programs, a named COMMON established at one level is available to all other levels.

Each named common area counts as one variable, regardless of the number of variables in the area. The maximum number of variables in a named COMMON area is 3223. More than one named COMMON area can be used in a program.

In program 1:

```
COMMON /PEOPLE/ NAME,ADDRESS,STATE,TOTAL
  .                          Named common areas with four
  .                          elements
IF STATE = 'CA' THEN EXECUTE 'CA.CALC' CAPTURING TAX
```

In program CA.CALC

```
COMMON /PEOPLE/ NAME,ADDRESS,STATE,TOTAL
TAX = TOTAL...               Uses named common areas defined in
PRINT TAX                    program that executed this program
END
```

CONVERT Statement

The CONVERT statement converts all occurrences of a character in a string to another character.

Syntax

CONVERT search.expr TO replace.expr IN var

search.expr — expression evaluating to a list of one or more characters to be converted

replace.expr — expression evaluating to a corresponding list of replacement characters

var — string variable in which characters are to be converted

Description

The search expression and replace expression correspond on a 1-for-1 basis. If the search expression contains more characters than the replace expression, any of the excess search characters found in var are deleted.

If the replace expression contains more characters than the search expression, the excess replacement characters are ignored.

```
UPPER = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
LOWER = 'abcdefghijklmnopqrstuvwxyz'
STRING = 'the quick brown fox'
CONVERT LOWER TO UPPER IN STRING
```

result:

```
STRING = 'THE QUICK BROWN FOX'
```

All lower case letters were converted to their upper case form. This example is equivalent to using the output conversion function OCONV(STRING,'MCU') (mask characters upper case).

```
A = "John Smith, 12 Main Street, Costa Mesa, CA"
CONVERT "," TO @FM IN A
```

result:

```
A = "John Smith^ 12 Main Street^ Costa Mesa^ CA"
```

The comma was converted to an attribute mark in A.

COS Function

The COS trigonometric function returns the cosine of an angle expressed in degrees.

Syntax

COS(expression)

expression specifies the number of degrees in the angle

Description

Values that are less than 0 or greater than 360 are adjusted to that range by modulo 360.

```
A = 60
PRINT COS(A)                Prints 0.5

A = 420
PRINT COS(A)                Adjusts to 60, then prints 0.5.
```

COUNT Function

The COUNT function counts the number of occurrences of a substring within a string.

Syntax

COUNT(string,substring)

string — characters to search; may be any valid expression and may contain any number of characters

substring — characters to search for; may be any valid expression and may contain any number of characters

Description

If the substring is not found, COUNT returns a value of zero.

If the substring specification is null, COUNT function returns the number of characters in the string (that is, a null matches on any character.)

A variation of the COUNT function is DCOUNT, which is particularly useful for counting elements in dynamic arrays. (See DCOUNT, listed alphabetically in this chapter.)

If the substring contains a repeating sequence, the count overlaps the sequences. For example, if the string contains the sequence ABABABAB and you wish to count the occurrences of ABAB, 3 is returned as the count.

`A = "1234ABC5723"` `X = COUNT(A,'23')`	Value returned in X is 2 since there are two occurrences of '23' in the string A.
`A = "12":AM:"ABC":AM:"57"` `X = COUNT(A,AM)`	Value returned in X is 2 since there are two occurrences of an attribute mark (AM) in the string A.
`X = COUNT('ABCDEFG',"")`	Value returned in X is 7 since a null substring matches any character.

CRT Statement

The CRT statement displays specified output at the terminal and is a synonym for the DISPLAY statement.

Syntax

CRT expression

expression may contain any valid BASIC expression, formatting operators, and @ functions; if the print-list is absent, only a carriage return and line feed are output

Description

The CRT statement can be used interchangeably with DISPLAY to print data at the terminal. The CRT statement is similar to the PRINT statement, except for the following:

- output from the CRT statement is always directed to the terminal, regardless of PRINTER ON statements or the P option on the RUN verb
- output of the CRT statement cannot be captured using the OUT. or CAPTURING clause in an EXECUTE statement; any output specified by CRT continues to be directed to the terminal
- output is not counted for purposes of determining line spacing for HEADING, FOOTING, or PAGE statements

```
PRINTER ON                  Causes PRINT statements to print
LOOP                        on printer.
 CRT "ALIGNED?":            Displays message on terminal.
 INPUT ANS                  Requests operator input.
UNTIL ANS="Y" DO
 PRINT FIRSTLINE            Prints on printer.
 PAGE                       Eject page.
REPEAT                      Repeat until "Y" entered at terminal.
```

DATA Statement

The DATA statement is used to store data for stacked input.

Syntax

DATA expression {,expression...}

expression may be any valid expression, and any number of expressions may be included in one DATA statement; each expression becomes one line of stacked input data. The list of expressions may continue on several lines; each line except the last must end with a comma.

Description

Each expression in a DATA statement generates one line of stacked input. Normally, an input request such as from an INPUT statement prints a prompt character on the terminal and waits for the user to enter data. When stacked input is present, however, each input request causes a line of data to be taken from the input stack, until the stack is empty or the program terminates and returns to TCL, at which time the input stack is unconditionally cleared.

DATA statements can be used to prestore input for other BASIC programs invoked by either the EXECUTE or CHAIN statement. One BASIC program can set up parameters using DATA statements and then invoke the next program, which retrieves the parameters with INPUT statements. For more information, see the EXECUTE and CHAIN statements listed alphabetically in this chapter.

Stacked input may also be generated by the EXECUTE statement and by a PROC. For more information, see the EXECUTE statement listed alphabetically in this chapter. For more information on PROCs, see the *Ultimate PROC Reference Guide.*

Note: *The first DATA statement in a program overwrites any unprocessed stacked input from a PROC.*

Stacked input is removed in the same order that it is added via DATA statements.

For information on clearing the data stack, see the CLEARDATA statement listed alphabetically in this chapter.

```
DATA A
DATA B
DATA C
CHAIN 'RUN BP TEST'
```

Stacks the values of A, B, and C for subsequent input requests. The first three input requests in program 'TEST' are satisfied by the stacked input.

DATE Function

The DATE function returns the current internal date.

Syntax

DATE()

Description

The internal date is the number of days since December 31, 1967.

The internal format is useful for sorting and comparisons, since the information is in numeric format.

For information on converting the internal date to external format, see the OCONV statement listed alphabetically in this chapter.

`Q = DATE()`	Assigns string value of current internal date to variable Q; for example, 7800
`PRINT OCONV(DATE(),'D')`	Converts the internal date to dd mon yyyy format, then prints it; for example, if internal date is 7800, prints 09 MAY 1989

DCOUNT Function

The DCOUNT function counts the number of substrings in a string that are separated by a specified delimiter. It returns the number of substrings counted.

Syntax

DCOUNT(string,delimiter)

string specifies the string to examine.

delimiter delimiter to use; may be string of more than one character.

Description

The function returns the number of substrings within the string that are separated by the delimiter. If the string is null, a value of zero is returned. If the delimiter is null, the value returned by DCOUNT is the number of characters in the string plus one.

Note that DCOUNT is similar to the COUNT function. However, the DCOUNT function differs from the COUNT function in that DCOUNT returns a count of substrings separated by the specified delimiter, rather than the number of occurrences of the delimiter within the string and, unless the string is null, is equivalent to COUNT + 1. (Please refer to the COUNT function, listed alphabetically in this chapter.)

The DCOUNT function is useful in manipulating Ultimate data files. It may be used to count the number of attributes in an item, or the number of values (or subvalues) within an attribute.

Note: *In the following examples, ^ represents an attribute mark,] represents a value mark.*

```
A = "123^456^ABC"
X = DCOUNT(A,@FM)
```
Value returned in X is 3.

```
A = "123]456^ABC]DEF]HIJ"
X = DCOUNT(A<1>,@VM)
```
Value returned in X is 2 as there are two values in the first attribute.

```
A = ""
X = DCOUNT(A,@FM)
```
Value returned in X is 0 since the string is null.

```
A = ABC^DEF^GHI^JKL
X = COUNT(A,@FM)
X = DCOUNT(A,@FM))
```
COUNT (number of delimiters) is 3
DCOUNT (number of substrings) is 4

DEL Statement

The DEL statement deletes the specified attribute, value, or subvalue from a dynamic array.

Syntax

DEL variable <attrib.no{, val.no{, subval.no}}>

variable	name of dynamic array
attrib.no	position of the attribute to be deleted
val.no	position of value to be deleted
subval.no	position of subvalue to be deleted

Description

The numbers must be enclosed in angle brackets. For example, <3,5,1> denotes attribute 3, value 5, subvalue 1.

This statement performs the same operation as the DELETE function, but in addition, the DEL statement stores the result back into the source variable. For more information see the DELETE function listed alphabetically in this chapter.

`DEL NAMELIST<5>`	Deletes attribute 5 from variable NAMELIST.
`DEL PAYHIST<2,4,6>`	Deletes subvalue 6 from value 4 in attribute 2 of variable PAYHIST.

DELETE Function

The DELETE function returns a dynamic array with a specified attribute, value, or subvalue deleted.

Syntax

DELETE(var,attrib.no{,val.no{, subval.no}})

var	dynamic array to be used in the function
attrib.no	position of the attribute to be deleted
val.no	position of value to be deleted
subval.no	position of subvalue to be deleted

Description

If val.no and subval.no have a value of 0 or are absent, the attribute specified by attrib.no is deleted. If val.no is present and subval.no is absent or has a value of 0, the value specified by val.no is deleted. If attrib.no, val.no, and subval.no are all non-zero, the subvalue specified by subval.no is deleted. The associated delimiter is also deleted.

The specified element is deleted in the returned array, not in the original array.

For a related statement, see the DEL statement listed alphabetically in this chapter.

`Y = DELETE(X,3)`	Assigns to Y the dynamic array obtained by deleting attribute 3 and its delimiter from dynamic array X.
`A=1;B=2` `DA = DELETE(DA,A,B)`	Deletes value 2 and its delimiter from attribute 1 of dynamic array DA.
`PRINT DELETE(X,7,1)`	Prints the dynamic array that results when value 1 of attribute 7 of dynamic array X is deleted.

DELETE Statement

The DELETE statement deletes an item from a file.

Syntax

DELETE {file.var,} item.id {ON ERROR stmts}

file.var	variable to which file was OPENed; if omitted, internal default file variable is used; the default is the file most recently opened without a file variable
item.id	name of item to delete
ON ERROR stmts	statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be accessed due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (For information on SYSTEM refer to the SYSTEM function listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when accessing local files. The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.

Description

No action is taken if a non-existent item is specified in the DELETE statement.

The BASIC program aborts with the appropriate error message if the specified file has not been opened prior to the execution of the DELETE statement. (Refer to Appendix B describing run-time error messages.)

UltiNet Considerations

The ON ERROR clause allows the program to retrieve the UltiNet error number and take appropriate action. Such action could, for instance, include printing the associated message text via a PUT statement or STOP statement, and resuming or terminating program execution.

If a remote file cannot be accessed due to network errors and no ON ERROR clause is present, the program will terminate with an error message.

`DELETE X, "XYZ"`	Deletes item XYZ in the file opened and assigned to variable X.
`Q="JOB"` `DELETE Q`	Deletes item JOB in the file opened without a file variable.

`DELETE X, "XYZ" ON ERROR`
`  ERRNUM=SYSTEM(0)`
`  GOSUB PROCESSERR`
`  GOTO TOP`
`END`
	Deletes item XYZ, or retrieves error number and performs local subroutine on UltiNet error number.

DIM Statement

A DIM statement declares the dimensions of an array.

Syntax

DIM array1.name(rows1{, cols1}) {,array2.name(rows2{,cols2})...}

array.name	variable to be used as name of array
rows	number of rows in the array; if the array is one-dimensional, rows is the number of elements in the array
cols	number of columns in array; if rows is zero, cols, if present, must be set to zero

Description

The DIM statement must precede any references to the array, and is therefore usually placed at the beginning of the program.

Any number of arrays may be dimensioned in one DIM statement. The list of arrays may continue on several lines; each line except the last must end with a comma.

Arrays can be redimensioned as necessary throughout the execution of the program.

The dimension parameters (rows,cols) may be literals (including 0) or variables. If variables are used, the size of the array is resolved at run time and the DIM statement must be executed. If the variable has not been assigned a value before the DIM statement is executed, the array is dimensioned to zero.

The maximum size to which an array can be dimensioned is 3223, which is the maximum number of variables in a program. If the array is dimensioned with a literal, each element in the array counts toward the total number of variables in the program. However, if the array is dimensioned to 0 or with a variable, the entire array counts as just one variable. Thus, you can have up to 3223 separate arrays (assuming you have no other variables) if the arrays are all dimensioned with 0 or variables. Each array dimensioned with 0 or a variable can later be redimensioned with up to 3223 elements.

If an array is dimensioned with a variable or to the value zero, and is subsequently used as the destination for MATPARSE or MATREAD, the array is automatically redimensioned using the number of attributes in the item as the new dimension for the array. This automatic redimensioning occurs each time a new item is read into the array. Immediately after the MATPARSE or MATREAD statement, the INMAT() function can be used to determine the current size of the array. *(Note: Two-dimensional arrays that are dimensioned to zero should not be used with MATPARSE and MATREAD.)*

If the array is redimensioned to a size with fewer elements than previously, the elements beyond the new size are cleared.

```
SZ = 0
DIM A(SZ)
OPEN 'TEST' TO TEST ELSE ABORT '201', 'TEST'
MATREAD A FROM TEST,'ITEM.1'
SZ = INMAT()                    Array is initially dimensioned to 0;
                                MATREAD redimensions it to size of
                                ITEM.1.

DIM MATRIX(10,12)               Specifies 10 by 12 matrix named
                                MATRIX.

DIM Q(10),R(10),                Specifies three arrays named Q, R,
    S(10)                       and S, each to contain 10 elements.

DIM M1(50,10),X(2)              Specifies 50 by 10 array named M1,
                                and two-element array named X.
```

DISPLAY Statement

The DISPLAY statement outputs data to the terminal and is a synonym for the CRT statement.

Syntax

DISPLAY {print.list}

print.list — may contain any valid BASIC expression, formatting operators, and @ functions; if the print-list is absent, only a carriage return and line feed are output

Description

The DISPLAY statement can be used interchangeably with CRT to print data at the terminal. The DISPLAY statement is similar to the PRINT statement in that both statements may be used to print data at the terminal, but DISPLAY differs in the following respects:

- output from the DISPLAY statement is always directed to the terminal, regardless of PRINTER ON statements or the P option on the RUN verb
- output of the DISPLAY statement cannot be captured using the OUT. or CAPTURING clause in an EXECUTE statement; any output specified by DISPLAY continues to be directed to the terminal
- output is not counted for purposes of determining line spacing for HEADING, FOOTING, or PAGE statements

```
PRINTER ON                      Causes PRINT statements to print
LOOP                            on printer.
 DISPLAY "ALIGNED?":            Displays message on terminal.
 INPUT ANS                      Requests operator input.
UNTIL ANS="Y" DO
 PRINT FIRSTLINE                Prints on printer.
 PAGE                           Eject page.
REPEAT                          Repeat until "Y" entered at terminal.
```

EBCDIC Function

The EBCDIC function returns the EBCDIC value of an ASCII string.

Syntax

EBCDIC(expression)

expression — string value to be converted from ASCII, the normal Ultimate string representation, to EBCDIC.

Description

The inverse of this function is the ASCII function. (Please refer to the ASCII function, listed alphabetically in this chapter.)

`B = EBCDIC(A)`	Assigns the EBCDIC value of variable A to variable B.

ECHO Statement

The ECHO statement enables or disables the echoing of characters at the user's terminal.

Syntax

ECHO OFF
ECHO ON
ECHO expression

OFF	disables the echo on the terminal; characters typed on the keyboard are not displayed on the screen
ON	enables the echo on the terminal; characters typed on the keyboard are displayed on the screen
expression	determines setting; must evaluate to a numeric value; a value of zero (0) is equivalent to OFF, and all other values are equivalent to ON

```
PRINT 'Enter password:  ':
ECHO OFF                    The value input for PW is not
INPUT PW                    displayed.
ECHO ON                     Subsequent input will be displayed.

DISPL = 0
ECHO DISP                   Input will not be displayed.

DISP = 1
ECHO DISP                   Input will be displayed.
```

END Statement

The END statement is used to designate the physical end of a program or the physical end of conditional statements.

Syntax

END

Description

Any statements appearing after an end-of-program END statement are ignored.

The END statement is not required at the end of a program.

The END statement is also used to designate the physical end of alternative sequences of statements within the IF statement and within statements ending with THEN, ELSE, LOCKED, or ON ERROR clauses. (Please refer to the description of the IF statement for details on using the END statement with alternative sequences of statements.)

```
   *
   A = 500
   B = 750
   C = 235
   D = 1300
REM COMPUTE PROFIT:
   REVENUE = A+B
   COST = C+D
   PROFIT = REVENUE - COST
REM PRINT RESULTS
   IF PROFIT  > 1 THEN GOTO 10
   PRINT "ZERO PROFIT OR LOSS"
   STOP                          If this path taken,program terminates
10 PRINT "POSITIVE PROFIT"
   END                           Physical program end
```

END CASE Statement

The END CASE statement is the last statement in the CASE statement sequence.

Please refer to the CASE statement for information about the entire CASE statement sequence.

ENTER Statement

The ENTER statement transfers control to a cataloged BASIC program and retains variable values from the first program.

Syntax

ENTER prog.name
ENTER @variable

prog.name item.id of the program to be ENTERed

variable contains name of program to be ENTERed

Description

The ENTER statement suppresses initialization of variables in the program being ENTERed in the same way the I option on the RUN verb suppresses initialization. This allows several programs which ENTER each other to be viewed as components of one large program, provided the variables in each individual program correspond correctly to their counterparts in the other programs.

Variables correspond based on the order in which they are declared or otherwise introduced in each program. COMMON statements may be used to ensure that the same variables are allocated in the same order (even if with different names) in all component programs.

It is permissible to ENTER a program that calls a subroutine, but it is not recommended that you ENTER a program from a subroutine. This is primarily because the way in which variables are assigned and the symbol table created for the BASIC debugger means that the values of defined variables are unpredictable.

Control is not returned to the BASIC program that executes the ENTER statement.

`ENTER PROGRAM.1`	Causes execution of the cataloged program "PROGRAM.1".

`N=2`
`PROG = "PROGRAM.":N`
`ENTER @PROG`
	Causes execution of the cataloged program "PROGRAM.2".

EOF Function

The EOF function tests either the argument list or the system message buffer for an end-of-file condition and returns the current status.

Syntax

EOF(ARG.)
EOF(MSG.)

ARG. the function examines the program's argument list; arguments are specified following the program name in the statement that invokes the program

MSG. the function examines the system message buffer, which contains the list of message numbers and associated parameters generated by the most recent EXECUTE statement

Description

The EOF function examines the specified list and returns a value of 1 if the end-of-file has been reached; otherwise, it returns a value of 0.

The values stored in ARG. or MSG. can be retrieved by the GET statement, which maintains a pointer into the list to determine the value to return. If the last GET statement attempted to read past the end of the specified list, the EOF function returns a "true" value (1); if not, it returns a "false" value (0). Thus, the EOF function allows a program to check for the end of the specified list .

Note: *The ELSE clause of the GET statement can also be used to determine the end of the list.*

```
LOOP
 GET(ARG.) X
UNTIL EOF(ARG.) DO
 PRINT X
REPEAT
```

Prints arguments stored in the ARG. list until the EOF function is true; then exits the loop.

EQUATE Statement

The EQUATE statement allows a symbol to be defined as the equivalent of a literal number, string constant, a variable, or CHAR() value.

Syntax

EQU{ATE} symbol TO equate.val {, symbol TO equate.val...}

symbol — name to assign to value; must be a previously undefined name. A symbol name has the same criteria as a variable name in that it starts with an alphabetic character followed by letters, numerals, periods, or dollar signs

equate.val — value to be assigned; may be a literal number or string, a variable, or an array element. The equate.val may also be a CHAR function; the CHAR function, however, is the only function allowed in an EQUATE statement

Description

The EQUATE statement must appear before the first reference to the symbol.

Any number of equated symbols can be defined in one EQUATE statement. The symbol list may be continued on several lines; each line except the last must end with a comma.

The EQUATE statement differs from an assignment statement where a variable is assigned a value via an = sign, in that there is no storage location generated for the symbol. Instead, the symbol becomes just another name for the equate.val. The advantage this offers is that the value is compiled directly into the object code and does not need to be re-assigned every time the program is executed.

The EQUATE statement is therefore particularly useful under the following two conditions:

- Where a constant is used frequently within a program, and therefore the program would read more clearly if the constant were given a symbolic name. For example, "AM" is the commonly used symbol for "attribute mark", one of the standard data delimiters.
- Where a MATREAD statement is used to read in an entire item from a file and disperse it into a dimensioned array. In this case, the

EQUATE statement may be used to give symbolic names to the individual array elements, which makes the program more meaningful. For example:

```
DIM ITEM(20)
EQUATE BIRTHDATE TO ITEM(1),
    SOC.SEC.NO. TO ITEM(2),
    SALARY TO ITEM(3)
```

In this case, the variables BIRTHDATE, SOC.SEC.NO. and SALARY are equivalent to the first three elements of the array ITEM. These meaningful names are then used in the remainder of the program.

`EQUATE X TO Y`	Symbol X and variable Y may be used interchangeably within the program.
`EQUATE PI TO 3.1416`	Symbol PI is compiled as the value 3.1416.
`EQUATE MINUS1 TO -1`	Symbol MINUS1 is compiled as the value -1.
`EQUATE AM TO CHAR(254)`	Symbol AM is equivalent to the ASCII character generated by the CHAR function.
`EQUATE PART TO ITEM(3),` `  NAME TO ITEM(4)`	Symbol PART is equivalent to element 3 of array ITEM, and NAME to element 4 of the same array.

ERRTEXT Function

The ERRTEXT function returns the text associated with a specified ERRMSG file item.

Syntax

ERRTEXT(errmsg.id{,errmsg.param{,...})}

errmsg.id	item.id of item in ERRMSG file item to be displayed; can be expression
errmsg.param	parameters used by ERRMSG item; can be expressions

Description

The ERRMSG file can contain multi-level data files, where each data file is in a different language. The system command SET-LANGUAGE is used to specify the particular data level that is used. The ERRTEXT function formats the message appropriately for each language. (For information on the SET-LANGUAGE command, see the *System Commands Guide, Version 2*.)

```
PRINTERR ERRTEXT('202',CUSTID)
```

result:

```
'custid' not on file.
```

This inserts the value of the variable CUSTID into error message 202, then prints the message on the last line of the terminal.

```
IF X = "" THEN
   Y = ERRTEXT(204,FILE.NAME)
   GOSUB PRINT.SUB
END
```

result:

If X is null, the text of ERRMSG 204, with file.name inserted, is assigned to the variable Y, and the program calls PRINT.SUB; the value of Y is [204] File definition *'file.name'* is missing.

EXECUTE Statement

The EXECUTE statement allows a BASIC program to execute any valid TCL command and use the results of the command in later processing.

Syntax

EXECUTE expression {{,{//}} redir.clause {{,{//}} redir.clause}...}

expression — contains string in the format of a TCL command just as it would be entered at the terminal; it may be a verb, PROC, or cataloged BASIC program, followed by any parameters and options.

// — provided for compatibility with earlier revisions of the Ultimate operating system; double slashes (//) cause a compilation error if they precede any parameter that does not end in a period (for example, IN.). In general, Ultimate advises using only a space or a comma as a separator between clauses.

redir.clause — specifies source or destination for data in executed statement; may be any of the following:

CAPTURING var — output to terminal from executed statement is redirected to var; if the statement being executed produces more than one line of data, each line in var is delimited by attribute marks; the last line of data is always terminated by an attribute mark; equivalent to OUT. > or OUT. = clause

IN. < expr
IN. = expr — data in expr is stacked as input for the executed statement; if the statement to be executed accepts more than one line of data, each line of data in expr must be delimited by attribute marks; equivalent to STACKING clause

OUT. > var OUT. = var	output to terminal from executed statement is redirected to var; if the statement being executed produces more than one line of data, each line in var is delimited by attribute marks; the last line of data is always terminated by an attribute mark; equivalent to CAPTURING clause
PASSLIST expr	expr contains select list to be redirected to statement being executed; the elements of the list, typically item.ids, must be delimited by attribute marks; there is an attribute mark after the last element; equivalent to SELECT. < clause
RETURNING var	returns all ERRMSG message numbers and parameters generated by executed statement; ERRMSG numbers are separated by attribute marks; parameters within a message are separated by value marks
RTNLIST var	select list generated by executed command is redirected to var; elements in the list, typically item.ids, are delimited by attribute marks; there is an attribute mark after the last element; equivalent to SELECT. > clause
SELECT. < expr	expr contains select list is to be redirected to statement being executed; the elements of the list, typically item.ids, must be delimited by attribute marks; there is an attribute mark after the last element; equivalent to PASSLIST clause
SELECT. > var	select list generated by executed command is redirected to var; elements in the list, typically item.ids, are delimited by attribute marks; there is an attribute mark after the last element; equivalent to RTNLIST
STACKING expr	data in expr is stacked as input for the executed statement; if the statement to be executed accepts more than one line of data, each line of data in expr

must be delimited by attribute marks; equivalent to IN. < or IN. =

Description

Data stacked by the STACKING and IN. parameters can be used by other BASIC programs and by TCL-II verbs such as COPY and ED. Data cannot be stacked for use in PROC statements.

After the statement is executed, program control returns to the statement following the EXECUTE statement. If the EXECUTE statement changed the operating environment of the system in any way, the environment is not restored. Commands that may change the operating environment include the following:

- BREAK-KEY-OFF/BREAK-KEY-ON
- OFF (the system logs off and command is **not** returned to the BASIC program)
- spooler verbs such as SP-ASSIGN
- tape verbs such as T-ATT
- TABS
- terminal verbs, such as TERM

The following commands have no effect when EXECUTEd:

- CHARGE-TO
- LOGTO

The GET(MSG.) form of the GET statement may be used to obtain, one at a time, the messages generated by the statement that was executed.

When formatting the EXECUTE statement in your BASIC program, you may begin a new line after any comma.

Select Lists

A select list produced by an EXECUTE statement cannot be automatically carried over to the next EXECUTE statement. It can be redirected or used in a READNEXT statement in the same program. Thus, the following lists all items in MD:

```
EXECUTE "SELECT MD 'ED'"
EXECUTE "LIST MD"
```

A select list can be used as follows to list the selected items:

```
EXECUTE "SELECT MD 'ED'", RTNLIST X
EXECUTE "LIST MD", PASSLIST X
```

A select list can be used as follows to read the next item:

```
   EXECUTE "SELECT MD 'ED'"
10 READNEXT ID ELSE STOP
   PRINT ID
   GOTO 10
```

When the select list is stored in a variable, the variable may be used in the FROM parameter in a READNEXT statement.

```
   EXECUTE "SELECT MD 'ED'" RTNLIST X
10 READNEXT ID FROM X ELSE STOP
   PRINT ID
   GOTO 10
```

The select list may be used as a dynamic array in that elements may be retrieved directly from X without affecting its function as a list. For example, A = X<17> will put the 17th item.id into A and X can still be used in a READNEXT statement. However, the list is **not** a dynamic array. If an element is changed in the list, the list is then converted to a dynamic array and can no longer be used as a select list. In that case, to use READNEXT with the data, the program can SELECT the dynamic array to a list.

Note: *Unlike a dynamic array, a select list can contain more than 32,767 items; however, you cannot extract past the 32,767th element in the list.*

If a program uses DCOUNT to count the number of item.ids in the select list, the number returned is one higher than the actual number of elements in the list.

The RETURNING and CAPTURING clauses have no effect when a SELECT verb is executed.

Note: *The $COMPATIBILITY compiler directive affects the way EXECUTE functions with select lists and data stacks; for information, see the $COMPATIBILITY directive.*

```
EXECUTE "WHO"
```

The command WHO is executed; the output is displayed on user's screen; program control continues in sequence.

```
EXECUTE "WHO", OUT. > X
IF X<1> # "0 SYSPROG" THEN
 PRINT "MUST BE ON LINE 0"
 STOP
END
```

The command WHO is executed; the output is redirected to variable X. X is tested for access to program, resulting in either a message and halt or program execution.

```
EXECUTE "COPY BP PROG1",
 STACKING "COPY.PROG1",
 RETURNING MSGS
```

The command COPY is executed, using the string "COPY.PROG1" as stacked input for COPY. The ERRMSG number and any parameters produced for it are returned in MSGS.

```
EXECUTE "ED BP X",
 IN. < "L22":@FM:"EX",
 OUT. > X
```

The EXECUTE statement allows multiple redirection variables. Two lines of data, "L22" and "EX" are redirected to the command ED. The output is redirected to variable X.

```
EXECUTE 'SELECT EMPFILE WITH SAL >= "10000"' RTNLIST X
EXECUTE 'LIST EMP.ADDR.FILE' PASSLIST X
```

In the first EXECUTE statement, the select list is redirected to variable X, with the item.ids separated by attribute marks. The select list is then redirected to the LIST command in the second EXECUTE statement.

EXIT Statement

The EXIT statement transfers control out of a program loop initiated by a LOOP statement.

Syntax

EXIT

Description

When executed, EXIT transfers control to the next statement after the REPEAT statement of a loop. When loops are embedded within other loops, each EXIT transfers control to the statement after the next REPEAT. The EXIT statement must be used within a LOOP ... REPEAT program loop.

```
LOOP
   READNEXT ID ELSE EXIT
   GOSUB PROCESSIT
REPEAT
PRINT "DONE"
```

Subroutine PROCESSIT is called after each value from a preselected list is read by READNEXT. When the list is exhausted, the program loop is exited, causing the message"DONE" to be printed.

EXP Function

The EXP function returns the value of the natural logarithm base *e* (2.7183) raised to a specified power.

Syntax

EXP(expression)

expression power to which *e* is raised; may be any numeric expression

Description

The EXP (exponential) function raises the number *e* to the value of the expression. The EXP function is the inverse of the LN (natural logarithm) function.

Note: *The value returned by the EXP function is not affected by the PRECISION statement.*

```
PRINT EXP(1)                    Prints 2.718281828459

A = EXP(10)
PRINT A                         Prints 22026.46579474
```

EXTRACT Function

The EXTRACT function returns an attribute, a value, or a subvalue from a dynamic array.

Syntax

EXTRACT(expr,attrib.no{,val.no{,subval.no}})

expr	dynamic array to extract data from
attrib.no	position of the attribute to be extracted
val.no	position of the value to be extracted
subval.no	position of the subvalue to be extracted

Description

If val.no and subval.no are absent or have a value of 0, the attribute specified by attrib.no is extracted. If val.no is present and subval.no is absent or has a value of 0, the value specified by val.no is extracted. If attrib.no, val.no, and subval.no are all non-zero, the subvalue specified by subval.no is extracted.

The EXTRACT function has the same effect as following a dynamic array reference by attribute, value, and subvalue numbers in angle brackets. That is, EXTRACT(X,4,1) is equivalent to X<4,1>.

`Y=EXTRACT(X,2)`	Assigns attribute 2 of dynamic array X to variable Y.

`A=3`
`B=2`
`Q1=EXTRACT(ARR,A,B,A+1)`
	Assigns subvalue 4 of value 2 of attribute 3 of dynamic array ARR to variable Q1.

`IF EXTRACT(B,3,2)>5 THEN`
`  PRINT MSG`
`  GOSUB 100`
`END`
	If value 2 of attribute 3 of dynamic array B is greater than 5, the value of MSG is printed and a subroutine call is made to statement 100.

FADD Function

The FADD (floating point addition) function adds two floating point numbers and returns the result as a floating point number.

Syntax

FADD(fx, fy)

fx valid floating point number to be added

fy valid floating point number to be added

Description

A standard or string number must be converted to floating point before it is used in the FADD function. The FFLT function is provided to convert a number to floating point format. The FFIX function is provided to convert a floating point number to a string number. (Please refer to the FFLT and FFIX functions listed alphabetically in this chapter.)

If either fx or fy contains a non-floating point value, an error message is generated.

The result of the FADD function is a floating point number. Thus, the function can be used in any expression where a floating point number would be valid.

`TOTAL=FADD(SUB1,SUB2)`	Assigns sum of variables SUB1 and SUB2 to variable TOTAL.
`A=FFLT('1.0304')` `B=FADD(A,A)`	Assigns to variable A the floating point value of 1.0304 (10304E-4), then assigns the value of adding A to itself to variable B.
`X=FADD(A, FADD(B,C))`	Uses floating point sum of variables B and C in floating point addition with variable A; assigns sum to variable X.

FCMP Function

The FCMP (floating point compare) function compares two floating point numbers.

Syntax

FCMP(fx,fy)

fx valid floating point number to be compared

fy valid floating point number to be compared

Description

A standard or string number must be converted to floating point before it is used in the FCMP function. The FFLT function is provided to convert a number to floating point format. The FFIX function is provided to convert a floating point number to a string number. (Please refer to the FFLT and FFIX functions listed alphabetically in this chapter.)

If either fx or fy contains a non-floating point value, an error message is generated.

The result of the FCMP function is a number. If fx is less than fy, the result is -1. If fx and fy are equal, the result is 0. If fx is greater than fy, the result is 1.

The function can be used in any expression where a number or string would be valid.

```
IF FCMP(FX,FY) = 0 THEN GOTO 100
```

The result of the comparison determines whether program execution branches to statement 100 or continues in sequence.

```
ON 2+FCMP(VAL1,VAL2) GOTO 10,110,120
```

The result of the comparison creates an index of 1,2, or 3 for the ON GOTO statement.

FDIV Function

The FDIV (floating point division) function divides the first floating point number by the second and returns the result as a floating point number.

Syntax

FDIV(fx,fy)

fx valid floating point number to be divided

fy valid floating point number to be used as divisor

Description

A standard or string number must be converted to floating point before it is used in the FDIV function. The FFLT function is provided to convert a number to floating point format. The FFIX function is provided to convert a floating point number to a string number. (Please refer to the FFLT and FFIX functions listed alphabetically in this chapter.)

If either fx or fy contains a non-floating point value, an error message is generated.

The result of the FDIV function is a floating point number. Thus, the function can be used in any expression where a floating point number would be valid.

`V = FDIV(D,T)`	Assigns result of variable D divided by T to variable V.
`A = FDIV("1030476E-6",B)`	Assigns to variable A the result of dividing floating point value of constant 1.030476 (1030476E-6) by variable B.
`X = FDIV(A, FDIV(B,C))`	Uses floating point result of variable B divided by variable C in floating point division with variable A; assigns result to variable X.

FFIX Function

The FFIX (fix a floating point number) function returns the value of a floating point number as a string number.

Syntax

FFIX(fx{,n})

fx any valid floating point number

n any valid integer number; n is used to specify the maximum number of digits to the right of the decimal point to be returned in the result. If n is omitted or is negative, the result contains all digits to the right of the decimal point that are in fx. If n is less than the number of digits to the right of the decimal in fx, the unused digits are truncated. The result is not rounded.

Description

This function is intended to be used after floating point arithmetic functions: FADD, FSUB, FMUL, FDIV. (Please refer to these functions listed alphabetically in this chapter.)

The result of the FFIX function is a string number. The function can be used in any expression where a string or string number would be valid.

`PRINT FFIX(FADD(FX,FY))`	The result of the floating point addition is converted to a string number and printed.
`A=FFIX(FMUL("4E-6",FFLT(B)),4)`	The variable B is converted into a floating point number for the floating point multiplication operation; the result is converted to a string number with a maximum of 4 decimal places and assigned to variable A.

FFLT Function

The FFLT (float a number or string number) function converts a number or string number into a floating point number.

Syntax

FFLT(x)

x any number or string number; if the number contains more than 13 significant digits, it is truncated to 13 significant digits

Description

This function is intended to be used before floating point arithmetic functions: FADD, FSUB, FMUL, FDIV. (Please refer to these functions listed alphabetically in this chapter.)

The result of the FFLT function is a floating point number. Thus, it can be used in any expression where a floating point number or a string would be valid.

This function must precede floating point arithmetic performed on a standard number or string number.

`X=FFLT(Y)`	The floating point value of the number Y is assigned to X.
`A=FMUL(FFLT(X),FFLT(Y))`	The variables X and Y are converted to floating point, then used in a floating point multiplication function; the result is assigned to variable A.
`FLOAT.PI=FFLT("3.14159")`	The constant value is converted to floating point and assigned to FLOAT.PI.

FIELD Function

The FIELD function returns one or more substrings from a string delimited by a specified character.

Syntax

FIELD(string.expr,delimiter,start.field{,no.fields})

string.expr	string to be used
delimiter	delimiting character of a field; if delimiter evaluates to more than one character, only the first character is used
start.field	starting occurrence of field to return; if start.field is 1 and the delimiter is null ("") or is not in the string, the entire string is returned. If start.field is greater than 1 and the delimiter is null ("") or if start.field is greater than the number of fields left in the string, a null value is returned. If start.field is less than 1, 1 is used as the value.
no.fields	number of fields to return; if omitted or less than 1, 1 is assumed. If no.fields is greater than the number of fields remaining in the string, the remainder of the string is returned.

All parameters can be literals or expressions.

Description

The boundary delimiter characters of the returned substring are not returned.

The first delimited substring has only an end delimiter character. For example, if the string is A*B*C and the delimiter is *, the first field is A; if the string is *A*B*C and the delimiter is *, the first field is null ("").

The last delimited substring has only a beginning delimiter character. For example, if the string is A*B*C and the delimiter is *, the last field is C; if the string is A*B*C* and the delimiter is *, the last field is null ("").

The multiple extract form of the FIELD function is equivalent to using a G processing code in the ICONV or OCONV functions, where start.field is one greater than the field to skip used in the G processing code.

The COL1() and COL2() functions can be used in conjunction with the FIELD function.

```
X = 'A*B*C*D*E*F'
Y = FIELD(X,'*',2,3)
```

The multiple fields parameter causes three fields to be extracted from X starting with the second field and returns the result , 'B*C*D', to Y; the field delimiter is an asterisk (*). This is equivalent to the OCONV function: Z = OCONV(X,'G1*3')

```
T = "1234A6789A9876A"
G = FIELD(T,"A",1)
```

Assigns the string value "1234" to variable G.

```
T = "1234A6789A9876A"
G = FIELD(T,"A",3)
```

Assigns the string value "9876" to variable G.

```
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.

FMT Function

The FMT function formats a string to a pattern; it can be used as an alternative to a format mask in an assignment statement. (Format masks are described in Chapter 2, Working with Data.)

Syntax

FMT(expression,format.mask)

expression variable to be formatted

format.mask format mask; may contain any valid set for mask codes

```
Y = 12000
X = FMT(Y,'R2(#10)')
```

result:

```
X = 12000.00
```

This function yields the same results as the following assignment statement using a format operator:

```
X = Y 'R2(#10)'
```

FMUL Function

The FMUL (floating point multiplication) function multiplies two floating point numbers and returns the result as a floating point number.

Syntax

FMUL(fx,fy)

fx valid floating point number to be multiplied

fy valid floating point number to be used as multiplier

Description

A standard or string number must be converted to floating point before it is used in the FMUL function. The FFLT function is provided to convert a number to floating point format. The FFIX function is provided to convert a floating point number to a string number. (Please refer to the FFLT and FFIX functions listed alphabetically in this chapter.)

If either fx or fy contains a non-floating point value, an error message is generated.

The result of the FMUL function is a floating point number. Thus, the function can be used in any expression where a floating point number or a string would be valid.

`PAY=FMUL(HOURS,RATE)`	The variable PAY is assigned the product of HOURS times RATE.
`A=FMUL("1030476E-6",B)`	The floating point constant 1030476E-6 (1.030476) is multiplied by variable B and the result is assigned to variable A
`X=FMUL(A, FMUL(B,C))`	The product of variables B and C is multiplied with variable A; the result is assigned to X.

FOOTING Statement

The FOOTING statement causes the specified text string to be printed at the bottom of each page of output.

Syntax

FOOTING "{text } {'options'} {text } {'options'} ..."

text — text to printed as part of footing

options — special instructions to print processor; must be enclosed in single quotes (multiple options may be enclosed in one set of quotes). The following options are available:

C	Centers the line
Cn	Centers with specified line length
D	Inserts current date in dd mon yyyy format
L	Inserts carriage return/line feed; prints blank line
P	Inserts current page number, right-justified in a field of 4 blanks
PN	Inserts current page number, left-justified with no blanks
Pn	Inserts current page number, left-justified in a field of n blanks
T	Inserts current time and date

Description

The FOOTING statement is printed only if there is also a HEADING statement. The specified footing is automatically printed at the bottom of each page.

The footing may be changed at any time in the BASIC program by another FOOTING statement; this change takes effect on the current page, unless the last line on the page has already been printed. Footings and headings can be turned off by the PAGING OFF statement, which is described alphabetically in this chapter.

The first FOOTING or HEADING statement executed in a program initializes the page parameters. Page numbers are assigned in ascending order starting with page 1.

The FOOTING statement affects only print file zero, the default output device.

`FOOTING "'L'TIME & DATE: 'T'"`	A blank line is printed, followed by the text "TIME & DATE:" and the current time and date.
`FOOTING "'LC60'PAGE 'P'"`	A blank line is printed, then on the next line, the text "PAGE" and the current page number are centered within a page width of 60.
`FOOTING "'LTPL'"`	A blank line is printed, followed by the current time, date, and page number on the next line, followed by another blank line.

FOR/NEXT Statement

A FOR/NEXT loop causes execution of a set of statements for successive values of a specified variable until a specified limit is reached. The FOR statement is used to specify the beginning point of a program loop; the NEXT statement specifies the ending point of the loop.

Syntax

```
FOR variable = expr1 TO expr2 {STEP expr3} {WHILE expr4}
FOR variable = expr1 TO expr2 {STEP expr3} {UNTIL expr5}
  .
  .
NEXT {variable}
```

variable contains value to control looping

expr1 initial value for variable

expr2 limit value; when the limit value is exceeded, program control proceeds to the statement after the NEXT statement; expr2 is evaluated on each iteration of the loop

expr3 increment value to be added to the value of the variable at the end of each pass through the loop; if the STEP phrase is absent, the increment value is assumed to be +1

expr4 test for end of loop; if it evaluates to false (zero), program control passes to the statement immediately following the accompanying NEXT statement; if it evaluates to true (non-zero) the loop repeats; expr4 is evaluated for each iteration of the loop

expr5 test for end of loop; if it evaluates to true (non-zero), program control passes to the statement immediately following the accompanying NEXT statement; if it evaluates to false (zero) the loop repeats; expr5 is evaluated for each iteration of the loop

Only one of the optional condition clauses, WHILE or UNTIL, may be used in a FOR statement.

Description

A loop is a portion of a program written in such a way that it executes repeatedly until some test condition is met. FOR/NEXT loops may be

"nested"; a nested loop is a loop which is wholly contained within another loop. For example, the following statements illustrate a two-level nested loop:

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

The inner loop is executed ten times for each of ten passes through the outer loop; that is, the statement PRINT B(I,J) is executed 100 times, causing matrix B to 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 may be nested up to 44 levels. However, a nested loop must be completely contained within the range of the outer loop; that is, the ranges of the loops may not cross.

```
FOR A=1 TO X
 .
 .
NEXT
```
Limiting value is current value of expression X; increment value is +1.

```
FOR K=10 TO 1 STEP -1
 .
 .
NEXT K
```
Increment value is -1; variable K decrements by -1 for each of 10 passes through the loop.

```
ST="X"
FOR B=1 TO 10 UNTIL ST="XXXXX"
  ST := "X"
NEXT B
```
Loop execute four times: an "X" is concatenated to the string value of variable ST until the string equals "XXXXX".

```
A=0
FOR J=1 TO 10 WHILE A<25
  A=A+1
NEXT J
```
Loop executes ten times: variable J reaches 10 before variable A reaches 25.

FSUB Function

The FSUB (floating point subtraction) function subtracts the second floating point number from the first floating point number and returns the result as a floating point number.

Syntax

FSUB(fx,fy)

fx valid floating point number

fy valid floating point number to subtract from fx

Description

A standard or string number must be converted to floating point before it is used in the FSUB function. The FFLT function is provided to convert a number to floating point format. The FFIX function is provided to convert a floating point number to a string number. (Please refer to the FFLT and FFIX functions listed alphabetically in this chapter.)

If either fx or fy contains a non-floating point value, an error message is generated.

The result of the FSUB function is a floating point number. Thus, the function can be used in any expression where a floating point number would be valid.

`TOTAL=FSUB(SUBTOT1,SUBTOT2)`	Assigns difference of variables SUBTOT1 and SUBTOT2 to variable TOTAL.
`A=FSUB("1030476E-6",B)`	Assigns to variable A the difference of floating point constant 1030476E-6 (1.030476) and variable B.
`X=FSUB(A, FSUB(B,C))`	Uses the difference of variable B and C in floating point subtraction with variable A; the result is assigned to X.

GET Statement

The GET statement retrieves data from either the program argument list or the system message buffer.

Syntax

GET(ARG.{, arg.no}) var {THEN stmts} {ELSE stmts}
GET(MSG.{, arg.no}) var {THEN stmts} {ELSE stmts}

ARG.	retrieves one argument, if any, from list of arguments specified after the program name in the TCL command that invoked the program; any string preceded by a space is considered an argument
MSG.	returns next message identifier and related parameters, if any, placed in message buffer by the last EXECUTE statement or by a PUT statement in an EXECUTEd program
arg.no	integer that specifies the position of the element in the list to retrieve. If arg.no is zero or is not present, the next element on the list is returned; if this is the first GET statement executed, the first element on the list is returned.
var	variable in which returned element is placed
THEN stmts	statements to execute if element is returned to var
ELSE stmts	statements to execute if no element is present in specified position

Description

One or more GET(MSG.) statements can be used to retrieve the system messages generated by a program invoked via an EXECUTE statement. Only the ERRMSG item.ids and parameters are copied to MSG.

The message is returned in the following format:

msg.id]parm1]parm2]...]parmn

where

msg.id	message identifier (ERRMSG item.id)
]	value mark
parm1...parmn	values associated with the message

The list of system messages is reset to null just prior to the execution of an EXECUTE statement.

Note: *The redirection clause RETURNING used with EXECUTE returns all ERRMSG item.ids and parameters in one variable and has no effect on the values returned by MSG.*

The EOF function is available to test for end-of-argument or end-of-message list. (Refer to the EOF function listed alphabetically in this chapter.)

The GET(ARG.) statement retrieves characters specified after the program name in the TCL command that invoked the program. The command that invoked the program could be a RUN statement, or if the program is cataloged, just the program name.

Note: *ARG. and MSG. are predefined keywords with special meaning in the GET statement and should not be used as ordinary variables in other statements.*

```
EXECUTE 'PROG1 ':A
GET(ARG.) P1 ELSE P1 = 0
GET(ARG.) P2 ELSE P2 = 0
```

The first GET statement retrieves the first argument on the list; the second retrieves the next argument.

```
GET(MSG.,1) ERR1 ELSE GOTO START
PRINT ERR1; STOP
```

The GET statement retrieves the first message; if found, it is printed and the program terminates. If there are no system messages, the program branches to START.

```
GET(MSG.) MSG
IF MSG <1,1> = "B10" THEN
 GOTO UD
 END
```

The GET statement retrieves the first message; the IF statement tests the first value and branches to UD if the message-id is "B10".

GOSUB Statement

The GOSUB statement transfers control to a local subroutine.

Syntax

GOSUB label

label — label of statement to which control is transferred; execution proceeds from that statement until a RETURN or RETURN TO statement is encountered, which returns control to the statement following the GOSUB statement or specified line number

Description

A local subroutine is a subroutine that is contained within the program that calls it. The GOSUB statement is always used with a RETURN {TO} statement.

If a statement does not exist with the specified statement-label, an error message is printed at compile time.

For information on a related statement, see the ON GOSUB statement listed alphabetically in this chapter.

```
GOSUB FIRST
  .
STOP
FIRST:
  .
  .
RETURN
```

Transfers control to subroutine FIRST. After the subroutine's RETURN statement is encountered, control returns to the statement after the GOSUB statement.

GOTO Statement

The GOTO statement unconditionally transfers program control to a specified statement-label within the BASIC program.

Syntax

GO{TO} label

label label of statement to which control is transferred

Description

Control may be transferred to statements following the GOTO statement, as well as to statements preceding the GOTO statement.

If a statement does not exist with the specified statement-label, an error message is printed at compile time.

For information on a related statement, see the ON GOTO statement listed alphabetically in this chapter.

```
100 A=0                    Label 100
      .
      .
200 GOTO 500               Branch to statement-label 500
      .
      .
500 B=A+C                  Label 500
    D=100
   .
  GOTO 100                 Repeat program
```

HEADING Statement

The HEADING statement causes the specified text string to be printed as the next page heading. If the output is directed to the terminal, it also causes the display to pause at the end of each page.

Syntax

HEADING "{text} {'options'} {text} {'options'} ..."

text — text to printed as part of the heading

options — special instructions to print processor; must be enclosed in single quotes (multiple options may be enclosed in one set of quotes). The following options are available:

- C — Centers the line
- Cn — Centers with specified line length
- D — Inserts current date
- L — Inserts carriage return/line feed; prints blank line
- P — Inserts current page number, right-justified in a field of 4 blanks
- PN — Inserts current page number, left-justified with no blanks
- Pn — Inserts current page number, left-justified in a field of n blanks
- T — Inserts current time and date

Description

The first FOOTING or HEADING statement executed in a program initializes the page parameters. The specified head is automatically printed at the top of each page.

The heading may be changed at any time in the BASIC program by another HEADING statement; this change takes effect when the end of the current page is reached. Footings and headings can be turned off by the PAGING OFF statement, which is described alphabetically in this chapter.

Page numbers are assigned in ascending order starting with page 1.

The HEADING statement affects only print file zero, the default output device.

`HEADING "TIME & DATE: 'TL'"`	The text "TIME & DATE:" and the current time and date are printed, followed by a blank line.
`HEADING "'C60'PAGE 'PL'"`	The text "PAGE" followed by the current page number is centered within a page width of 60; a blank line is then printed.
`HEADING "'TPL'":A`	The current time, date, and page number followed by the value of variable A are printed as the heading.

ICONV Function

The ICONV function converts a string according to a specified type of input conversion.

Syntax

ICONV(string,code)

string string value to convert

code input conversion code; the following codes are available for input conversions:

- D convert date to internal format
- G extract group of characters
- L return string length
- MCx mask characters by numeric, alpha, or upper/lower case; or convert hexadecimal to decimal, or decimal to hexadecimal
- ML mask left-justified decimal data
- MP convert integer to packed decimal
- MR mask right-justified decimal data
- MT convert time to internal format
- MX convert hexadecimal to ASCII
- P test pattern match
- R test numeric range
- T convert by table translation. The table file and translation criteria must be given. (Please refer to the section "Defining File Translation" in the *Ultimate Recall and Update User Guide* for details.)

  Note: *This type of conversion is inefficient if several items or attributes will be accessed.*

- U convert by subroutine call to standard user exit (see Appendix E) or user-defined assembly routine. The absolute address of the routine must be given. The value of the string may be a parameter to be passed to the subroutine, or a null string if none is needed. If two or more parameters are to be passed, they must be

compressed into a single string in string and parsed by the called routine. (For details, refer to the *Ultimate Assembly Language Reference Manual.*)

These conversion codes are the same as those used for Recall Conversions and Correlatives. For a detailed treatment of these capabilities, refer to the *Ultimate Recall and Update User Guide.*

Description

Values that cannot be successfully converted cause a null string to be returned as a result. For example, a string that is not a valid date causes ICONV to return a null string when used with the "D" conversion code.

Note: *"MR" and "ML" conversions may also be done with format strings. (For details on format strings, refer to the chapter "Working with Data" in this manual.)*

The following conversion codes used in Recall cannot be used with the ICONV function:

A	arithmetic
B	BASIC subroutine call
C	concatenate
F	function
S	substitution

```
IDATE = ICONV('07-01-89','D')
```

Assigns the string value '7853' (the internal date) to the variable IDATE.

```
CHR=ICONV("41","MX")
```

Assigns the string "A" (corresponding to hexadecimal value "41") to variable CHR.

```
IF ICONV(T,"D")="" THEN GOTO 10
```

If the "D" conversion returns a null, that is, the value of T is not a valid date, branch to 10.

IF Statement

The IF statement provides the conditional execution of a sequence of BASIC statements, or the conditional execution of one of two sequences of statements. The IF statement can be written on a single line, or on multiple lines, as shown in the syntax section. These forms are functionally identical.

Syntax

IF expression {THEN stmts1} {ELSE stmts2}

```
IF expression THEN
  stmts1
  ...
  END ELSE
  stmts2
  ...
  END
```

multi-line THEN/ELSE statements

```
IF expression THEN
  stmts1
  ...
  END
```

multi-line THEN statements

```
IF expression ELSE
  stmts2
  ...
  END
```

multi-line ELSE statements

expression	any legal BASIC expression
THEN stmts1	statement or sequence of statements to be executed if the result of the test condition is true (non-zero)
ELSE stmts2	statement or sequence of statements to be executed if the result of the expression is false (zero)

Either the THEN clause or the ELSE clause may be omitted, but not both.

Description

The sequence of statements in the THEN and ELSE clauses may consist of one or more statements on the same line. If more than one statement is contained on one line, they must be separated by semicolons. Any statements may appear in the THEN and ELSE clauses, including additional IF statements.

If the statement sequences in the THEN and ELSE clauses are placed on multiple program lines, each sequence is terminated by an END.

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

Prints "MATCH" if value of A is the string "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
```

The value of A is printed if Q is numeric and non-zero. If Q=0, the value of B is printed and the program is terminated. If Q is non-numeric, a message is displayed and zero is used as its value.

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

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

```
IF VAL # 0 THEN
  PRINT MESSAGE
  VAL=100
END
```

If the value of VAL is non-zero, the value of MESSAGE is printed, and VAL is assigned a value of 100; otherwise, control passes to the next statement following END.

INDEX Function

The INDEX function searches a string for the occurrence of a substring and returns the starting column position of that substring.

Syntax

INDEX(string,substring,occurrence)

string	string to be examined
substring	substring to search for
occurrence	occurrence of substring within string

Description

The function returns a numeric value which is the starting column position of the substring within the string. If the substring is not found, a value of 0 is returned.

The INDEX function parameters may be any valid expressions.

```
A = INDEX("ABCAB","A",2)
```

Second occurrence of "A" is at column position 4 of "ABCAB", so value of 4 is assigned to variable A.

```
X = "1234ABC"
Y = "ABC"
IF INDEX(X,Y,1)=5 THEN
  GOTO 3
  END
```

"ABC" starts at column position 5 of "1234ABC", so the IF statement transfers control to statement 3.

```
Q = INDEX("PROGRAM","S",5)
```

"S" does not occur in the string "PROGRAM", so value of 0 is assigned to variable Q.

INMAT() Function

The INMAT() function is used to return information about elements in dimensioned arrays and modulos of files that were opened.

Syntax

INMAT()

Description

INMAT() can be used to return values generated by the following statements:

DIM
MATPARSE
MATREAD{U}
OPEN

INMAT() returns the **actual number** of elements in a dimensioned array after a MATREAD{U} or MATPARSE in the following cases:

- when the array is initially dimensioned as zero
- when the array is dimensioned with a variable
- when the array is dimensioned with a literal that is greater than the actual number of attributes read or parsed into it

INMAT() returns **zero** as the number of elements in a dimensioned array after a MATREAD{U} or MATPARSE statement in the following case:

- when the array is dimensioned with a non-zero literal and the number of elements being read or parsed is greater than the dimensioned size of the array

INMAT() also returns **zero** after a DIM statement is executed.

INMAT() returns the **modulo** of the file after an OPEN statement.

Caution! *The value of INMAT() is volatile; if the value is needed, it should be retrieved immediately after the statement that generates it has completed.*

```
DIM A(0)
MATREAD A FROM TESTFILE,'ITEM.1' ELSE STOP
SZ = INMAT()                    The number of elements in the array
                                A is returned in SZ.
```

INPUT Statement

The INPUT statement is used to request data. The cursor position and data format can also be specified.

Syntax

INPUT {@(x,y){:}} var {,len}{:}{format}{_}{THEN/ELSE stmts}
INPUT var ,0

@(x,y) specified cursor position; column x, row y

: used immediately after @(x,y), displays existing value as the default before updating; if no existing value, null is assigned

var variable to which the response is assigned

len maximum length of input; default length is 140 characters

: used after var or len, does not execute carriage return and line feed at end of input

format format string for input validation and output formatting; may contain any Ultimate format string characters; alternatively, it may contain any valid Recall date conversion code. (Input verification is described in this topic. For details on using format strings for output formatting, refer to the chapter "Working with Data" in this manual; for information on Recall date conversion codes, refer to the *Ultimate Recall and Update User Guide*.)

_ if user attempts to enter more than the maximum number of characters, bell is sounded each time a character is entered until RETURN or LINEFEED is pressed

THEN statements to execute after RETURN or LINEFEED is pressed when at least one character has been input by the user

ELSE statements to execute when only RETURN or LINEFEED is pressed by the user

0 0 (zero) accepts single character and no editing is performed; this allows non-printable characters such as <ESC> to be input;

the character is not echoed. This form must be specified with no other parameters. If other parameters are used, 0 is assumed to specify length and not single character input. A variable with value zero is also assumed to specify length and not single character input. (A length of zero does not provide meaningful input and is not recommended.)

Note: *The unprintable characters represented by <CTRL-Q> and <CTRL-S> cannot be accepted if X-ON/X-OFF protocol is enabled. These characters are used to set X-ON and X-OFF for the terminal.*

All options must be in the order shown above.

Description

An INPUT statement causes a prompt character to be printed at the user's terminal. The prompt character can be changed by the PROMPT statement. (For more information about prompt characters, please refer to the PROMPT statement listed alphabetically in this chapter.)

The @(x,y) option allows the input to be placed at a specified cursor position; x and y may be any BASIC expressions. The prompt character is displayed one character prior to the x coordinate. It is recommended that x not be zero.

If @(x,y) is followed by a colon (:), the existing value of var, if any, is displayed, formatted according to format, if present. If there is no existing value, null is assumed. If @(x,y) is followed by a colon (:), and if format is specified as numerics (%), the field is zero-filled before displaying the value of var; if format is specified as blanks (#), the field is dot-filled before displaying the value of var.

If @(x,y) is omitted, the prompt character is displayed at the current cursor position, which may vary, depending on the results of any previous INPUT statement. For accurate cursor positioning, it is recommended that the @(x,y) function be used.

Maximum input is 140 characters unless len specifies a different value. If the user enters the specified number of characters, an automatic RETURN is executed unless the underscore (_) option is present. If the optional _ is used, the operator must physically press RETURN or LINEFEED to indicate end of input. If the optional _ is used and the

operator attempts to enter more than the specified number of characters before pressing RETURN, the bell is sounded and the extra characters are not accepted.

If len is an expression, the expression can use operators with precedence levels 1-4 only, that is, dynamic array extraction, substring, exponentiation, multiplication, division, addition, and subtraction.

If the colon (:) is used after the var or len, the RETURN is inhibited on the screen; the cursor remains positioned after the input data. However, if an error message is printed, either because of PRINTERR or because of input verification, the cursor is moved to the line following the line on which the error message is displayed.

When using a format mask, data is converted on output and input. Thus, if a date is to be input, the default, if any, should be stored in internal format; it is displayed in external format. Input values are converted from external format and stored in var in internal format; they are redisplayed in the external form specified by format.

The THEN and ELSE clauses are both optional; one, both, or none may be used. These clauses may be on a single line or multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.

If either THEN or ELSE is used, a null input (only RETURN) causes var to retain its old value. If no THEN or ELSE is present, null input stores a null string ("") in var.

The following statements and functions may have an effect on INPUT, or may be affected by INPUT:

INPUTCLEAR
INPUTCONTROL
PRINTERR
PROMPT
SYSTEM(11)
SYSTEM(12)

For more information on these features, refer to the description of each listed alphabetically in this chapter.

Input Verification

When there is a format mask in the INPUT statement, input verification is performed on new, non-null input. If there is a value in var that is displayed, no input verification is performed before displaying it.

If format contains a numeric mask or a format mask with percent signs (%), numeric checking is performed. If format contains a length specification (for example, #10), length checking is performed. If format is 'D' or any other valid date format, a date verification is performed. If a numeric mask contains an N (suppress minus signs), negative values are rejected.

Only one type of verification is performed and numeric masks take precedence over format masks. For example, if the statement contains both a decimal digit specification and a length specification (such as R2(#4)), only numeric checking is performed. Other than causing a numeric check, the decimal digit specification has no meaning for input verification.

If numeric checking is specified, the system deletes a leading $ (dollar sign) or - (minus sign), if present. If it then encounters any non-numeric character, other than a thousands separator or decimal point, the input is rejected.

If there is a format mask and no numeric mask, the length of the input is checked. If justification is specified and is a V, the exact number of characters specified in the mask must be entered. If justification is not specified or is an R or L, not more than the specified number of characters can be entered. In addition, if the mask contains a % sign, only a numeric character is accepted in that position. If the mask contains an & (ampersand), only an alphabetic character is accepted in that position. If any other character is specified in the mask, the entered character in that position is checked against the character in the mask. If it does not match, the mask character is ignored and does not count in determining the total length of input.

If format is empty (""), consists of only justification (L, R, or V), or justification and an empty format mask (L()), INPUT expects only a RETURN or LINEFEED; any other entry is rejected and a message, "Entry is too long", is displayed.

If the input data does not conform, a warning message is printed at the bottom of the screen, and the user is reprompted for input. Also, if typeahead is in effect, the typeahead buffer is cleared. Warning messages are automatically cleared when a correctly formatted value is input. When a warning message is printed or cleared, the cursor remains on the line following the error message unless the cursor is repositioned using @(x,y).

The possible warning messages are:

```
Entry must be a NUMBER
Entry must be a DATE
Entry is too long
Entry must be greater than or equal to ZERO
Entry does not match its pattern:  pattern
```

Stacked Input

If stacked input is present, the next line of stacked input is used instead of requesting data from the terminal (see DATA statement).

If both INPUT and DATA statements are used in a program, any INPUT statements intended to process input from an external source (such as a PROC or user input) should appear in the program prior to any DATA statements. If a PROC passes data to the program as an argument via the PROC stack, the input is stacked at the program outset and should be used as soon as possible since the first DATA statement overwrites any unprocessed stacked input.

The following example uses the "This is user input" text string immediately. If the DATA statement had been before the INPUT statement, the values 1 and 2 would overwrite the text string.

```
    MYPROC(proc)                    INPUTTER (BASIC prog)
001 PQ                          001 INPUT STREAM
002 HRUN BP INPUTTER            002 PRINT STREAM
003 STON                        003 DATA '1', '2'
004 H"This is user input"       004 ...
005 P                           005 END
```

`INPUT VAR`	Requests a value for variable VAR.
`INPUT X,3`	Requests input for variable X. When three characters have been entered, an automatic carriage return is executed.
`INPUT X,3_`	Same as above but waits and beeps for RETURN if more than three characters are entered.
`INPUT @(20,10):SOC.SEC 'V(%%%-%%-%%%%)'`	Requests input formatted as for social security numbers; verifies that only numbers are entered; value is stored without dashes. For example, if 423-15-6897 is entered, the variable SOC.SEC contains the value 423156897.
`INPUT @(25,2):INV.DATE 'D'`	Expects input in date format.
`INPUT ZIP,5:`	Requests a value for ZIP; no carriage return/line feed is printed after a value is entered. Program continues if five characters are entered.
`INPUT X THEN GOSUB 20`	Executes subroutine unless only a RETURN pressed.
`INPUT @(35,7):AMOUNT 'R2,'`	Expects input with two values to the right of the decimal point.

`PRINT 'Press <ESC> to return to main menu, any other key to continue: ':`
`INPUT A,0`
`IF SEQ(A)=27 THEN GOTO MAIN`
	If <ESC> (ASCII code 27) is pressed, program goes to routine at MAIN.

INPUTCLEAR Statement

The INPUTCLEAR statement allows users to clear the typeahead buffer for the port on which the BASIC program is running.

Syntax

INPUTCLEAR

Description

When typeahead is enabled (the default case on most systems), users may enter data in anticipation of input requests, before the system has even printed a prompt character. Data typed in ahead is not echoed on the screen until the input request (a BASIC INPUT statement, for example) is executed.

The INPUTCLEAR statement clears any data in the typeahead buffer, forcing new input to be entered for the next input request. This may be useful when errors are discovered and the typeahead data must not be used under the error conditions.

Note: *The typeahead buffer is also cleared by the INPUT statement when it detects an error in the input and by the PRINTERR statement. (Please refer to the INPUT and PRINTERR statements listed alphabetically in this chapter.)*

INPUTCONTROL Statement

The INPUTCONTROL statement is used to control data input for subsequent INPUT statements.

Syntax

INPUTCONTROL {option ... }

option any combination of the following:

CCD{ELETE} Control characters are ignored in subsequent INPUT statements; if FUNC is specified, ignores all control characters except those defined in Table 3-3; if FUNC is not specified, ignores all control characters except <CTRL-X>, BACKSPACE, LINEFEED, RETURN, and TAB. (Control characters have ASCII values less than 32; see Appendix D for a list.)

EDIT Enters word-processing mode to edit an input field if the <F1> key (EDIT) is pressed when input is subsequently requested using an INPUT statement of the form:

```
INPUT @(x,y):var {,length} mask ...
```

Mask must be left-justified. This option is operational only when FUNC is also specified. When this option is used, the <F1> key cannot be recognized at the BASIC program level.

FUNC{KEYS} Enables keys, such as function keys, to be recognized as input terminators; value of terminator key is returned as an ASCII value in SYSTEM(12); Table 3-3 lists the keys and corresponding values returned in SYSTEM(12).

REV{VIDEO} Displays the data entry field in reverse video for subsequent INPUT statements that are of the form:

```
INPUT @(x,y):var {,length} {mask} ...
```

The field is displayed in reverse video unless var has not been assigned a value and no mask is specified.

Description

Any options not explicitly specified in the INPUTCONTROL statement are turned off.

This statement supersedes the U option on the RUN verb used in releases prior to Release 190.

If word-processing mode is entered, the following commands are available during input :

<←>	(cursor left) moves cursor left ; cannot move past first character
<→>	(cursor right) moves cursor right; cannot move past position following last character that was entered
<Backspace>	deletes character to left of cursor, moves cursor to left and shifts remaining characters to left; cannot backspace past first character
<CTRL-X>	redisplays default input, terminates input mode; moves cursor to first position in field
<Delete>	deletes character under cursor, shifts remaining characters to left
<F1>	enters word processing mode in replace mode, moves cursor to first position of field; subsequent use of <F1> toggles between replace and insert mode, cursor remains at current location
<Home>	moves cursor to first position of field
<RETURN>	terminates input
<Tab>	moves cursor right to next word; if at the last word, moves to position following last character

When FUNCKEYS is on, the affected keys are encoded to give BASIC programs a standard view of terminal input, regardless of the specific terminal is being used. The value of the key that was used to terminate input can be retrieved by using SYSTEM(12). In order for FUNCKEYS to encode the value, the affected keys must return the values as defined in the TERMDEF item for the current TERM type.

Table 3-3 lists the terminal keys that are affected by FUNCKEYS and the ASCII value each returns.

Table 3-3 FUNCKEYS Values

Input Key	ASCII value returned in SYSTEM(12)
<Backspace>	8[1]
<Tab>	9
<Linefeed>	10[2]
<Return>	13
<CTRL-X>	24[3]
<Del>	127
<F1> - <F16>	209-224
Shifted <F1> - <F16>	225-240
<Home>	241
<←> (cursor left)	242
<↑> (cursor up)	243
<↓> (cursor down)	244
<→> (cursor right)	245
Unrecognizable key sequence with "lead-in" characters appearing to be Function or Cursor keys.	246

[1]Backspacing at the beginning of a field terminates INPUT. If the terminal's <←> (cursor left) key is the same as the <Backspace> key, the code returned is 242, not 8.

[2]If the terminal's <↓> (cursor down) key is the same as the <Linefeed> key, the code returned is 244, not 10.

[3]Entering <CTRL-X> at the beginning of a field terminates INPUT. At other points it has its standard meaning of clearing input and returning to beginning of field.

INS Statement

The INS statement is used to insert data into a dynamic array.

Syntax

INS expression BEFORE var <attrib.no{,val.no{,subval.no}}>

expression	value to be inserted into the dynamic array; can be an expression using operators with precedence levels 1-4 only (dynamic array extraction, substring, exponentiation, multiplication, division, addition, and subtraction).
var	dynamic array in which to insert expression
attrib.no	position of the attribute to be inserted; if -1, expression is inserted after last attribute, or if last attribute is null, replaces last attribute
val.no	position of the value to be inserted; if -1, expression is inserted after last value in specified attribute, or if last value is null, replaces last value
subval.no	position of the subvalue to be inserted; if -1, expression is inserted after last subvalue in specified attribute and value, or if last subvalue is null, replaces last subvalue

Description

If val.no and subval.no are absent or have a value of 0, the expression is inserted preceding the attribute specified by attrib.no. If val.no is present and subval.no is absent or has a value of 0, the expression is inserted preceding the value specified by val.no. If attrib.no, val.no, and subval.no are all non-zero, the expression is inserted preceding the subvalue specified by subval.no.

This statement performs the same operation as the INSERT function, but in addition, it stores the result back into the source variable. (Refer to the INSERT function listed alphabetically in this chapter.)

```
INS "JOHN" BEFORE NAME<3,1>
```

Inserts the string "JOHN" before the first value of the third attribute of dynamic array NAME.

INSERT Function

The INSERT function returns a dynamic array with a specified attribute, value, or subvalue inserted.

Syntax

INSERT(var,attrib.no{,val.no{,subval.no}};expr)
INSERT(var,attrib.no,val.no,subval.no,expr)

var	dynamic array in which to insert expression
attrib.no	position of the attribute to be inserted; if -1, expression is inserted after last attribute, or if last attribute is null, replaces last attribute
val.no	position of the value to be inserted; if -1, expression is inserted after last value in specified attribute, or if last value is null, replaces last value
subval.no	position of the subvalue to be inserted; if -1, expression is inserted after last subvalue in specified attribute and value, or if last subvalue is null, replaces last subvalue
expr	value to be inserted into the dynamic array; can be an expression using operators with precedence levels 1-4 only (dynamic array extraction, substring, exponentiation, multiplication, division, addition, and subtraction).

In the first form, a semicolon separates the attribute, value, and subvalue numbers from the new data expression (expr); trailing zero value and subvalue numbers and commas are not required.

Description

If val.no and subval.no are absent or have a value of 0, the expression is inserted preceding the attribute specified by attrib.no. If val.no is present and subval.no is absent or has a value of 0, the expression is inserted preceding the value specified by val.no. If attrib.no, val.no, and subval.no are all non-zero, the expression is inserted preceding the subvalue specified by subval.no.

The following example shows two ways to use the INSERT function:

First form: `X = INSERT(X,10;'XXXXX')`
Second form: `X = INSERT(X,10,0,0,'XXXXX')`

In both forms, the value "XXXXX" is inserted as an attribute before attribute 10, creating a new attribute.

Note: *The INS statement may be used to insert an attribute, value, or subvalue into a dynamic array and store the result back into the variable containing the original dynamic array. For more information, please refer to the INS statement, listed alphabetically in this chapter.*

`Y = INSERT(X,3,2;"XYZ")`
or
`Y = INSERT(X,3,2,0,"XYZ")`
	Inserts string value "XYZ" before value 2 of attribute 3 of dynamic array X, creating a new value; assigns the resulting dynamic array to variable Y.

`NEW = "VALUE"`
`T = INSERT(T,9;NEW)`
or
`T = INSERT(T,9,0,0,NEW)`
	Inserts string value "VALUE" before attribute 9 of dynamic array T, creating a new attribute.

`A = "123456789"`
`B = INSERT(B,3,-1;A)`
or
`B = INSERT(B,3,-1,0,A)`
	Inserts the value "123456789" after the last value of attribute 3 of dynamic array B.

`Z = INSERT(W,5,2,1;"B")`
or
`Z = INSERT(W,5,2,1,"B")`
	Inserts the string value "B" before subvalue 1 of value 2 of attribute 5 in dynamic array W, thus creating a new subvalue, and assigns the resulting dynamic array to variable Z.

INT Function

The INT function returns the integer portion of a value; any fractional portion is truncated.

Syntax

INT(expr)

expr value from which to extract the integer portion

```
A = 3.55
B = 3.6
C = INT(A)          Assigns the value 3 to variable C.
D = INT(B)          Assigns the value 3 to variable D.
E = INT(A+B)        Assigns the value 7 to variable E.

J = INT(5/3)        Assigns the value 1 to variable J.
```

LEN Function

The LEN function returns the length of a string.

Syntax

LEN(expr)

expr value whose length is to be determined

Description

The LEN function returns the length of the string specified by the expression in number of characters.

```
Q = LEN("123")          Assigns the value 3 to variable Q.

X = "123"
Y = "ABC"
Z = LEN(X CAT Y)        Assigns the value 6 to variable Z.
```

LET Statement

The LET statement is an optional part of the = (assignment) statement.

Syntax

{LET} variable = expression

variable — variable to be assigned a value

expression — value to assign; may be a literal or variable

Description

For more information on assigning values, please see the = (Assignment) statement, listed at the beginning of this chapter.

```
LET A = 'HELLO'
A = 'HELLO'
```

These two statements are equivalent.

LN Function

The LN function returns the natural logarithm of a number.

Syntax

LN(expression)

expression — any numeric expression; if less than or equal to zero, the LN function returns a value of zero

Description

The LN (natural logarithm) function generates the natural (base e) logarithm of the expression. The LN function is the inverse of the EXP (exponent) function.

```
PRINT LN(10)                    Prints 2.30258509299

A = LN(22026.5)
PRINT A                         Prints 10.00000155291
```

LOCATE Statement

The LOCATE statement is used to find the position of an attribute, a value, or a subvalue within a dynamic array.

Syntax

LOCATE(expr, item{,attrib.no{,val.no}}; var {; seq}){THEN stmts} {ELSE stmts}

LOCATE expr IN item{<attrib.no{,val.no}>}{, start}{BY seq} SETTING var {THEN stmts} {ELSE stmts}

expr	element to be located in dynamic array
item	dynamic array
var	variable into which the position of expr1 is to be stored; if expr1 is not located, var contains the position in the array where the element could be inserted
attrib.no	limits search to specified attribute for location of value; location in var is location of value
val.no	limits search to specified value within specified attribute for location of subvalue; location in var is location of subvalue
seq	specifies order in which item is sorted; can be "A", ascending sequence, or "D", descending sequence; any other values are ignored. The second character of seq, if present, determines the justification; "R" indicates right justification; any other value, including null, indicates left justification.
start	starting position for the search; may be attribute, value, or subvalue number; if start is not present, the default of 1 is assumed
THEN stmts	statements to execute if the element is located
ELSE stmts	statement to execute if the element is not located

Note: *The first form starts the search at the beginning of the dynamic array, while the second form starts the search at a specified attribute, value, or subvalue.*

Description

If neither attrib.no nor val.no is specified, LOCATE searches for an entire attribute and the position returned in var is the location of the attribute.

If seq is specified and the expression is not found, the number returned in var is the position where the expression would be had it been found. If seq is not specified and the expression is not found, the number returned in var is the position following the last element searched.

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

LOCATE eliminates the need for a loop that specifically extracts and tests the attribute, then takes one of two alternative paths before the next item can be searched.

The position returned in var may be used in an INSERT function or INS statement to place the sought element into its proper location. For example, if a program needs to locate or insert data in an item, one statement can perform both the LOCATE and INSERT operations:

```
LOCATE('D',ITEM,4;VAR) ELSE
   ITEM = INSERT(ITEM,4,VAR,0,'D')
   END

LOCATE 'D' IN ITEM<4> SETTING VAR ELSE
   INS 'D' BEFORE ITEM<4,VAR>
   END
```

If the string 'D' is not found in attribute 4 of ITEM, it is inserted as a value at the end of the attribute.

```
LOCATE('55',ITEM,3,1;VAR;'AR') ELSE
  ITEM = INSERT(ITEM,3,1,VAR,'55')
  END
```

All subvalues in third attribute, first value, of dynamic array 'ITEM' are searched for the numeric literal '55'; if the numeric is found, its position is returned in VAR; if it is not found, the position in which it would have been found, using an ascending, right-justified sort, is returned in VAR.

```
LOCATE(STR,REC;VAR) THEN
  NAME=REC<VAR>
  END ELSE
  NAME='INVALID'
  END
PRINT NAME
```

The element STR is sought in item REC; the resulting index is in VAR. Depending on the result, the variable NAME is assigned and printed.

```
LOCATE "JOHN" IN NLIST<3> SETTING X ELSE
  INS "JOHN" BEFORE NLIST<3,X>
  END
```

If "JOHN" is not found, it is inserted at the position returned in X.

```
LOCATE PRODGRP IN RES<1>,1 BY 'AL' SETTING POS ELSE
  INS PRODGRP BEFORE RES<1,POS> ;* product group
  INS ''      BEFORE RES<2,POS> ;* total value
  INS 0       BEFORE RES<3,POS> ;* total quantity
  END
```

LOCK Statement

The LOCK statement provides a file and execution lock capability for BASIC programs.

Syntax

LOCK expression {ELSE stmts}

expression — specifies the execution lock to be set (0-47); if the execution lock is currently unlocked, the statement sets the lock. If a number greater than 47 is specified, it is adjusted by mod 48 to a number less than 47

ELSE stmts — statements to be executed if the specified execution lock is already set by another process; may be placed on the same line separated by semicolons, or may be placed on multiple lines terminated by an END, as in the IF statement

Description

If the specified lock is already set by the current program, the LOCK statement has no effect.

The LOCK statement sets an execution lock that "locks out" other BASIC programs while the lock remains set. When any other BASIC program attempts to set the same lock, that program either executes an alternative set of statements or pauses until the lock is released via an UNLOCK statement by the program which set the lock.

If the specified execution lock has already been set by another program and the ELSE clause is not included in the statement, program execution pauses until the lock is released by the other program. If the specified execution lock has been set and the ELSE clause is included, the statements following the ELSE are executed.

The Ultimate system provides 48 execution locks, numbered from 0 through 47. Execution locks are used as program control devices; they may also be used as file locks to prevent multiple BASIC programs from updating the same files simultaneously.

The following is an example of an execution lock sequence:

Process A sets execution lock 42 before executing code that should not be executed by more than one process at a time. Process B, executing the same program, reaches the "LOCK 42" instruction. If Process A has not yet unlocked 42, Process B cannot proceed to execute that section of code. Process B must wait until Process A has unlocked 42. The code is protected from more than one process executing it at the same time.

When a program terminates execution for any reason, including the BASIC debugger END command, any execution locks still locked for that line are unlocked.

For information on the UNLOCK statement, please refer to the UNLOCK statement listed alphabetically in this chapter.

`LOCK 15 ELSE STOP`	Sets execution lock 15; if lock 15 is already set, program terminates.
`LOCK 2`	Sets execution lock 2; if lock 2 is already set program halts temporarily until lock 2 is released.

`LOCK 10 ELSE`
`  PRINT X`
`  GOTO 5`
`  END`
	Sets execution lock 10; if lock 10 is already set, the value of X is printed and the program branches to statement 5.

LOOP Statement

The LOOP statement constructs a program loop.

Syntax

LOOP {stmts1} {WHILE expression DO {stmts2}} REPEAT
LOOP {stmts1} {UNTIL expression DO {stmts3}} REPEAT

stmts1	statements to be executed before testing for an end of loop condition
expression	expression that evaluates to true (non-zero value) or false (zero)
stmts2	statements to be executed as long as expression is true
stmts3	statements to be executed as long as expression is false

Description

Both the WHILE clause and UNTIL clause are optional. If neither clause is used, an endless loop can be constructed that repeatedly executes all statements (if any) between LOOP and REPEAT unless control is transferred outside the loop by a statement such as EXIT or GOTO. (Please refer to the EXIT and GOTO statements, listed alphabetically in this chapter.)

Loops may be nested 50 levels deep.

Execution of a LOOP statement with a WHILE or UNTIL clause proceeds as follows. First, the statements (if any) following LOOP are executed. Then, the WHILE or UNTIL expression is evaluated. One of the following is then performed, depending on the form used:

- If the WHILE form is used, and if the expression evaluates to true (non-zero), the statements following DO (if any) are executed and program control returns to the beginning of the loop. If the expression evaluates to false (zero), program control passes out of the loop and proceeds with the statement that follows REPEAT.
- If the UNTIL form is used, and if the expression evaluates to false (zero), then the statements following DO (if any) are executed and program control loops back to the beginning of the loop. If the

expression evaluates to true (non-zero), program control passes out of the loop and proceeds with the statement following REPEAT.

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

`J=0`
`LOOP`
`  PRINT J`
`  J=J+1`
`  WHILE J<4 DO REPEAT`
	Loop executes four times; sequential values of variable J from 0 through 3 are printed.

`Q=6`
`LOOP Q=Q-1 WHILE Q DO`
`  PRINT Q`
`  REPEAT`
	Loop executes five times; values of variable Q are printed in decreasing order from 5 to 1.

`Q=6`
`LOOP PRINT Q WHILE Q DO`
`  Q=Q-1 REPEAT`
	Loop executes seven times; values of variable Q are printed in decreasing order from 6 to 0.

`B=1`
`LOOP UNTIL B=6 DO`
`  B=B+1`
`  PRINT B`
`REPEAT`
	Loop executes five times; sequential values of variable B from 2 through 6 are printed.

MAT = Statement

The MAT = statement is used to assign a constant value to each element in an array, or to copy one array to another.

Syntax

MAT var1 = expression
MAT var1 = MAT var2

var1	array to which values are to be assigned or copied; must have been previously dimensioned via a COMMON or DIM statement
expression	value to assign to each element of the array; may be any legal expression
var2	array from which values are to be copied; must have been previously dimensioned via a COMMON or DIM statement; the number of elements in var1 and var2 must be the same

Description

Arrays are copied in row major order, with the second subscript (column) varying first. The first element of the array var2 becomes the first element of the array var1, the second element of var2 becomes the second element of var1, and so forth. Consider the following 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`

The above program dimensions two arrays with ten elements, assigns the numbers 1 through 10 to array Y elements, then copies array Y to array X, giving the array elements the indicated values.

`MAT TABLE=1`	Assigns a value of 1 to each element of array TABLE.
`MAT XYZ=A+B/C`	Assigns the expression value to each element of array XYZ.

`DIM A(20), B(20)`
`  .`
`  .`
`MAT A = MAT B`
	Dimensions two arrays of equal length, and copies to elements of A the values of corresponding elements of B.

`DIM TAB1(10,10), TAB2(50,2)`
`  .`
`  .`
`MAT TAB1 = MAT TAB2`
	Dimensions two arrays of the same number of elements (100), and copies TAB2 values to TAB1 in row major order.

MATCHFIELD Function

The MATCHFIELD function performs a MATCH operation, and if the string matches the specified pattern, MATCHFIELD returns a portion of the string.

Syntax

MATCHFIELD(string.expr,pattern,return.field)

string.expr string to be used

pattern valid pattern match (each element in the pattern defines a field in the string); the valid match patterns (where 'n' is an integer) are:

0X	zero or more characters of any type
nX	exactly n characters of any type
{~}0A	zero or more alphabetic characters
{~}nA	exactly n alphabetic characters
{~}0N	zero or more numeric characters
{~}nN	exactly n numeric characters
"text"	literal string enclosed in single or double quotes

return.field number of the field in the string to return

All parameters can be literals or expressions.

Description

The MATCHFIELD function returns the portion (field) in the string that matches the pattern element number specified in the return.field parameter.

The tilde (~) negates the pattern match. For example, 3N matches a pattern of exactly three numeric characters and ~3N matches a string of exactly three non-numeric characters. ~3N does not match a string of three characters containing any numeric character.

A null string matches the following patterns:

```
0A, 0N, 0X, '', and ""
```

If either the 0A or 0N pattern is specified, matching continues until a character is encountered that does not match the pattern (either alphabetic

or numeric). If another pattern follows a 0A or 0N pattern, that pattern is used in the match as soon as a character is encountered that does not match 0A or 0N pattern.

The 0X pattern matches the entire string; if a pattern match is appended to 0X, the match returns false. Also, if the 0X pattern is specified and the return.field is anything other than 1, null is returned.

Alternative patterns may be specified within a single match expression by concatenating the alternatives, separated by a value mark. For example, "3N:VM:"" can be used to match a field that contains either three numbers or a null string.

To result in a match, the value of the string expression must match the entire value of the pattern expression. If the string does not match the pattern or pattern alternatives, then MATCHFIELD returns a null value.

The return.field expression specifies an element in the match pattern and the string.expr. Because the match pattern may be made up of several distinct pattern elements, the return.field is used to return only the part of the string that matches the return.field pattern element.

```
S = 'ABC-123-XYZ'
A = MATCHFIELD(S,'3A-3N-3A',2)
```

The specified five-field pattern is matched and the second field of the matched string ('-') is returned in A.

```
A = 'ABCDEFGHIJKLMNOP123'
M = MATCHFIELD(A,'0A3N',2)
```

The string is compared to the pattern 0A until a non-alphabetic character is encountered; the 3N pattern is then used. The second field of the matched string ('123') is returned.

MATPARSE Statement

The MATPARSE statement is used to copy a dynamic array or other delimited string into a dimensioned array.

Syntax

MATPARSE dim.array FROM string,delimiter

dim.array	previously defined dimensioned array
string	string whose elements are to be copied
delimiter	character that delimits elements in string

Description

If the dimensioned array is defined with a fixed size, MATPARSE places each delimited value in the string into successive elements of the dimensioned array, up to the size of the array. If there are more delimited values in the string than there are elements in the dimensioned array, the extra delimited values are all placed in the last element of the dimensioned array. If there are fewer delimited values in the string than there are elements in the array, the remaining elements are set to null.

If the dimensioned array is defined with a size of zero or with a variable size, MATPARSE places each delimited value in the string into successive elements of the array and redimensions the array to the number of elements in the string. The INMAT() function can be used to determine the size of the dimensioned array. If the number of delimited values in the string is greater than the maximum size of a dimensioned array (currently 3223), the remaining values are placed in the last element in the array. In this case, INMAT() returns a value of zero.

```
SZ = 0
DIM A(SZ)
MATPARSE A FROM DY.ARRAY,@FM
SZ = INMAT()                    Array is initially dimensioned to 0;
                                MATPARSE redimensions it to
                                size of DY.ARRAY.
```

MATREAD{U} Statement

The MATREAD{U} statement reads a file item and assigns the value of each attribute to consecutive dimensioned array elements.

Syntax

MATREAD{U} var FROM {file.var,} item.id {ON ERROR stmts} {LOCKED stmts} THEN/ELSE stmts

U	locks the item lock associated with the item to be accessed. If the item is currently locked by another BASIC program, the read operation cannot be performed until the item is released, written, or deleted by the user that locked it. The item does not have to exist in order for MATREADU to lock it; in this case, MATREADU executes the ELSE statements, but still locks the associated item lock. (The letter U is appended to the statement name to imply **update**, not **unlock**.)
var	dimensioned array into which the item is read
file.var	variable to which file was previously OPENed; if the file.var is omitted, the internal default file variable is used (that is, the file most recently opened without a file variable); if the specified file has not been opened prior to the execution of the MATREAD statement, the program aborts
item.id	name of item to be accessed
ON ERROR stmts	statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be read due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when reading local files.

The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.

LOCKED stmts — statements to execute if the MATREAD statement is unable to lock the item because another program has already locked it; statements may be on a single line separated by semicolons or on multiple lines terminated by END, as in the multi-line IF statement

THEN stmts — statements to be executed after item is successfully read into array; the THEN statements may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multiple line IF statement

ELSE stmts — statements to be executed if the specified item does not exist; in this case, the contents of the array remain unchanged; the ELSE statements may appear on one line separated by semicolons or on multiple lines terminated by an END, as in the multiple line IF statement.

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

Description

The MATREAD statement reads the file item specified by the item.id expression and assigns the value of each attribute to consecutive elements of the dimensioned array specified by variable var.

If the dimensioned array is defined with a size of zero or with a variable size, MATREAD reads each attribute from the item into one element of the array, redimensioning the array if the number of attributes is different from the dimensioned size of the array. INMAT() returns the number of elements in the array. If the number of attributes in the item is greater than the maximum size of a dimensioned array (currently 3223), the remaining attributes are placed in the last element in the array. In this case, INMAT() returns a value of zero.

If the dimensioned array is defined with a fixed size, and if the number of attributes in the item is less than the DIMensioned size of the array, the trailing elements are assigned a null string. If the number of attributes in the item exceeds the DIMensioned size of the array, the remaining attributes, separated by attribute marks, are assigned to the last element of the array.

UltiNet Considerations

The ON ERROR clause allows the program to retrieve the UltiNet error number and take appropriate action. Such action could, for instance, include printing the associated message text via a PUT statement or STOP statement, and resuming or terminating program execution.

If a remote file cannot be read due to network errors and no ON ERROR clause is present, the program terminates with an error message.

Item Locks

Item locks can be used to prevent updating an item by two or more programs simultaneously while still allowing multiple programs to access the file.

Item locks are assigned based on the group of the file which contains (or would contain) the item and a hash value derived from the item.id. Items in different groups (in the same file or in different files) are never assigned the same item lock, but it is possible for more than one item in the same group to hash to, and be assigned, the same item lock.

If an item is currently unlocked, setting a corresponding item lock prevents access to the item, and any other items in the same group with the same item lock hash value, by other BASIC programs using the MATREADU, READU, or READVU statements. The program setting the lock, however, is allowed to lock other items in the same group with the same hash value using these statements.

An item is unlocked when it, or any other item sharing the same item lock, is updated by a WRITE, WRITEV, MATWRITE, or DELETE statement, or when it is unlocked by a RELEASE statement, or when the BASIC program is terminated. An item can be updated without unlocking it by using the WRITEU, WRITEVU, or the MATWRITEU statement.

There is a maximum number of item locks that may be locked at any one time. This number may vary from release to release. If a program

attempts to lock an item when all item locks are already set, it is suspended until a lock is unlocked.

Note: *Locked items can still be retrieved by the READ, READV, and MATREAD statements and by other system software, such as Recall, that do not pay attention to item locks.*

For information on read locks set by the system, see the section in Chapter 5 called "Read Locks."

```
DIM ITEM(20)
OPEN 'LOG' TO F1 ELSE STOP
MATREAD ITEM FROM F1, 'TEST' ELSE STOP
```

Reads the item named TEST from the data file named LOG and assigns the string value of each attribute to consecutive elements of array ITEM, starting with the first element.

```
MATREAD ITEM FROM F1, 'TEST' ON ERROR
  ERRNUM=SYSTEM(0)
  GOSUB PROCESSERR
  GOTO TOP
END ELSE STOP
```

Reads as above; if a network error occurs, retrieves error number and performs local subroutine on UltiNet error number.

```
MATREADU T FROM XM, "N4" ELSE NULL
```

The item is locked regardless of whether it exists or not.

```
MATREADU ITEM FROM ID LOCKED GOTO 900 ELSE NULL
```

If the item is currently locked, the program branches to label 900.

```
MATREADU ITEM FROM ID ON ERROR GOTO PROCERR ELSE NULL
```

If the item cannot be read due to a network error, the program branches to local subroutine PROCERR for processing the UltiNet error number.

MATWRITE{U} Statement

The MATWRITE{U} statement writes an item from a dimensioned array, and assigns the value of each element to consecutive attributes.

Syntax

MATWRITE{U} var ON {file.var,} item.id {ON ERROR stmts}

U	specifies update mode; does not unlock an item after completing the write operation; if the item is not locked before the MATWRITEU statement is executed, it is locked afterwards. (The letter U is appended to the statement name to imply **update**, not **unlock**.)
var	dimensioned array that contains information to be written
file.var	variable to which file was previously OPENed; if the file.var is omitted, the internal default file variable is used (that is, the file most recently opened without a file variable).
item.id	item to be written; any attributes currently in item are replaced with the string value of the consecutive elements of var. If the item does not exist, a new item is created.
ON ERROR stmts	statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be cleared due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when writing local files. The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.

Description

When the specified name of the item to be written is the same as the name of an existing item on the file, the existing item is overwritten.

Each element in the array becomes an attribute in the item being written, except that null attributes at the end of the item are deleted. If an element of the array contains strings delimited by attribute marks, each delimited string becomes an attribute in the item.

MATWRITE assigns a value of 0 to all attributes whose corresponding array elements have not been assigned a value. A message is displayed if zero is assigned.

If the U option is not specified, the MATWRITE statement clears any item lock associated with the item being written. Item locks may be set with the MATREADU, READU, and READVU statements to prevent simultaneous updates of the same item by more than one program. (For more information on item locks, please refer to the MATREADU, READU, and READVU statements, listed alphabetically in this chapter.)

The U option of the MATWRITE statement is intended primarily for master file updates when several transactions are being processed and an update of the master item is made following each transaction update.

Note: *The RELEASE statement can be also used to unlock an item. (Please refer to the RELEASE statement, listed alphabetically in this chapter.)*

The ON ERROR clause allows the program to retrieve the UltiNet error number and take appropriate action. Such action could, for instance, include printing the associated message text via a PUT statement or STOP statement, and resuming or terminating program execution.

If a remote file cannot be written due to network errors and no ON ERROR clause is present, the program terminates with an error message.

```
DIM ITEM(10)
OPEN 'TEST' TO F1 ELSE STOP
FOR I=1 TO 10
ITEM(I)=I
NEXT I
MATWRITE ITEM ON F1,"JUNK"
```

Writes an item named JUNK in the file named TEST. The item written will contain ten attributes whose string values are 1 through 10.

```
MATWRITE ITEM ON F1,"JUNK" ON ERROR
  ERRNUM=SYSTEM(0)
  GOSUB PROCESSERR
  GOTO TOP
END
```

Writes as above, or retrieves error number and performs local subroutine on UltiNet error number.

```
MATWRITEU ARRAY ON FILE.NAME,ID
```

Replaces the attributes of the item specified by ID (in the file opened and assigned to variable FILE.NAME) with the consecutive elements of ARRAY. Does not unlock the group.

```
MATWRITEU A ON ID ON ERROR GOTO PROCESSERR
```

Writes elements of A to item specified by ID, or branches to process UltiNet error number.

MOD Function

The MOD function generates the remainder of one number divided by another.

Syntax

MOD(dividend,divisor)

dividend	expression to be divided
divisor	expression to divide by

Description

The MOD function is the same as the REM function, listed alphabetically in this chapter.

`Q = MOD(11,3)`	Assigns the value 2 to variable Q.

NEXT Statement

The NEXT statement is used to specify the ending point of a FOR/NEXT program loop. A NEXT statement is always used in conjunction with a FOR statement.

Syntax

NEXT {variable}

variable if used, should be the same as corresponding variable in the FOR statement; however, variable is ignored by compiler

Description

The NEXT statement returns program control to the beginning of the loop after a new value of the variable has been computed.

If variable is not specified, the variable assigned to the last FOR statement without a corresponding NEXT is used.

For more information, see the description of FOR, listed alphabetically in this chapter.

```
FOR A=1 TO 20
  .
  .
NEXT A
```

Loop is executed until value of A is 20.

```
FOR K = 1 TO 10
  FOR L = 1 TO 5
     .
     .
  NEXT
NEXT
```

Nested loops; the first NEXT uses the variable of the last FOR (that is, L) as its control. The second NEXT uses K as its control.

NOT Function

The NOT function returns a value of true (1) if the given expression evaluates to 0 and a value of false (0) if the expression evaluates to a non-zero quantity.

Syntax

NOT(expr)

expr expression to be evaluated; must be numeric

Description

The NOT function returns the logical inverse of the specified expression; it returns a value of true if the expression evaluates to 0, and returns a value of false if the expression evaluates to a non-zero quantity.

`X = A AND NOT(B)`	Assigns the value 1 to variable X if current value of variable A is not 0 and current value of variable B is 0; otherwise assigns a value of 0 to X.
`IF NOT(X) THEN STOP`	Program terminates if current value of variable X is 0.
`PRINT NOT(M) OR NOT(NUM(N))`	Prints a value of 1 if current value of variable M is 0 or current value of variable N is a non-numeric string; otherwise prints a zero.

NULL Statement

The NULL statement specifies a non-operation. It may be used anywhere in the program where a BASIC statement is required.

Syntax

NULL

Description

The NULL statement is used in situations where a BASIC statement is required, but no operation or action is desired.

`INPUT X ELSE NULL`	Assigns an input value to variable X if the value is non-null (that is, if the operator enters more than just a carriage return in response to the INPUT prompt character). If the input value is null, X will retain its old value. Without the ELSE NULL clause, the INPUT statement would assign X a null value if no value (just a <CR>) were entered.
`10 NULL`	This statement does not result in any operation or action; however, since is is preceded by a statement label (10) it may be used as a program entry point for GOTO or GOSUB statements elsewhere in the program.
`READ A FROM "ABC" ELSE NULL`	File item ABC is read and assigned to variable A. If ABC does not exist, no action is taken.

NUM Function

The NUM function returns a value of true (1) if the given expression evaluates to a number or a numeric string; otherwise it returns a value of false (0).

Syntax

NUM(expr)

expr expression to be evaluated

Description

The NUM function tests the given expression for a numeric value. It returns a value of true if the expression evaluates to a number or numeric string, and returns a value of false if the expression evaluates to an alphabetic or other non-numeric string.

The NUM function considers a numeric string to be one of the following:

- a sequence of decimal digits, optionally preceded by a plus or minus sign, and optionally containing a decimal point
- a null string ("")

`A1=NUM(123)`	Assigns a value of 1 to variable A1.
`A2=NUM("123")`	Assigns a value of 1 to variable A2.
`A3=NUM("12C")`	Assigns a value of 0 to variable A3.
`A4=NUM("")`	Assigns a value of 1 to variable A4.

OCONV Function

The OCONV function converts a string according to a specified type of output conversion.

Syntax

OCONV(string,code)

string string to be converted

code output conversion code; the following codes are available for output conversions:

- D convert date to external format
- G extract group of characters
- L return string length
- MCx mask characters by numeric, alpha, or upper/lower case; or convert hexadecimal to decimal or decimal to hexadecimal
- ML mask left-justified decimal data
- MP convert packed decimal to integer
- MR mask right-justified decimal data
- MT convert time to external format
- MX convert ASCII to hexadecimal
- P test pattern match
- R test numeric range
- T convert by table translation. The table file and translation criteria must be given. (Please refer to the section "Defining File Translation" in the *Ultimate Recall and Update User Guide* for details.)

  Note: *This type of conversion is inefficient if several items or attributes will be accessed.*

- U convert by subroutine call to standard user exit (see Appendix E) or user-defined assembly routine. The absolute address of the routine must be given. The value of the string may be a parameter to be passed to the subroutine, or a null string if none is needed. If two or more parameters are to be passed, they must be

compressed into a single string in string and parsed by the called routine. (For details, refer to the *Ultimate Assembly Language Reference Manual.*)

Description

These conversion codes are the same as those used for Recall Conversions and Correlatives. For a detailed treatment of these capabilities, refer to the *Ultimate Recall and Update User Guide.*

The resulting value is always a string value.

Note: *"MR" and "ML" conversions may also be done with format strings. (For details on format strings, refer to the chapter "Working with Data" in this manual.)*

The following conversion codes used in Recall cannot be used with the OCONV function:

A	arithmetic
B	BASIC subroutine call
C	concatenate
F	function
S	substitution

```
COLOR=OCONV("RED^BLUE^WHITE","G1^1")
```
Extracts "BLUE" from the string and assigns it to the variable COLOR.

```
A="7853"
B="D"
XDATE = OCONV(A,B)
```
Assigns the string value "01 JUL 1989" (the external date) to the variable XDATE.

```
TEAMS=OCONV("TEAMS","TGAMES;X;1;1")
```
Reads the first attribute of item "TEAMS" in the file "GAMES".

ON GOSUB Statement

The ON GOSUB statement transfers program control to a local subroutine, based on an index and list of subroutine labels.

Syntax

ON expr GOSUB label.1, label.2, ...

expr — value to use as index into the list of labels; expr is evaluated and truncated to an integer value, if necessary. If the expression evaluates to less than 1 or greater than the number of labels, no action is taken and the statement immediately following the ON GOSUB is executed.

label.n — label of local subroutine

Description

The subroutines specified by the labels in the label list may precede or follow the ON GOSUB statement.

When a RETURN statement is encountered in the subroutine, control returns to the statement following the ON GOSUB.

The ON GOSUB statement may continue on multiple lines; each line except the last must conclude with a comma.

`ON I GOSUB 100,150,200`	Branches to subroutine (SUBROUTINE statement) located at 100, 150, or 200, depending on value of I being 1, 2, or 3, respectively.
`ON CHECK GOSUB ONE, TWO, THREE`	Branches to subroutine located at ONE, TWO, or THREE, depending on value of variable CHECK.

ON GOTO Statement

The ON GOTO statement transfers program control to a statement label within the current program, based on an index and list of statement labels. The program continues from that point.

Syntax

ON expr GOTO label.1, label.2,...

expr — value to use as index into the list of labels; expr is evaluated and truncated to an integer value, if necessary. If the expression evaluates to less than 1 or greater than the number of labels, no action is taken and the statement immediately following the ON GOTO is executed.

label.n — label of statement to which control is transferred

Description

The routines specified by the labels in the label list may precede or follow the ON GOTO statement.

The ON GOTO statement may continue on multiple lines; each line except the last must conclude with a comma.

`ON M GOTO 40, 61, 5, 7`	Transfers control to statement 40, 61, 5, or 7, depending on the value of M being 1, 2, 3, or 4 respectively.

`ON C GOTO ELEMENTARY,`
`  ELEMENTARY,`
`  ADVANCED`
	Transfers control to label ELEMENTARY if C = 1 or 2, or to label ADVANCED if C = 3.

`IF A GE 1 AND A LE 3 THEN`
`  ON A GOTO 110,120,130`
`END`
	The IF statement assures that A is in range for the computed GOTO statement.

OPEN Statement

The OPEN statement is used to select an Ultimate file for subsequent input, output, or update. It can also be used to open a subroutine for subsequent use with the @ form of the CALL statement.

Opening Files

Syntax

OPEN file.name {TO file.var} {ON ERROR stmts}{THEN /ELSE stmts}

file.name — Ultimate file name; must be specified in one of the following formats:

"filename"
"dictname,filename"
"DICT filename"
"DATA filename"
"DATA dictname,filename"

Unless DICT is specified, the data section of the file is opened; the DATA specification is provided for compatibility with previous revisions, but is not required

file.var — variable to which the file is assigned for subsequent reference; if file.var is omitted, the file is opened to the internal default file and subsequent I/O statements not specifying a file variable default to this file

ON ERROR stmts — statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be opened due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when opening local files.

The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.

THEN /ELSE stmts — The THEN statements, if any, are executed if the file is opened successfully; if file does not exist, the ELSE statements, if any, are executed. The statements in the THEN/ELSE clause may be placed on the same line, or may be placed on multiple lines terminated by an END, as in the multi-line IF statement.

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

Description

There is no limit to the number of files that may be open at one time.

```
OPEN 'ABC,X' TO D5 ELSE STOP
```

Opens data section X of file ABC and assigns it to variable D5. If ABC,X does not exist, program terminates.

```
OPEN 'TEST' ELSE
   PRINT ERRTEXT('201','TEST')
   GOTO 60
END
```

Assigns file to internal default file variable. If file does not exist, message 201 from the ERRMSG file is printed and control passes to label 60

```
OPEN 'DICT TRANS' TO DTRANS ELSE STOP
```

Opens DICT of file TRANS and assigns it to variable DTRANS.

```
OPEN 'EMPLOY' TO EMP ON ERROR
  ERRNUM=SYSTEM(0)
  GOSUB PROCESSERR
  GOTO TOP
END ELSE STOP
```

Opens DATA section of the file EMPLOY and assigns it to variable EMP or retrieves error number and performs local subroutine on UltiNet error.

Opening Subroutines

Syntax

OPEN 'SUB{ROUTINE}', '{file.name }sub.name' TO var THEN/ELSE statements

file.name — file that contains the subroutine; if file.name is not specified, the system looks for a cataloged subroutine in the master dictionary of the current user

sub.name — subroutine name

var — variable to be used in CALL @ statement

Description

This statement increases the speed of using indirect calls. The opening of the variable increases the speed of the CALL @ by approximately six times because the system saves the location of the subroutine and does not have to recalculate it each time.

The variable can be declared in a COMMON statement and, if a subroutine is opened to it in one program, the information can be passed to subsequent programs.

```
OPEN 'SUB', 'SUBR.ADD' TO S ELSE STOP 'B25',
"",'SUBR.ADD'
  .
  .
CALL @S(X,Y,Z)

DIM TAX(5)
OPEN 'SUB','TAX.NJ' TO TAX(1) ELSE ...
OPEN 'SUB','TAX.CA' TO TAX(2) ELSE ...
  .
  .
SUB = TAX(STATE)
CALL @SUB(AMT,RESULT)
```

PAGE Statement

The PAGE statement causes the current output device to start a new page. The page number may optionally be reset.

Syntax

PAGE {expr}

expr page number to be used on the new page being started. If a FOOTING that includes page numbering is in effect at the time the page number is changed, the footing is printed on the current page with a page number one less than expr.

Description

The PAGE statement is used to end a page before the maximum number of lines has been reached and automatic paging occurs. (The maximum number of print lines per page is controlled by the current TERM command specifications; for more information, see the TERM command in the *Ultimate System Commands Reference Guide*.)

The most recent FOOTING statement, if any, is used to output a footing on the completed page. The heading specified by the most recent HEADING statement, if any, is printed as a page heading on the new page.

The PAGE statement causes a new page to be started even if no heading or footing has been assigned.

`HEADING "ANNUAL REPORT"`
`FOOTING "XYZ CORPORATION"`
`PAGE`
	The PAGE statement causes both the specified heading and footing to be printed when the paging is executed.

`PAGE 1`	The current footing, if any, is printed on the current page with a page number of 0. A new page is started and the current heading, if any, is printed on the new page with a page number of 1.

PAGING Statement

The PAGING statement turns off heading and footing statements. The output is then produced as one continuous page and there is no longer a pause at the end of a page if the output is to the terminal.

Syntax

PAGING OFF

Description

The PAGING OFF statement affects both HEADING and FOOTING statements. A new HEADING statement or FOOTING statement must be executed in order for the program to resume automatic paging.

The PAGING OFF statement resets the page counter to zero.

```
HEADING "Delivery Summary  'CDLPL'"
  .
  .
PAGING OFF
  .
  .
HEADING "Scheduling Summary  'CDLPL'"
```

PRECISION Statement

The PRECISION statement allows the user to select the degree of precision to which all numeric values are calculated within a given program.

Syntax

PRECISION n

n number of digits in the decimal place; must be in the range 0–9

Description

The default precision value is 4; that is, unless otherwise specified, all numeric values are stored in an internal form with 4 decimal places, and all computations are performed to this degree of precision.

Only one PRECISION statement is allowed in a program. If more than one is encountered, a warning message is printed and the declaration is ignored.

Where external subroutines are used, the mainline program and all external subroutines must have the same PRECISION. If the precision is different between the calling program and the subroutine, a warning message is printed.

Setting a precision of zero implies that all values are treated as integers.

Note: *When a program uses floating point or string arithmetic, the program's PRECISION is ignored by the routines that perform those arithmetic calculations. PRECISION is also ignored by the EXP, LN, and PWR functions. (See Section 2 for an overview of floating point and string arithmetic.)*

`PRECISION 0`
`A = 3`
`B = A/2`
	All numeric values in the program are treated as integers. The value returned for B is 1, not 1.5.

`PRECISION 1`	All numeric values in the program are calculated to one fractional digit.

PRINT Statement

The PRINT statement outputs data to the current output device. The PRINT ON option allows output to multiple print files.

Syntax

PRINT {ON exp} {print.list}

ON expr — specifies print file number to which the print.list is directed; has effect only when PRINTER ON is specified. The print list number may be from 0 to 255, selected arbitrarily by the programmer. The expression may use operators with precedence levels 1-4 only (dynamic array extraction, substring, exponentiation, multiplication, division, addition, and subtraction).

print.list — list of information to display; may consist of a single expression or a series of expressions separated by commas, format strings, and @ functions for cursor control. The print-list may end with a colon; this inhibits the end-of-line carriage return and line feed and keeps the cursor at the current location. The expressions may be any legal BASIC expressions. If the print.list is absent, only a carriage return/line feed is output.

Description

The PRINT statement without the ON option is used to output variable or literal values to the terminal or other output destination.

By default, the output of the PRINT statement is to the terminal. The PRINTER statement may be used to route output from PRINT statements to the printer or other spooled output destination. (Refer to the PRINTER statement, listed alphabetically in this chapter.) The P option on the RUN verb also routes output to the printer or other spooled output destination. The CAPTURING and OUT. clauses of the EXECUTE statement may also redirect print output. To force output to the terminal, use the CRT or DISPLAY statements, which are listed alphabetically in this chapter.

PRINT ON can be used to build several reports at the same time, each having a different number. The contents of all print files used by the program, including print file zero, are output to the printer in sequence.

When the ON expression is omitted, print file zero is used.

Caution! *The HEADING and FOOTING statements affect only print file zero. Pagination must be handled by the program for print files other than zero. Lack of pagination will result in continuous printing across page boundaries.*

If PRINTER ON is not in effect, PRINT ON has no effect and all output is to the terminal as a single report.

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. If printing continues to multiple lines, tab positioning with the comma (,) is handled as if the printing is on one line. The first print element always starts logically at column 1, the next tab is 18 characters from that print position, and so on. If the line length does not divide evenly by 18, the columns in the second line do not line up under columns in the first line.

After the print.list has been printed, a carriage return/line feed is automatically executed. The carriage return/line feed can be suppressed by using the : (concatenation) operator at the end of the PRINT statement, in the form:

```
... expression:
```

If the print.list ends with a colon (:), the next value in the next PRINT statement is printed on the same line at the next character position.

The @ function may be used to position the characters at any point on the terminal. In general, the @ function, other than @(-100) and lower, is not meant for statements that are to be directed to the printer and may cause unexpected results.

`PRINT X`	Outputs the value of X to the terminal (if no PRINTER ON in effect).
`PRINTER ON` `PRINT X`	Outputs the value of X to printer assigned to print file 0.

`PRINTER ON`
`PRINT ON 24 X`
`PRINT ON 10 F1,F2,F3`
`PRINT ON 20 M,N,P`
	Outputs the value of X to print file 24, values of F1 through F3 to print file 10, and M,N,P to print file 20

`PRINT A:B:` `PRINT C:D`	Prints the current values of A, B, C, D contiguously across the page
`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 18 (that is, the next tab position).
`PRINT`	Prints an empty (blank) line.
`PRINT @(-1):"INPUT":`	Clears the screen, then prints the text "INPUT"; does not execute a carriage return or line feed.
`PRINT @(20,3):DESC "L(#20)":OCONV(TOT,"MR2") "R(#12)"`	On the terminal, prints the value of DESC left-justified in a field of 20 blanks, starting at column 20, row 3, followed by the converted value of TOT right-justified in a field of 12 blanks

PRINTER Statement

The PRINTER statement selects either the user's terminal or the printer for subsequent program output.

Syntax

PRINTER ON
PRINTER OFF
PRINTER CLOSE

ON directs program output data specified by subsequent PRINT, HEADING, FOOTING, or PAGE statements to be output to the printer, or other destination as specified by the system command SP-ASSIGN

OFF directs subsequent program output to the user's terminal

CLOSE causes all data currently stored in the intermediate buffer area to be immediately sent to the spooler; has no effect on PRINTER ON or PRINTER OFF status.

Description

Once executed, a PRINTER ON or PRINTER OFF statement remains in effect until a new PRINTER ON or PRINTER OFF statement is executed. If a PRINTER ON statement has not been executed, all output is to the user's terminal, unless the program was initiated by a RUN command with the P option.

When a PRINTER ON statement has been issued, subsequent output data (specified by PRINT, HEADING, FOOTING, or PAGE statements) is not immediately printed on the printer, unless immediate printing is specified in the system command SP-ASSIGN (for more information, see the *Ultimate System Commands Reference Guide.*). Rather, the data is stored in an intermediate buffer area and is automatically printed when the program terminates execution.

A PRINTER CLOSE statement may be used when the user's application requires that the data be printed on the printer prior to program termination. The PRINTER CLOSE statement applies only to output data directed to the printer; output to the terminal is always printed

immediately upon execution of the PRINT, HEADING, FOOTING, or PAGE statements.

`PRINTER ON`
`PRINT A`
`PRINTER OFF`
`PRINT B`
	Causes the value of variable A to be be printed on the printer when the program is finished executing, and the value of variable B to be printed on the terminal.

`PRINTER ON`
`PRINT A`
`PRINTER CLOSE`
`PRINT B`
	Causes the value of variable A to be immediately printed on the printer, and prints the value of variable B on the printer.

PRINTERR Statement

The PRINTERR statement prints a specified message on the bottom line of the terminal screen.

Syntax

PRINTERR msg

msg text to be displayed; may be any valid expression, including a literal string enclosed in quotation marks

Description

The PRINTERR statement is designed as a support function for the INPUT statement. It allows a program to signal an operator with a message relating to the operator's input.

The PRINTERR statement sets a flag so that the next time an INPUT statement is executed the bottom line is cleared. The PRINTERR statement also clears the typeahead buffer on systems with the typeahead feature.

```
10 INPUT @(15,5) A,3
   IF A > 500 THEN
    PRINTERR "Value too large"
    GOTO 10
  END
```

Prints message "Value too large" on the bottom of the screen and sets the flag to clear that line (the message line) on next INPUT statement.

PROCREAD Statement

The PROCREAD statement allows programs executed from PROC to read values in the primary input buffer and store them in a variable.

Syntax

PROCREAD var {THEN/ELSE statements}

var receives the PROC primary input buffer in form of a dynamic array

Description

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

If the program was invoked from a PROC, the PROCREAD statement creates a dynamic array from the PROC primary input buffer and assigns it to var; each attribute in the dynamic array contains one primary input buffer parameter. The statements after THEN, if any, are then executed.

If the program was not invoked from a PROC, the statements after ELSE, if any, are executed, and var retains its original value.

For more information on PROCs, refer to the *Ultimate PROC Reference Guide*.

The THEN clause and ELSE clause may continue on several lines. When multiple lines are used, the clause must end with an END statement, as in the multiple-line IF statement.

```
PROCREAD ITEM ELSE
  PRINT 'Not from PROC'
  GO 100
 END
PRINT ITEM
```

PROC primary input buffer is assigned to variable ITEM. If the program was not executed from PROC, the message is printed and control transfers to label 100. If the program is executed from PROC, then variable ITEM is printed.

```
PROCREAD BUFF ELSE
  BUFF=""
 END
PRINT BUFF
```

BUFF is set to null if program is not executed from PROC. The contents of BUFF are always printed.

```
PROCREAD BUFF ELSE
  PRINT 'ITEM not found'
  STOP
 END
FOR X=1 TO 10
  PRINT BUFF<X>
NEXT X
```

If the program was executed from PROC, BUFF is assigned a multiple parameter primary input buffer and is displayed as an array. If the program was not executed from PROC, the message is printed and the program is terminated.

PROCWRITE Statement

The PROCWRITE statement allows programs executed from PROC to write to the primary input buffer.

Syntax

PROCWRITE expr

expr dynamic array; each attribute becomes one parameter in the primary input buffer

Description

The PROCWRITE command writes the string value of the expression to the PROC primary input buffer. The attribute marks are converted to spaces, the delimiter for parameters in a PROC buffer. (Spaces in any of the elements of the dynamic array are passed to the PROC buffer and are treated as delimiters by PROC.)

This statement is ignored if the program was not executed from a PROC.

`PROCWRITE ITEM`	PROC primary input buffer is assigned the value of ITEM. If the program was not executed from PROC, the statement is ignored.
`PROCWRITE X+Y` `PROCREAD ITEM ELSE STOP`	If the program was executed from PROC, the primary input buffer is assigned the sum value of X and Y. The sum is stored in variable ITEM. If the program was not executed from PROC, the program is terminated.

PROGRAM Statement

The PROGRAM statement may be used to indicate the name of a program.

Syntax

PROG{RAM }{name}

name indicates the name of a program as a comment; this is ignored by the compiler, which uses the item.id of the program as the program name

Description

The PROGRAM statement is required in compile-and-go programs in an account's Master Dictionary and must be the first statement in such a program. The PROGRAM statement is not required in other programs and is ignored if used.

For information on using this feature, see the section in Chapter 1 called "Executing BASIC Source Programs (Compile and Go)."

PROGRAM	Indicates start of program.

PROMPT Statement

The PROMPT statement is used to select the character that is displayed by the INPUT statement to prompt the user. Any character may be selected.

Syntax

PROMPT expr

expr prompt character; if expr consists of more than one character, the first (leftmost) character is used; if expr is the null string (""), no prompt character is used

Description

The default prompt character is the question mark (?); it is used if a PROMPT statement is not specified.

When a PROMPT statement has been executed, it remains in effect until another PROMPT statement is executed or the program is terminated.

`PROMPT ":"`	Specifies that the character : is used as a prompt character for subsequent INPUT statements.
`PROMPT A`	Specifies that the current value of A is to be used as a prompt character.

PUT Statement

The PUT statement places a system message into the output of a program. The message is also placed in the system message buffer and may be passed back to a calling program.

Syntax

PUT(MSG.) expr1, {expr2, ...}

expr1 specifies the system message identifier (ERRMSG item.id)

exprn parameters associated with the message, if any

Description

The PUT statement allows a program to output messages and continue program execution. Messages can also be generated with the ABORT and STOP statements, but these statements terminate program execution after outputting a message.

MSG. is a predefined keyword with special meaning in the PUT statement and should not be used as an ordinary variable in other statements. As used in the PUT statement, MSG. refers to the list of messages generated during execution of the program. Each message consists of a message identifier and zero or more parameter values. (The message identifier is the item.id of an item in the system ERRMSG file.)

System messages are normally formatted and displayed on the user's terminal, or on the printer if a PRINTER ON statement has been executed. Messages are also copied to a buffer area; if the program is invoked by another program (the "calling" program) using the EXECUTE statement or by a PROC, the buffer can be later inspected by the calling program using GET or by the PROC using IF E. The messages placed in the MSG. buffer by PUT are not available to the program that placed them there.

`PUT(MSG.) 201,FILENAME`	Displays message 201 with one parameter, the string value of variable FILENAME; also copies message 201 and FILENAME value to MSG.

PWR Function

The PWR function raises a number to a specified power.

Syntax

PWR(base,exponent)

base value to be raised to a power

exponent value of the power; if zero, the function returns the value of one (1)

Description

If the base is zero and exponent is any number other than zero, the function returns a value of zero (0). If the values of base and exponent are such that the result would be greater than the largest allowable number, the function returns unpredictable numbers.

Note: *Another way to express the PWR function is X^Y, where X is raised to the Y power.*

The PWR function applies only to standard arithmetic, not the extended arithmetic package (string and floating point arithmetic). To raise a base resulting from an extended arithmetic function to a power requires including a special subroutine in the program. Figure 3-4 contains two subroutines recommended for this task.

```
A = 2                          The value 8 is returned in C.
B = 3
C = PWR(A,B)
```

For use with string arithmetic: *

```
SUBROUTINE(BASE,POWER,ANSWER)
ANSWER='1'
IF POWER ELSE RETURN
I=BASE
J=POWER
100  K=REM(J,2)
     J=NT(J/2)
     IF K THEN
       ANSWER=SMUL(ANSWER,I)
       IF J ELSE RETURN
     END
     I=SMUL(I,I)
     GOTO 100
END
```

For use with floating point arithmetic *

```
SUBROUTINE(BASE,POWER,ANSWER)
ANSWER='1E0'
IF POWER ELSE RETURN
I=BASE                    ;BASE is floating point
J=POWER                   ;POWER is numeric
100  K=REM(J,2)
     J=INT(J/2)
     IF K THEN
       ANSWER=FMUL(ANSWER,I)
       IF J ELSE RETURN
     END
     I=FMUL(I,I)
     GOTO 100
END
```

*These routines are based on an algorithm from Knuth's *The Art of Computer Programming*, Volume 2, Section 4.6.3, Page 399.)

Figure 3-4. Subroutines for Extended Arithmetic Power Function

READ{U} Statement

The READ{U} statement reads a file item and assigns its value in dynamic array format to a variable.

Syntax

READ{U} var FROM {file.var,} item.id {ON ERROR stmts} {LOCKED stmt} {THEN/ELSE stmt}

U	locks the item lock associated with the item to be accessed; if the item is currently locked by another BASIC program, the statement does not perform the read operation. The item does not have to exist in order for READU to lock it; in this case, READU executes the ELSE statements, but still locks the associated item lock. (The letter U is appended to the statement name to imply **update**, not **unlock**.)
var	variable into which the item is read; item is read in dynamic array format.
file.var	variable to which file was previously OPENed; if the file.var is omitted, the internal default file variable is used (that is, the file most recently opened without a file variable); if the specified file has not been opened prior to the execution of the READ statement, the program aborts.
item.id	name of item to be accessed.
ON ERROR stmts	statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be read due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when reading local files. The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be

terminated by an END statement as in the multi-line IF statement.

LOCKED stmts — statements to execute if the READ statement is unable to lock the item because another program has already locked it; statements may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multiple line IF statement.

THEN stmts — statements to be executed after item is successfully read into var; statements may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multiple line IF statement.

ELSE stmts — statements to be executed if the specified item does not exist; in this case, the contents of var remain unchanged; statements may appear on one line separated by semicolons or on multiple lines terminated by an END, as in the multiple line IF statement.

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

Description

The READU statement locks an item and then reads it. This can be used to prevent simultaneously updating an item by two or more users while still allowing multiple programs to access the file.

UltiNet Considerations

The ON ERROR clause allows the program to retrieve the UltiNet error number and take appropriate action. Such action could, for instance, include printing the associated message text via a PUT statement or STOP statement, and resuming or terminating program execution.

If a remote file cannot be read due to network errors and no ON ERROR clause is present, the program terminates with an error message.

Item Locks

Item locks are assigned based on the group of the disk file which contains (or would contain) an item and a hash value derived from the item.id. Items in different groups in the same file or in different files are never assigned the same item lock, but it is possible for more than one item in the same group to hash to, and be assigned, the same lock.

If an item is currently unlocked, setting an item lock prevents access to the item, and to any other items in the same group with the same item lock hash value, by other programs using the MATREADU, READU, or READVU statements. The program setting the lock, however, is allowed to lock other items in the same group with the same hash value using these statements.

An item will become unlocked when it, or any other item sharing the same item lock hash value, is updated by a WRITE, WRITEV, MATWRITE, or DELETE statement, or when it is unlocked by a RELEASE statement, or when the BASIC program is terminated. An item can be updated without unlocking it by using the WRITEU, WRITEVU, or the MATWRITEU statement.

There is a maximum number of item locks that can be locked at any one time. This number may vary from release to release. If a program attempts to lock an item when all item locks are already set, the program is suspended until a lock is unlocked.

Note: *Locked items can still be retrieved by the READ, READV, and MATREAD statements and by other system software, such as Recall, that do not pay attention to item locks.*

For information on read locks set by the system, see the section in Chapter 5 called "Read Locks."

```
READ A1 FROM X,"ABC" ELSE
  PRINT "NOT ABC"
  GOTO 70
  END
```

Reads item ABC from the file opened and assigned to file variable X and assigns its value to variable A1. If ABC does not exist, the text "NOT ABC" is printed and control passes to statement 70.

```
A="T1"
READ X FROM C,A ELSE STOP
```

Reads item T1 from the file opened and assigned to file variable C, and assigns its value to variable X. If TEST1 does not exist, program terminates.

```
READ Z FROM "Q" ELSE
  PRINT X; STOP
  END
```

Reads item Q from the file opened without a file variable and assigns its value to variable Z. If Q does not exist, prints value of X and terminates program.

```
READ Z FROM "Q" ON ERROR
  ERRNUM=SYSTEM(0)
  GOSUB PROCESSERR
  GOTO TOP
END ELSE PRINT X; STOP
```

Reads as above, or retrieves error number and performs local subroutine on UltiNet error number.

```
READU ITEM FROM INV, "30" LOCKED GOTO 500 ELSE STOP
```

Locks item "30". If the item is already locked, goes to label 500. If the item is not locked, the program continues; if the item does not exist, the program stops.

READNEXT Statement

The READNEXT statement reads the next element (typically, an item.id) from a select list. If multiple select lists are present, a select variable specifies the list to use.

Syntax

READNEXT var{,var2}{FROM select.var} {ON ERROR stmts}
THEN/ELSE stmts

var	variable into which string value of the next select list element (typically an item.id) is returned
var2	variable into which value number associated with this element is returned; if no value number is present, zero is returned
select.var	specifies a particular select list; if absent, the internal default select variable is used
ON ERROR stmts	statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be read due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when reading local files. The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.
THEN stmts	statements to be executed after select list element is successfully read into var; the THEN statements may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multiple line IF statement

ELSE stmts — statements to be executed if select list has been exhausted or if the select list does not exist; in this case, the contents of var remain unchanged; the ELSE statements may appear on one line separated by semicolons or on multiple lines terminated by an END, as in the multiple line IF statement

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

Description

A select list is a list of data generated by a BASIC SELECT statement, or by one of the system select commands: SELECT, SSELECT, QSELECT, or GET-LIST.

READNEXT can be used with select lists in one of the following ways:

- When one of the system select commands is executed immediately before running a BASIC program, the list generated is assigned to the BASIC program's internal default select variable. READNEXT statements (without specifying a select.var) can then be used to retrieve each list element in sequence.
- A select list may be generated by a command specified in an EXECUTE statement. If not redirected, the list is assigned to the internal default select variable of the program performing the EXECUTE. Alternatively, the select list may be redirected by the SELECT. or RTNLIST clause, setting up a variable as a select variable for use by READNEXT.
- The BASIC SELECT statement may be used to generate a select list, either to an explicit select variable or to the internal default select variable. However, if one of the system select commands was executed immediately before running the BASIC program, that list is assigned to the BASIC program's internal default select variable and the results of the first occurrence of a BASIC SELECT **are not** assigned to the internal default select variable.

For more information on SELECT and EXECUTE, please refer to these statements, listed alphabetically in this chapter.

The value number associated with a select list element is generated by the SSELECT command when performing an "exploding" sort. For

more information on exploding sorts, and on all the select verbs (SELECT, SSELECT, QSELECT, GET-LIST), see the *Ultimate System Commands Reference Guide.*

```
EXECUTE "SSELECT NAMEFILE BY LASTNAME",RTNLIST LIST
LOOP
  READNEXT ID FROM LIST ELSE STOP
  READ ITEM FROM NM,ID ELSE GOTO ERR
  PRINT ITEM<3>
REPEAT
```

The Recall SSELECT command is used to generate a select list of item.ids, corresponding to items in file NAMEFILE and sorted by attribute LASTNAME. The select list is saved in the variable LIST; the READNEXT statement then retrieves these item.ids from LIST in order. The corresponding item is read from the file opened to the variable NM.

```
EXECUTE "SSELECT AFILE BY-EXP INV#"
READNEXT ID,VN ELSE STOP
```

Uses default select list to read next item.id and assigns value number to variable VN.

```
SELECT FV
READNEXT A ELSE STOP
```

Selects item.ids from file opened to file variable FV and assigns them to default select variable; assigns the next element in the default list to variable A. If select list is exhausted, program terminates.

```
READNEXT A ON ERROR
  ERRNUM=SYSTEM(0)
  GOSUB PROCESSERR
  GOTO TOP
END ELSE STOP
```

Reads as example above, or retrieves error number and performs local subroutine on UltiNet error number.

READT{X} Statement

The READT{X} statement reads a tape record from magnetic tape. The tape unit and record length (block size) on the tape is as specified by the most recent T-ATT command.

Syntax

READT{X} var THEN/ELSE stmts

X	converts data to hexadecimal format; this feature allows binary data to be read by BASIC programs.
var	variable into which the string value of the next record from the current magnetic tape unit is returned.
THEN stmts	statements to be executed after record is successfully read into var; the THEN statements may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multiple line IF statement.
ELSE stmts	statements to be executed if record does not exist; in this case, the contents of var remain unchanged; the ELSE statements may appear on one line separated by semicolons or on multiple lines terminated by an END, as in the multiple line IF statement.

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

For information on converting hexadecimal data to binary, then writing it, see the description of WRITET.

Description

The tape unit must have previously been attached before issuing this command. If the tape unit has not been attached, or if an End-of-File (EOF) mark is read, the ELSE statements, if any, are executed, and the system function SYSTEM(0) returns a value of 5 (tape off line) or 6 (cartridge not formatted correctly for this operating system revision). (Please refer to the SYSTEM function, listed alphabetically in this chapter.)

If, however, the tape drive is inadvertently set to off line after the first tape instruction, the system allows the user to correct the condition. When a subsequent tape instruction is processed, the system displays:

```
Tape drive off line (C)ontinue/(Q)uit:
```

If C is entered, the system returns to the BASIC program and the tape instruction is re-executed. If Q is entered, the BASIC program is aborted and control returns to TCL. In either case, the ELSE statements are not executed and the BASIC program has no way to detect such adverse action.

Caution! *The tape drive should never be put off line while it is running under the control of any tape operation (BASIC, T-LOAD, T-DUMP, etc.). If it is inadvertently set to off line, the tape drive may lose its momentum and the tape read/write head may not be aligned with the current data block on tape. Even though the system allows the user to (C)ontinue, it is not guaranteed that valid data is then read or written.*

`READT B ELSE`
`  PRINT "NO"`
`  GOTO 5`
`END`
	The next tape record is read and its value assigned to variable B. If EOF is read or tape unit is not attached, "NO" is printed and control passes to statement 5.

`READTX A`
`V = A[35,6]`
`PAY = OCONV(V,'MCX')`
	Reads data as hexadecimal characters, extracts characters from a fixed position, and converts them to an ASCII string

READV{U} Statement

The READV{U} statement is used to read a single attribute value from an item in a file.

Syntax

READV{U} var FROM {file.var,} item.id, attrib.no {ON ERROR stmts} {LOCKED stmts} THEN/ELSE stmts

U	locks the item lock associated with the item to be accessed; if the item is currently locked by another BASIC program, the statement does not perform the read operation. The item does not have to exist in order for READVU to lock it; in this case, READVU executes the ELSE statements, but still locks the associated item lock. (The letter U is appended to the statement name to imply **update**, not **unlock**.)
var	variable into which the attribute is read
file.var	variable to which file was previously OPENed; if the file.var is omitted, the internal default file variable is used (that is, the file most recently opened without a file variable); if the specified file has not been opened prior to the execution of the READ statement, the program aborts
item.id	name of item to be accessed
attrib.no	attribute number to be accessed; if attribute does not exist, a null is assigned to var
ON ERROR stmts	statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be read due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when reading local files.

The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.

LOCKED stmts — statements to execute if the READV statement is unable to lock the item because another program has already locked it; statements may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multiple line IF statement

THEN stmts — statements to be executed after item is successfully read into var; statements may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multiple line IF statement

ELSE stmts — statements to be executed if the specified **item** (not attribute) does not exist; in this case, the contents of var remain unchanged; statements may appear on one line separated by semicolons or on multiple lines terminated by an END, as in the multiple line IF statement.

Description

The READV statement makes efficient use of system resources when a single attribute needs to be accessed from an item. However, when it is used repeatedly to access several attributes, this efficiency is lost. When several attributes need to be accessed, either the READ or MATREAD statement should be used to read an item into a BASIC variable. Then dynamic array subscripts (for READ) or dimensioned array subscripts (for MATREAD) should be used with the variable to reference individual attributes.

Item Locks

Item locks are assigned based on the group of the disk file which contains (or would contain) an item and a hash value derived from the item.id. Items in different groups in the same file or in different files are never assigned the same item lock, but it is possible for more than one item in the same group to hash to, and be assigned, the same lock.

If an item is currently unlocked, setting an item lock prevents access to the item, and to any other items in the same group with the same item lock hash value, by other programs using the MATREADU, READU, or READVU statements. The program setting the lock, however, is allowed to lock other items in the same group with the same hash value using these statements.

An item will become unlocked when it, or any other item sharing the same item lock hash value, is updated by a WRITE, WRITEV, MATWRITE, or DELETE statement, or when it is unlocked by a RELEASE statement, or when the BASIC program is terminated. An item can be updated without unlocking it by using the WRITEU, WRITEVU, or the MATWRITEU statement.

There is a maximum number of item locks which may be locked at any one time. This number may vary from release to release. If a program attempts to lock an item when all item locks are already set, the program is suspended until a lock is unlocked.

Note: *Locked items can still be retrieved by the READ, READV, and MATREAD statements and by other system software, such as Recall, that do not pay attention to item locks.*

For information on read locks set by the system, see the section in Chapter 5 called "Read Locks."

```
READV X FROM A, "TEST", 5 ELSE
   PRINT ERR
   GOTO 70
END
```

Reads fifth attribute of item TEST (in the file opened and assigned to variable A) and assigns value to variable X. If item TEST is non-existent, then value of ERR is printed and control passes to statement 70.

```
READV X FROM A, "TEST",5 ON ERROR
   ERRNUM=SYSTEM(0)
   GOSUB PROCESSERR
   GOTO TOP
END ELSE PRINT ERR; GOTO 70
```

Reads as above, or retrieves error number and performs local subroutine on UltiNet error number.

```
READVU ATT FROM B, "REC",6 ELSE STOP
```

Lock item REC. Read attribute 6 to variable ATT or, if REC is non-existent, execute the ELSE clause. The item remains locked in either case.

```
READVU NAME FROM B, "REC",6 LOCKED GOTO BUSY ELSE STOP
```

As above, except that if REC is already locked, branch to statement label BUSY.

```
READVU NAME FROM B, "REC",6 ON ERROR GOTO PROCESSERR
ELSE STOP
```

As first READVU example above, or branch to local subroutine to process UltiNet error number.

RELEASE Statement

The RELEASE statement unlocks specified items or all items locked by the program. It can also be used to release named COMMON areas.

Syntax

RELEASE {{file.var,} item.id} {ON ERROR stmts}
RELEASE /name/

file.var	variable to which file was previously OPENed; if the file.var is omitted, the internal default file variable is used (that is, the file most recently opened without a file variable)
item.id	name of item to be released
ON ERROR stmts	statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be released due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when releasing local files. The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.
/name/	name of COMMON area to be released; the slashes (/) are required

Description

If the RELEASE statement is used without a file.var and without an item.id, all items that have been locked by the program are unlocked.

The RELEASE statement is useful when an abnormal condition is encountered during multiple file updates. A typical sequence is to mark

the item with an abnormal status, write it to the file and then RELEASE all other locked items.

When a named COMMON block is released, its name is removed from the list of named COMMON blocks and all space used by the variables is released to free (overflow) space. However, the variable names cannot be reused unless the named COMMON statement is re-executed.

A named COMMON block can be released by any program that uses it.

UltiNet Considerations

The ON ERROR clause allows the program to retrieve the UltiNet error number and take appropriate action. Such action could, for instance, include printing the associated message text via a PUT statement or STOP statement, and resuming or terminating program execution.

If items in a remote file cannot be released due to network errors and no ON ERROR clause is present, the program terminates with an error message.

`RELEASE`	Releases all items locked by the program.
`RELEASE CUST.FILE, PART.NO`	Releases item lock corresponding to PART.NO in the file opened and assigned to variable CUST.FILE.
`RELEASE AFILE, "ITEM3"`	Releases ITEM3's item lock.

`RELEASE AFILE, "ITEM3" ON ERROR`
`  ERRNUM=SYSTEM(0)`
`  GOSUB PROCESSERR`
`  GOTO TOP`
`END`
	Releases as above, or retrieves error number and performs local subroutine on UltiNet error number.

`RELEASE /PEOPLE/`	Releases named common area; information in the variables is no longer accessible.

REM Function

The REM function returns the remainder of one number divided by another.

Syntax

REM(dividend,divisor)

dividend numeric expression

divisor numeric expression

Description

The REM function is equivalent to the MOD (modulo) function. (Please refer to the MOD function, listed alphabetically in this chapter.)

`Q = REM(11,3)`	Assigns the value 2 to variable Q.

REM Statement

The REMark statement, which can be specified as REM, !, or *, is used to write non-executable comments about a program. Remarks can identify a function or section of program code, as well as explain its purpose and method.

Syntax

REM text ...
! text ...
* text ...

Description

REM, ! , or * must be placed at the beginning of the statement, but may appear anywhere on a line (for example, after another statement on the same line). A semicolon must be used to separate a remark statement from any other statement on the same line. The text may be any arbitrary characters, up to the end of the line of code.

Remark statements do not affect program execution.

When writing comments on a line whose elements are separated by commas and that continues on to the next line, a semicolon must follow the comma and the comments must start with an asterisk (*). The following statements are included:

```
CALL sub(a,b,..., ; *comments
                ...,x)

COM{MON} a,b,..., ; *comments
                ...,x

DIM{ENSION} a(n),b(m),..., ; *comments
                ...,x(z)

EQU{ATE} a TO b, c TO d,..., ; *comments
                ...,x TO y
```

```
REM Program to print the numbers from 1 to 10
 I = 1;                        * Start with 1
BEG: PRINT I;                  * print the value
 IF I = 10 THEN STOP;          * stop if done
 I += 1;                       * increment I by 1
 GOTO BEG;                     * begin loop again
 END

010 COMMON FIRST,              ; *This is the first comment
011        CTR,                ; *This is another comment
012        LAST                ; *This is the last comment
```

REMOVE Statement

The REMOVE statement places the next element of a dynamic array into a specified variable and returns a value that corresponds to the delimiter encountered following the element. The REMOVE statement does not change the dynamic array.

Syntax

REMOVE var FROM array.var SETTING delimiter.var

var	variable to which the substring is assigned
array.var	dynamic array being used
delimiter.var	variable whose value indicates the delimiter character for the substring last assigned

Description

The REMOVE statement is useful for extracting successive elements of a dynamic array because it avoids repeated scanning.

Each time REMOVE is executed, the next element in the dynamic array is assigned to the specified variable. An internal pointer is left pointing to the delimiter following the element just assigned and a value is assigned to the delimiter.var that indicates the type of delimiter.

The REMOVE statement can be executed repeatedly until all substrings have been processed.

The pointer is reset to the beginning of the array if the array.var is used in an assignment statement. For example, you can reset the pointer in the current array by assigning the array to itself. The pointer is also reset if you attempt to REMOVE past the end of the array.

The possible values returned in the delimiter.var are:

0	segment mark (SM) for end-of-string
2	attribute mark (AM)
3	value mark (VM)
4	subvalue mark (SVM)
5	ASCII value 250 or 251
6	ASCII value 249

```
A = 'ABC':AM:'123':VM:'456':AM:'XYZ'
FOR I =1 TO 4
  REMOVE STRING FROM A SETTING DELIMITER
  PRINT "STRING = ":STRING,"DELIMITER = ":DELIMITER
NEXT
```

result:

STRING = ABC DELIMITER = 2
STRING = 123 DELIMITER = 3
STRING = 456 DELIMITER = 2
STRING = XYZ DELIMITER = 0

The first pass extracts an attribute ('ABC'). The second pass extracts a value ('123'). The third pass extracts an attribute ('456'). The fourth pass extracts the last attribute ('XYZ'), which is delimited by a segment mark.

REPEAT Statement

The REPEAT statement is the last statement in a LOOP statement sequence. Please refer to the LOOP statement for information about the entire LOOP statement sequence.

REPLACE Function

The REPLACE function returns a dynamic array with a specified attribute, value, or subvalue replaced.

Syntax

REPLACE(var,attrib.no{,val.no{,subval.no}};expr)
REPLACE(var,attrib.no,val.no,subval.no,expr)

var	dynamic array in which to replace expression
attrib.no	position of the attribute to be replaced; if -1, expression is inserted after last attribute
val.no	position of the value to be replaced; if -1, expression is inserted after last value in specified attribute
subval.no	position of the subvalue to be replaced; if -1, expression is inserted after last subvalue in specified attribute and value
expr	value to use for replacement; can be an expression using operators with precedence levels 1-4 only (dynamic array extraction, substring, exponentiation, multiplication, division, addition, and subtraction).

In the first form, a semicolon separates the attribute, value, and subvalue numbers from the new data expression (expr); trailing zero value and subvalue numbers are not required.

Description

If val.no and subval.no are absent or have a value of 0, the entire attribute specified by attrib.no is replaced. If val.no is present and subval.no is absent or has a value of 0, the entire value specified by val.no is replaced. If attrib.no, val.no, and subval.no are all non-zero, the subvalue specified by subval.no is replaced.

The following example shows two ways to code a function:

First Form:

```
X=REPLACE(X,10;'XXXXX')
```

Second Form:

```
X=REPLACE(X,10,0,0,'XXXXX')
```

The value "XXXXX" replaces the existing data in attribute 10.

Note: *An assignment statement can also be used to replace an attribute, value, or subvalue in a dynamic array and store the result back into the variable containing the original dynamic array. For example, X<2>=6 is equivalent to X=REPLACE(X,2;6). For more information, please see the = (Assignment) statement, listed alphabetically in this chapter.*

`X=REPLACE(X,4;"")`	Replaces attribute 4 of dynamic array X with the empty (null) string.
`Y=REPLACE(X,4,0,0,"")`	Replaces attribute 4 of dynamic array X with the empty (null) string, and assigns resulting dynamic array to Y.
`VAL="TEST STRING"` `D=REPLACE(D,4,3,2,VAL)`	Replaces subvalue 2 of value 3 of attribute 4 in dynamic array D with the string value "TEST STRING".
`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;"XXX")`	Replaces value 3 of attribute 2 of dynamic array B with the value "XXX", and assigns resulting dynamic array to A.

RETURN (TO) Statement

The RETURN {TO} statement returns from a subroutine.

Syntax

RETURN {TO label}

label label of statement within the local BASIC program to which the subroutine is to return

Description

The RETURN statement, without the TO clause, transfers control from a subroutine back to the statement immediately following the statement that invoked the subroutine. The subroutine can be either a local or an external subroutine.

To insure proper flow of control, each local subroutine must be exited by using a RETURN {TO} statement, not a GOTO statement.

Note: *Local subroutines are invoked by the GOSUB statement; external subroutines are invoked by the CALL statement. Further discussion of local subroutines may be found under the GOSUB statement, and of external subroutines under the CALL and SUBROUTINE statements. Please refer to these statements, listed alphabetically in this chapter.*

```
A=A+1
MSG = A:' orders processed'
GOSUB 100
STOP
  .
  .
100 * Common print routine
    PRINT @(-1)
    PRINT @(20,2):MSG
    RETURN
```

REUSE Function

The REUSE function can be used when performing arithmetic operations on two dynamic arrays that may have unequal numbers of attributes, values, or subvalues. In these cases, REUSE reuses the value of the last attribute, value, or subvalue within the same level until the next higher delimiter is encountered.

Syntax

REUSE(array.expr)

array.expr expression that evaluates to the dynamic array to be used; it may also be a single value

Description

With REUSE, the value of the last attribute, value, or subvalue in the same level is reused until the next higher delimiter is encountered.

Without REUSE, a zero is used as the value when a corresponding attribute, value, or subvalue is not available (if the operation is division, a one is used if the divisor is missing). The substitution stops when the next higher delimiter is encountered.

If the array.expr evaluates to a single value, the REUSE function repetitively processes that value against each element in the dynamic array.

REUSE can be used only as part of an expression; it cannot be assigned to a variable.

```
ARRAY1 =  1:AM: 2:VM: 2:VM: 2:AM: 3:VM:10
ARRAY2 = 10:AM:20:VM:20:VM:20:AM:30
ARRAY3 = ARRAY1 + REUSE(ARRAY2)
```

result:

```
ARRAY3 = 11:AM:22:VM:22:VM:22:AM:33:VM:40
```

The elements of ARRAY3 are composed of the sums of 6 elements in ARRAY1 and 5 elements in ARRAY2, with the last ARRAY2 element (30) being used twice.

```
ARRAY1 = 110:VM:100:AM:120:AM:140
ARRAY2 = ARRAY1 * REUSE(0)
```

result:

```
ARRAY2 = 0:VM:0:AM:0:AM:0
```

Each element of ARRAY1 is multiplied by zero (0), which assigns the value 0 to each element in ARRAY2.

```
ARRAY1 = 1:AM: 2:VM:4:AM: 6:VM: 7:AM: 8
ARRAY2 = 5:AM:10      :AM:20:VM:30
ARRAY3 = ARRAY1 + ARRAY2
ARRAY4 = ARRAY1 + REUSE(ARRAY2)
```

result:

```
ARRAY3 = 6:AM:12:VM: 4:AM:26:VM:37:AM: 8
ARRAY4 = 6:AM:12:VM:14:AM:26:VM:37:AM:38
```

ARRAY3 has four attributes; the second and third attributes are multi-valued, and 0 was used twice because of missing elements in ARRAY2.

For ARRAY4, elements in the second and fourth attributes in ARRAY2 were each REUSEd to account for the missing elements.

REWIND Statement

The REWIND statement rewinds the tape on the unit specified by the most recent T-ATT command.

Syntax

REWIND {THEN stmts} {ELSE stmts}

THEN stmts — statements to be executed after tape is successfully rewound; the THEN statements may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multiple line IF statement

ELSE stmts — statements to be executed if tape is not attached; the ELSE statements may appear on one line separated by semicolons or on multiple lines terminated by an END, as in the multiple line IF statement

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

Description

The tape unit must have previously been attached before issuing this command. If the tape unit has not been attached, the ELSE statements, if any, are executed, and the system function SYSTEM(0) returns a value of 5 (tape off line) or 6 (cartridge not formatted correctly for this operating system revision). (Please refer to the SYSTEM function, listed alphabetically in this chapter.)

`REWIND ELSE STOP`	Tape is rewound to BOT.

RND Function

The RND function returns a random number.

Syntax

RND(expr)

expr number on which to base random number; if necessary, expr is truncated (not rounded) to nearest integer

Description

The seed used in the random number generation is reloaded each time the system is coldstarted.

If expr is between 0 and 32767, the RND function generates an integer between 0 and one less than the number specified in expr.

If expr is 0 or 65536, the RND function generates an integer that is machine-dependent and is not reliable as a random number.

If expr is between 32768 and 65535, the RND function generates an integer between 0 and (65536-expr).

If expr is greater than 65536, it is adjusted to a number less than 65,356 by modulo 65536.

If expr is less than zero, the absolute value of the number is used.

`Z = RND(10) + 1`	Assigns a random number between 1 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 the variable B.

`Y = RND(51)`	Assigns a random number between 0 and 50 (inclusive) to the variable Y.

RQM Statement

The RQM (release quantum) statement suspends program execution for a specified time.

Syntax

RQM {expr}

expr if expr is a numeric value, defines number of seconds RQM is to pause program; if expression is a string value in the form hh:mm, defines time of day program is to end pause

Description

If no expression is used, RQM pauses for one second.

If the value of expression is less than or equal to zero, a command is sent to the kernel to perform a kernel RQM, which deactivates the process until its next timeslice.

The time-shared environment of the Ultimate system allows concurrent execution of several programs, with each program executing for a specific time period (called a time-slice or quantum) and then pausing while other programs continue execution. The RQM statement relinquishes the program's current time-slice and causes it to "sleep" for one second.

The RQM statement is equivalent to the SLEEP statement, which is listed alphabetically in this chapter.

```
RQM 5                                  Pauses program for five seconds.

A = '13:05'
RQM A                                  Pauses program until 1:05 p.m.

* PROGRAM SEGMENT TO SOUND TERMINAL BELL FIVE TIMES.
BELL=CHAR(7)
FOR I=1 TO 5
  PRINT BELL:                          RQM statement causes the terminal to
  RQM                                  beep at one-second intervals.
NEXT
```

SADD Function

The SADD (string addition) function adds two string numbers and returns the result as a string number.

Syntax

SADD(x,y)

x any valid number or string number of any magnitude and precision

y any valid number or string number of any magnitude and precision

Description

The result of the SADD function is a string number. Thus, the function can be used in any expression where a string or string number would be valid, but not necessarily where a standard number would be valid. This is because string numbers may exceed the range of numbers which can be accommodated with standard arithmetic operators.

If either x or y contains non-numeric data, an error message is generated and the result of the addition is zero.

`T=SADD(S1,S2)`	Assigns sum of variables S1 and S2 to variable T.
`PRINT SADD(X,".004")`	Prints sum of variable X and string constant .004.
`A=SADD("1.03047",B)`	Assigns the sum of string constant 1.03047 and variable B to variable A.
`X=SADD(A,SADD(B,C))`	Uses string sum of variables B and C in string addition with variable A; assigns sum to variable X.

SCMP Function

The SCMP (string compare) function compares two string numbers.

Syntax

SCMP(x,y)

x any valid number or string number of any magnitude and precision

y any valid number or string number of any magnitude and precision

Description

The result of the SCMP function is a number: -1, 0, or 1. If x is less than y, the result is -1. If x and y are equal, the result is 0. If x is greater than y, the result is 1.

If either x or y contains non-numeric data, an error message is generated and the result of the comparison is zero (0).

The function can be used in any expression where a number or string would be valid.

```
IF SCMP(X,Y) = 0 THEN GOTO 100
```

The result of the comparison determines whether program execution branches to statement 100 or continues in sequence.

```
IF SCMP(X,Y) < 0 THEN
 PRINT X:' IS LESS THAN ':Y
END
```

The PRINT operation is executed only if the result of the IF statement is true (-1 was the result of the SCMP function).

```
ON 2+SCMP(VAL1,VAL2) GOTO 10, 110,120
```

The result of the comparison is used to create an index of 1,2, or 3 for the ON GOTO statement.

SDIV Function

The SDIV (string division) function divides the first string number by the second and returns the result as a string number.

Syntax

SDIV(x,y)

x dividend; any valid number or string number of any magnitude and precision

y divisor; any valid number or string number with up to 13 significant digits and any precision

Description

The result of the SDIV function is a string number. Thus, the function can be used in any expression where a string or string number would be valid, but not necessarily where a standard number would be valid. This is because string numbers may exceed the range of numbers which can be accommodated with standard arithmetic operators.

The divisor in SDIV is restricted to 13 significant digits and the quotient is restricted to 14 significant digits.

If either x or y contains non-numeric data, an error message is generated and the result of the division is zero. If y is zero, an error message is displayed that states division by zero is illegal; the result is zero.

`VELOCITY=SDIV(DISTANCE,TIME)`	Assigns result of variables DISTANCE divided by TIME to variable VELOCITY
`PRINT SDIV(X,".004")`	Prints quotient of variable X divided by string constant .004.
`A=SDIV("1.030476",B)`	Assigns to variable A the result of dividing string constant 1.030476 by variable B.
`X=SDIV(A, SDIV(B,C))`	Uses string result of variable B divided by variable C in string division with variable A; assigns sum to variable X.
`Y=SDIV("10","3")`	Result is 3.3333333333333.

SEEK Statement

The SEEK statement positions the buffer pointer in either the program argument list or the system message buffer.

Syntax

SEEK(ARG.{, arg.no}) {THEN stmts} {ELSE stmts}
SEEK(MSG.{, arg.no}) {THEN stmts} {ELSE stmts}

ARG.	uses list of arguments, if any, following the program name in the TCL command that invoked the program; any string preceded by a space is considered an argument
MSG.	uses list of message identifiers and parameters, if any, resulting from the last EXECUTE statement or by a PUT statement in an EXECUTEd program
arg.no	integer that specifies the position of the element in the list to move the pointer to; if arg.no is not present, the next pointer is positioned to the next element on the list; if this is the first SEEK statement to be executed, the pointer is moved to the first element on the list
THEN stmts	statements to execute if pointer is positioned to element
ELSE stmts	statements to execute if no element is present in specified position

Description

One or more SEEK(MSG.) statements can be used to locate the system messages generated by a program invoked via an EXECUTE statement. Only the ERRMSG item.ids and parameters are copied to MSG. (MSG. is reset to null just prior to the execution of an EXECUTE statement.)

The EOF function is available to test for end-of-argument or end-of-message list. (Please refer to the EOF function listed alphabetically in this chapter.)

Note: *ARG. and MSG. are predefined keywords with special meaning and should not be used as ordinary variables in other statements.*

The SEEK statement is similar to the GET statement except that no data transfer takes place. Also, the SEEK statement can be used in conjunction with the GET statement. Once the internal pointer position has been set by a SEEK statement, a subsequent GET statement will return the argument set up by the SEEK statement. For further information on GET, as well as the ARG. and MSG. keywords, please refer to the GET statement, listed alphabetically in this chapter.

```
SEEK (ARG., 3) THEN GOSUB 10000 ELSE
  PRINT "NOT ENOUGH ARGUMENTS"
  STOP
END
```

If the argument is found, the subroutine is executed; otherwise, the message is printed and program execution terminates.

SELECT Statement

The SELECT statement builds a select list from a file or a dynamic array for use with the READNEXT statement.

Syntax

SELECT {var} {TO select.var} {ON ERROR stmts}

var — either a file variable previously initialized in an OPEN statement, or a variable whose current value is a dynamic array

select.var — variable to which the select list is to be assigned; if omitted, the list is assigned to the program's internal default select variable.

If the BASIC SELECT statement is used to generate a select list to the internal default select variable, and if one of the system select commands was executed immediately before running the BASIC program, that list is assigned to the BASIC program's internal default select variable and the results of the first occurrence of a BASIC SELECT **are not** assigned to the internal default select variable.

Note: A select variable has meaning only in SELECT and READNEXT statements; its value outside these statements is undefined.

ON ERROR stmts — statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be selected due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when opening local files.

The statements may be on a single line or on multiple lines. If multiple lines are used, the clause

must be terminated by an END statement as in the multi-line IF statement.

Description

If a file variable is used, SELECT builds a list of item.ids corresponding to all items in the file. If a dynamic array is used, SELECT builds a list whose elements are copies of the attributes in the dynamic array. Only the first values of multi-valued attributes are selected.

When selecting file items, the SELECT statement builds the same list of item.ids as would be built by the Recall SELECT command without any selection criteria. But unlike the Recall SELECT command, which reads the entire file at one time, the BASIC SELECT statement reads one group of items at a time. This means that if items are being created in the selected file, the BASIC SELECT list may include the new items (depending on their groups and how far the SELECT has progressed) whereas the Recall SELECT list will not.

Select lists are discussed in more detail under the READNEXT statement, listed alphabetically in this chapter.

UltiNet Considerations

The purpose of the ON ERROR clause is to allow the program to retrieve the UltiNet error number and take appropriate action. Such action could, for instance, include printing the associated message text via a PUT statement or STOP statement, and resuming or terminating program execution.

If a remote file cannot be accessed due to network errors and no ON ERROR clause is present, the program terminates with an error message.

`SELECT`	Builds a list of item.ids using the default variable of the last file opened without a file variable.
`SELECT BP TO BLIST`	Builds a list of item.ids for the file opened and assigned to file variable BP. Assigns the list to select-variable BLIST.
`READ A FROM FILEX,'ALIST' ELSE STOP` `SELECT A`	Creates a select list of the attributes in item ALIST.

`SELECT A ON ERROR`
`  ERRNUM=SYSTEM(0)`
`  GOSUB PROCESSERR`
`  GOTO TOP`
`END`
	Creates select list as above, or retrieves error number and performs local subroutine on UltiNet error number.

SEQ Function

The SEQ function converts an ASCII character to its corresponding decimal value.

Syntax

SEQ(expr)

expr ASCII character string to be converted; the first character of the string is converted to its corresponding decimal value

Description

For a complete list of ASCII codes, refer to Appendix D of this manual.

The SEQ function is the inverse of the CHAR function. (Please refer to the CHAR function, listed alphabetically in this chapter.)

```
S = 'No errors'
L = LEN(STRING)
DIM C(L)
FOR I=1 TO L
  C(I) = SEQ(S[I,1])
NEXT
```

Puts the decimal equivalents of individual characters of string S into elements of dimensioned array C.

SIN Function

The SIN function returns the sine of an angle expressed in degrees.

Syntax

SIN(expr)

expr number of degrees in the angle

Description

The function generates the sine of the angle.

```
A = 32                          Returns 0.5299, the sine of 32°.
PRINT SIN(A)
```

SLEEP Statement

The SLEEP statement pauses a program for a specified amount of time, which can be expressed either as number of seconds or a specific time of day.

Syntax

SLEEP {expr}

expr if expr is a numeric value, defines number of seconds SLEEP is to pause program; if expr is a string value in the form hh:mm, defines time of day program is to end pause

Description

If the value of expression is less than or equal to zero, a command is sent to the kernel to perform a kernel RQM, which deactivates the process until its next timeslice.

If no expression is used, SLEEP pauses for one second.

The SLEEP statement is equivalent to the RQM statement, which is listed alphabetically in this chapter.

```
SLEEP 5                    Pauses program for five seconds.

A = '13:05'
SLEEP A                    Pauses program until 1:05 p.m.
```

SMUL Function

The SMUL (string multiplication) function multiplies two string numbers and returns the result as a string number.

Syntax

SMUL(x,y)

x any valid number or string number of any magnitude and precision

y any valid number or string number of any magnitude and precision

Description

The result of the SMUL function is a string number. This function can be used in any expression where a string or string number would be valid, but not necessarily where a standard number would be valid. This is because string numbers may exceed the range of numbers which can be accommodated with standard arithmetic operators.

`PAY=SMUL(HOURS,RATE)`	The variable PAY is assigned the product of HOURS times RATE.
`PRINT SMUL(X,"1.0015")`	The variable X is multiplied by constant 1.0015 and the result is printed.
`A=SMUL("1.030476",B)`	The constant 1.030476 is multiplied by variable B and the result is assigned to variable A.
`X=SMUL(A, SMUL(B,C))`	The product of variables B and C is multiplied by variable A; the result is assigned to X.

SORT Function

The SORT function sorts elements in a dynamic array. The sort is in ASCII sequence.

Syntax

SORT(var)

var evaluates to dynamic array

Description

The elements are sorted in ascending order, left justified.

The dynamic array is sorted by the highest system delimiter in the array. That is, if the dynamic array contains any attribute marks, the sort is by attributes; values and subvalues are unaffected and remain with the original attribute. If the dynamic array contains value marks and no attribute marks, the sort is by value; subvalues are unaffected and remain with the original value. If the dynamic array contains subvalue marks and neither attribute nor value marks, the sort is by subvalue.

```
A = 'Hello':AM:'Goodbye:VM:'See you later'
S = SORT(A)
```

There is an attribute mark in the string, so the sort is according to attributes; the first value of each attribute determines the order.

result:

```
S = 'Goodbye:VM:'See you later':AM:'Hello'
```

```
A = 'Hello':VM:'Goodbye:VM:'See you later'
S = SORT(A)
```

There are no attribute marks in the string, but there are value marks, so the sort is by values.

result:

S = 'Goodbye:VM:'Hello':VM:'See you later'

SOUNDEX Function

The SOUNDEX function returns the soundex code for a specified string. The code consists of the first letter of the word, plus values for the next three consonants or combinations of consonants.

Syntax

SOUNDEX(expr)

expr string to be used

Description

These codes can be useful for cross references based on words or names that sound alike. Soundex codes also overcome problems with upper case and lower case, typographical errors, and misspellings in a database.

The first value in the soundex code is the first alphabetic character in the string. Subsequent values in the soundex codes are numeric values given to consonants. If two or more characters with the same numeric value are adjacent, only one value is returned. Characters that are not consonants, other than the first character, are ignored.

Words with a similar arrangement of consonants have similar soundex codes, regardless of the actual spelling; similar sounding consonants may have the same soundex code. If two adjacent consonants have the same soundex code, only one instance of the code is returned. For example, the following words all have the same soundex code (L6):

Laura	Lora	Laurie	lorry
Lorrie	Lori	LARRY	

Table 3-4 lists the soundex codes returned by the function.

```
A = SOUNDEX("SMITH")        A = "S53"
B = SOUNDEX("SMITHS")       B = "S532"
C = SOUNDEX("SMITHSON")     C = "S532"
D = SOUNDEX("123")          D = ""
```

Table 3-4. Soundex Codes

Letters	Code
a,e,i,o,u,h,w,y	null
b,f,p,v	1
c,g,j,k,q,s,x,z	2
d,t	3
l	4
m,n	5
r	6

SPACE Function

The SPACE function generates a string value containing a specified number of blank spaces.

Syntax

SPACE(expr)

expr number of blank spaces to be generated

`B = 14` `C = SPACE(B)`	Assigns to variable C a string of 14 blank spaces.
`DIM M(10)` `MAT M = SPACE(20)`	Assigns a string consisting of 20 blanks to each of the 10 elements of array M.
`PRINT SPACE(10):"name: "`	Prints 10 spaces followed by the characters **name:**

SQRT Function

The SQRT function returns the positive square root of a positive number.

Syntax

SQRT(expr)

expr positive number for which to generate the square root; if expr evaluates to less than or equal to zero, the function returns a value of zero

```
Y = SQRT(36)
```

Assigns the value 6 to variable Y.

SSUB Function

The SSUB (string subtraction) function subtracts the second string number from the first string number and returns the result as a string number.

Syntax

SSUB(x,y)

x any valid number or string number of any magnitude and precision

y any valid number or string number of any magnitude and precision

Description

The result of the SSUB function is a string number. Thus, the function can be used in any expression where a string or string number would be valid, but not necessarily where a standard number would be valid. This is because string numbers may exceed the range of numbers which can be accommodated with standard arithmetic operators.

`T=SSUB(S1,S2)`	Assigns difference of variables S1 and S2 to variable T.
`PRINT SSUB(X,".004")`	Prints difference of variable X and string constant .004.
`A=SSUB("1.03047",B)`	Assigns the difference of string constant 1.03047 and variable B to variable A.
`X=SSUB(A, SSUB(B,C))`	Uses the difference of variable B and C in string subtraction with variable A; the result is assigned to X.

STOP Statement

The STOP statement terminates program execution.

Syntax

STOP {errnum{,param, param, ...}}

errnum error message number (item.id) in the ERRMSG file

param parameters to be used within the error message format; must be separated by commas; may be variables or literals

Description

A STOP statement may be placed anywhere within the BASIC program.

STOP terminates a BASIC program but, if a PROC is used to invoke the program, STOP does not terminate the PROC.

The ABORT statement can also be used for program termination. However, unlike STOP, ABORT also terminates PROC execution if the program was invoked from a PROC. (Refer to the ABORT statement, listed alphabetically in this chapter.)

```
PRINT 'PLEASE ENTER FILE NAME':
INPUT FN
OPEN FN TO FFN ELSE STOP 201, FN
```

This program requests a file name from the user and attempts to open the file. If an incorrect file name is entered, the standard system error message "[201] 'xxx' IS NOT A FILE" is printed and the program is terminated.

STORAGE Statement

The STORAGE statement allows a program to change the three buffer sizes used for storing string data in variables.

Syntax

STORAGE small-buffer, med-buffer, large-buffer

small-buffer — size of small buffer; must be multiple of 10 and must be less than med-buffer and large-buffer. The default small buffer size is 50 bytes.

med-buffer — size of medium buffer; must be multiple of 10 and must be greater than small-buffer and less than large-buffer. The default medium buffer size is 150 bytes.

large-buffer — size of large buffer; must be multiple of 10 and must be greater than med-buffer and small-buffer. The default large buffer unit size is 250 bytes.

Description

When a variable takes on a string value, the string is stored directly in the variable's descriptor area if it is less than nine characters long. If it is nine or more characters long, the string is stored in a buffer to which the descriptor then points.

The buffers used to hold these strings are built in certain sizes specified by three numeric parameters: the size of a small buffer, the size of a medium buffer, and the size of a large buffer unit. The size of a large buffer is one or more large buffer units.

`STORAGE 40, 100, 180`	Defines buffer sizes at 40 (small), 100 (medium), and 180 (large).
`STORAGE 100, 200, 300`	Defines buffer sizes at 100 (small), 200 (medium), and 300 (large).

STR Function

The STR function generates a string value containing a specified number of occurrences of a specified string.

Syntax

STR(string,count)

string the string to be replicated

count number of occurrences to be generated.

```
VAR = STR("A",5)
```

Assigns to variable VAR a string containing five A's.

```
VAR = 'A':CHAR(254)
N = STR(VAR,4)
```

Assigns to variable N the string value containing four substrings consisting of the letter A followed by an attribute mark.

```
PRINT STR('0123456789',SYSTEM(2)/10)
```

Prints the numbers 0 thru 9 across the page (SYSTEM(2) contains the current terminal page width).

SUBROUTINE Statement

The SUBROUTINE statement provides external subroutine capabilities for a BASIC program.

Syntax

SUBROUTINE {name} {(argument list)}

name — may be used to indicate the name of the subroutine, but this is ignored by the compiler, which uses the item.id of the subroutine as the subroutine name

argument list — one or more variables, separated by commas, that take on the actual values passed to the subroutine; list may be continued on multiple lines; each line except the last must end with a comma. If the argument is the name of an array given in a DIM statement, the word MAT must precede the array name; arrays passed through an argument list must be dimensioned in both the calling program and the subroutine. Multiple arrays may be passed, as needed.

Description

An external subroutine is a subroutine that is compiled separately from the program or programs that call it.

The SUBROUTINE statement is used to identify the program as a subroutine and must be the first statement in the program. A program that begins with the SUBROUTINE statement cannot be run except by a CALL statement from another program.

The SUBROUTINE statement is used in conjunction with the CALL statement. The CALL statement transfers control to the external subroutine, which may then return control using the RETURN statement. (Please refer to the CALL and RETURN statements, listed alphabetically in this chapter.)

There is no correspondence between variable names or labels in the calling program and the subroutine. The only information passed between the calling program and the subroutine is the values of the arguments. In addition, each subroutine has its own internal select

variable and file variable. A list selected to the internal select variable of one program is not passed to the internal select variable of another program. Similarly, the name of a file opened to the internal file variable of one program is not passed to internal file variable of another program.

When the CALL statement is executed, subroutine arguments are first evaluated and their values assigned to the corresponding variables named in the subroutine's SUBROUTINE statement. These variables may then be assigned new values by the subroutine. When control returns to the calling program, any variables used as subroutine arguments are updated to reflect the most recent values of the corresponding variables in the subroutine. Constants and other expressions used as subroutine arguments are not changed.

Care should be taken not to update the same variable referenced by more than one name in an external subroutine. This can occur if a variable in COMMON is also passed as a subroutine parameter.

External subroutines may call other external subroutines, including themselves.

An external subroutine must begin with a SUBROUTINE statement and should contain a RETURN statement. GOSUB and RETURN statements may be used within the subroutine, but when a RETURN is executed with no corresponding GOSUB, control passes to the statement following the corresponding CALL statement in the calling program. If the subroutine terminates execution without executing a RETURN (such as by executing a STOP statement, or by "running out" of statements at the end of the subroutine), control never returns to the calling program.

The CHAIN statement should not be used to chain from an external subroutine to another BASIC program.

Dimensioned arrays may be passed to external subroutines. Array dimensions may be different between CALLing program and subroutine, as long as the total number of elements matches.

Arrays are copied in row major order; that is, all columns in row 1 are copied before the first column in row 2, and so on.

```
SUBROUTINE REVERSE(I,X)
```
Subroutine REVERSE has two arguments.

```
SUBROUTINE REPORT
```
Subroutine REPORT has no arguments.

```
SUBROUTINE VENDOR(NAME,ADDRESS,NUMBER)
```
Subroutine VENDOR uses three values that are passed from the main program.

```
DIM A(4,10),B(10,5)
CALL REV(MAT A, MAT B)
```
Subroutine REV accepts two input array variables, one of size 40 and one of size 50 elements.

```
SUBROUTINE REV(MAT C, MAT B)
DIM C(4,10),B(50)
```

SUM Function

The SUM function sums the lowest level elements in a dynamic array.

Syntax

SUM(expr)

expr evaluates to dynamic array to be summed

Description

The elements at the lowest level are summed until the next higher level is encountered. For example, if the lowest delimited level is a subvalue, then all subvalues are summed within each value; when the value mark is encountered, the result is stored in that value.

The SUM function is useful to eliminate the lowest level of element present in a numeric dynamic array by combining at the next highest level. For example, if a dynamic array containing subvalues is summed, the lowest level in the returned array is value.

The SUM function can be used repeatedly until only one element remains.

```
A = 1:AM:21:VM:22:AM:311:SVM:312:VM:321:SVM:322
FOR I = 1 TO 3
  A=SUM(A)
NEXT
```

The first loop sums the subvalues to the next value delimiter (or end-of-string). The second loop sums the values to the next attribute delimiter (or end-of-string). The third loop sums the attributes.

result:

```
A = 1:AM:21:VM:22:AM:623:VM:643
A = 1:AM:43:AM:1266
A = 1310
```

SYSTEM Function

The SYSTEM function returns system information such as error status codes (generated as a result of a previous BASIC statement) or parameters such as the page-number or page-width.

Syntax

SYSTEM(expr)

expr number that indicates value to return; the values are shown in Table 3-5; the value must be in the range 0 through the maximum value as defined in the table. If it is outside the allowable range, SYSTEM returns a value as if the expression evaluated to zero (the error function).

Description

If the expression used in SYSTEM is zero, the function returns a value determined by the last executed BASIC statement that set an error condition. Examples of such BASIC statements are the tape commands such as READT and WRITET if the ELSE branch executes. SYSTEM(0), therefore, allows one to determine exactly what error has occurred when the program follows the ELSE branch of these statements. If the ELSE branch was not followed, the value returned by SYSTEM(0) is zero.

The SYSTEM function, with non-zero values of the expression, returns parameters that have been set external to the BASIC program. Table 3-5 lists the function expressions.

Table 3-5. SYSTEM Values (1 of 4)

Value	Information Returned	
0	Error function value:	
	1	Tape unit is not attached
	2	EOF read from tape unit
	3	Attempted to write null string
	4	Attempted to write variable longer than tape record length
	5	Tape unit is off-line
	6	Cartridge is not formatted correctly
	2001-2339	UltiNet error code; see *UltiNet User Guide* for specifics
1	1	PRINTER ON or (P) option used in RUN
	0	data is being printed to the terminal.
2	Current page-size (page-width in columns)	
3	Current page-depth (number of lines in page)	
4	Number of lines remaining in current page	
5	Current page-number	
6	Current line-counter (number of lines printed)	
7	One-character terminal-type code	
8	Current tape record length	
9	System serial number	

Table 3-5. SYSTEM Values (2 of 4)

Value	Information Returned
10	Code for machine BASIC program is running on: D0 LSI11-based systems without typeahead D1 LSI11-based systems with typeahead and regular memory D2 LSI11-based systems with typeahead and dual-ported memory D3 LSI11 3030 systems D4 LSI11 3040/3050 systems H0 Ultimate WCS-based system H1 Ultimate HPP-based system H2 Ultimate 7000 system H3 Ultimate 8Mb 6000 system I IBM system M 1400-based system V0 VAX 3x system V4 VAX ECP-based system
11	number of characters in typeahead buffer
12	ASCII value of terminating character of last INPUT statement; if INPUTCONTROL FUNCKEYS not set, could either be ASCII value 10 (LINEFEED) or 13 (RETURN). For a list of values returned when INPUTCONTROL FUNCKEYS is set, see Table 3-3 included with the description of INPUTCONTROL.
13	reserved
14	returns item in ERRMSG file called INPUT@; item contains messages used by system command UPDATE
15	reserved

Table 3-5. SYSTEM Values (3 of 4)

Value	Information Returned
16	cause of abort; valid only in subroutine called by TRAP ON THEN CALL statement; may be the following 0 program termination - ABORT, END, or STOP statement in program 1 BREAK key pressed 2 END command entered in BASIC or system debugger 3 OFF command entered in BASIC or system debugger 4 SET-LOGOFF has been invoked and DSR has dropped, or the process has been logged off by another process 5 CHAIN 'OFF' or EXECUTE 'OFF' statement in program Bnnn BASIC ERRMSG item.id of error that caused trap
17	file name and name of program currently being executed; if in subroutine called by TRAP ON THEN CALL, contains name of program that called subroutine
18	returns TCL statement that invoked current program; statement is formatted as dynamic array. Elements in the statement are separated by attribute marks. If an element is enclosed in delimiters, the delimiters are removed. This function is equivalent to @SENTENCE
19	port number of current process; equivalent to @USERNO
20	date format: 0 USA (mm/dd/yy) 1 European (dd/mm/yy) 2 Swedish (yy/mm/dd)

Table 3-5. SYSTEM Values (4 of 4)

Value	Information Returned
21	level of nested execute; the highest level is 0; equivalent to @EXECLEVEL
22	last hold file number; equivalent to @HOLDFILE
23	privilege level of current process; equivalent to @PRIVILEGE
24	spooler assignments; equivalent to @SPOOLOPTS
25	returns 1 if external select list is active, else returns 0; equivalent to @SELECT
26	current account name; equivalent to @WHO
27	two-digit language code of the language assigned to current port; equivalent to @LANGUAGE

TAN Function

The TAN function returns the tangent of an angle expressed in degrees.

Syntax

TAN(expr)

expr number of degrees in the angle

Description

The function generates the tangent of the angle.

```
A = 32                          Returns 0.6248, the tangent of 32°.
PRINT TAN(A)
```

TIME Function

The TIME function returns the internal time of day.

Syntax

TIME()

Description

This function returns the internal time of day, which is the number of seconds past midnight.

`A = TIME()`	Assigns current internal time to variable A.
`PRINT OCONV(TIME(),"MT")`	Prints current time in hh:mm format; for example, `09:16`

TIMEDATE Function

The TIMEDATE function returns the current time and date in external format.

Syntax

TIMEDATE()

Description

The TIMEDATE function returns the current time and date in external format, which is

hh:mm:ss dd mon yyyy

hh=hours
mm=minutes
ss=seconds
dd=day
mon=month of year
yyyy=year

```
B = TIMEDATE()
PRINT B
```

assigns to B the current time and date in external format, then prints the string; for example,

```
08:30:23  06 MAY 1990
```

TRAP ON THEN CALL Statement

The TRAP ON THEN CALL statement provides program control for certain unusual conditions.

Syntax

TRAP ON THEN CALL trap.subr

trap.subr name of external subroutine to be called if unusual condition occurs; no parameters can be passed

Description

The TRAP ON THEN CALL statement specifies the name of an external subroutine to be called if one of the following conditions occurs:

- execution of ABORT, END, STOP, CHAIN 'OFF', or EXECUTE 'OFF' statement
- BREAK key pressed; trapped only if BREAK OFF is active
- execution of END or OFF command in BASIC or system debugger
- DSR drops or line is logged off by another process
- any runtime error that would normally cause the program to enter the BASIC debugger

The trap subroutine is **not** called when the TRAP ON statement is encountered; it is called only if one of the above unusual conditions occurs.

Although no parameters can be passed between the trap subroutine and the program in which the trap occurred, values can be passed through COMMON variables. The subroutine called by the trap may decide on further action; this could be, for example, terminating, logging off, or capturing the reason for the trap in a COMMON variable and RETURNing to the main program. Within the subroutine, SYSTEM(16) can be used to determine the reason for the trap and SYSTEM(17) used to determine the name of the program that called the subroutine.

If the trap subroutine executes its RETURN statement, the path the system follows depends on the reason for the trap. Table 3-6 lists the possible values for SYSTEM(16) and where the subroutine returns.

SYSTEM(17) returns the file and program name of the program that was executing at the time of the trap; the information is in the following format:

file.name prog.name

Once a trap is set up in a program, it remains set unless a subsequent TRAP ON is encountered. The last trap that was encountered at the current EXECUTE level is the active trap. A trap is passed to any subroutine that is called after the trap is set up; a trap that is set up in a subroutine is passed back to the calling program. For example, if one trap is set up in the main program, and a second is set up in a subroutine, the trap set up in the main program is in effect until the TRAP ON statement in the subroutine is encountered. The trap set up by the subroutine remains in effect, even after the subroutine returns to the main program.

BASIC programs that are CHAINed to after a trap is set up inherit the trap if the CHAIN statement includes the I option with the verb. PROCs and system commands that are chained to do not inherit traps.

Programs that are EXECUTEd do not inherit traps. However, if a TRAP ON is set up at one level and a condition that would produce a SYSTEM(16) value of 2, 3, 4, or 5 occurs in a lower EXECUTE level where no TRAP ON is set, the system passes control back to the trap subroutine in the EXECUTE level with the TRAP ON. The SYSTEM(16) value is preserved if it is 2, 3, or 4. If the condition would produce a value of 5 if the TRAP ON were in the same level, SYSTEM(16) returns a value of 3. (For an explanation of SYSTEM(16) values, see Table 3-6.)

During the execution of the trap subroutine, the trapping mechanism is disabled; this is to prevent an error within the subroutine from causing the system to get into an unbreakable loop. However, if DSR drops or the process is logged off by another process while the trap subroutine is executing, the condition is saved and the trap subroutine is reentered with the saved condition as soon as the trap subroutine RETURNs out of the current condition.

Table 3-6. SYSTEM(16) Values

Value	Description	Path After RETURN
0	Program termination - ABORT, END, or STOP statement in program	returns to program in which trap occurred and follows through on statement.
1	BREAK key pressed when BREAK OFF is in effect	returns to program in which trap occurred
2	END command entered in BASIC or system debugger	returns to program in which trap occurred
3	OFF command entered in BASIC or system debugger	returns to program in which trap occurred
4	SET-LOGOFF has been invoked and DSR has dropped, or the process has been logged off by another process	returns to program in which trap occurred
5	CHAIN 'OFF' or EXECUTE 'OFF' statement in program	logs user off unless statement causing trap is in an EXECUTEd program and a TRAP ON has been set in a previous level. In this case, the RETURN enters the trap subroutine at that level and reports SYSTEM(16) = 3
Bnn	ERRMSG item.id of error that caused trap	returns to TCL or, if in an EXECUTEd program, to previous level

The system determines the statement to return to as follows:

- to beginning of statement that caused trap if statement not completely executed
- to statement following the location of trap if trap occurred after statement was executed

- to INPUT prompt if trap occurred while waiting for input; if it is desired to force the program to return to the statement following the INPUT, the DATA statement can be used in the trap subroutine to stack data for the INPUT statement

```
TRAP ON THEN CALL TRAP.SUBR
.
.
*                              Debugger entered
*OFF                           OFF typed in BASIC debugger
```

result:
TRAP.SUBR is called when OFF is typed with SYSTEM(16) = 3. If RETURN statement in TRAP.SUBR is executed, program returns to statement that was being executed when debugger entered.

```
TRAP ON THEN CALL TRAP.SUBR
.
.
CHAIN 'OFF'                    OFF is encountered in program.
```

result:
Trap subroutine is entered with SYSTEM(16) = 5. If RETURN statement in TRAP.SUBR is executed, user is logged off.

```
PROG1:
 TRAP ON THEN CALL TRAP.SUBR
 EXECUTE "RUN BP PROG2"
 .
 .
PROG2:                          No trap is specified in program
 EXECUTE "RUN BP PROG3"
 .
 .
PROG3:                          No trap is specified in program
 .
 .
 *                              Debugger entered
 *OFF                           OFF typed in BASIC debugger
```

result:

System returns to last level in which trap has been set and calls the trap subroutine. In this example, this is the subroutine TRAP.SUBR specified in PROG1; SYSTEM(16) = 3. If the RETURN statement in TRAP.SUBR is executed, program returns to statement following EXECUTE statement in PROG1.

TRIM Function

The TRIM function removes extraneous blank spaces from a specified string.

Syntax

TRIM{B|F}(string.expr)

B	remove only the trailing (back) spaces from a string
F	remove only leading (front) spaces from a string
string.expr	string being trimmed

Description

The TRIM function, without the B or F suffix, deletes leading and trailing blanks, as well as any multiple blanks within the expression.

The B and F suffixes cannot both be specified at the same time.

```
A="        GOOD MORNING,        MR. BRIGGS"
A=TRIM(A)
PRINT A
```

Deletes extra spaces before and within expression with the result:

```
GOOD MORNING, MR. BRIGGS
```

```
A = '   A  B C   '
A = TRIMB(A)
```

The back spaces are trimmed, with the result:

```
A = '   A  B C'
```

```
A = '   A  B C   '
A = TRIMF(A)
```

The front spaces are trimmed, with the result:

```
A = 'A  B C   '
```

UNLOCK Statement

The UNLOCK statement releases the specified execution locks set by the LOCK statement.

Syntax

UNLOCK {expr}

expr integer between 0 and 47 that specifies lock to be released (cleared); if expr is omitted, all execution locks that were previously set by the program are released. If a number greater than 47 is specified, it is adjusted by mod 48 to a number less than 47

Description

The UNLOCK statement operates in conjunction with the LOCK statement, which sets an execution lock, to provide a file and execution lock capability for BASIC programs. (Please refer to the LOCK statement, listed alphabetically in this chapter.)

Execution locks may be used as file locks to prevent multiple BASIC programs from updating the same files simultaneously. The Ultimate system provides 48 execution locks numbered from 0 through 47.

An execution lock can be unlocked only by the program that locked it. An attempt to unlock an execution lock that the program did not lock has no effect.

All execution locks set by a program are automatically released upon termination of the program, even if it the program is terminated by the END command from the BASIC debugger.

`UNLOCK 47`	Resets execution lock 47.
`UNLOCK`	Resets all execution locks previously set by the program.

UNTIL Statement

The UNTIL statement is an optional statement within the FOR/NEXT or LOOP statement sequences.

Please refer to the FOR statement or the LOOP statement for information about the entire statement sequence.

USERTEXT Function

The USERTEXT function returns the text of a specified USERMSG file item.

Syntax

USERTEXT(item.id{,param1{, ... }})

item.id the item.id of the USERMSG file item to be returned.

param parameters to be passed to the USERMSG item.

Description

The USERMSG file follows the same the format as the ERRMSG file, but the items are created and maintained by the users. The USERMSG file is described in Appendix F, USERMSG file.

The USERMSG file can contain multi-level data files, where each data file is in a different language. The system command SET-LANGUAGE is used to specify the particular data level that is used. The system variable @LANGUAGE or system function SYSTEM(27) can be used to determine the current language settings. @LANGUAGE is described in the section System Variables in Chapter 2; SYSTEM(27) is described with the SYSTEM function listed alphabetically in this chapter.

The USERTEXT function formats the message appropriately for each language.

For information on printing messages from the ERRMSG file, see the ERRTEXT function, listed alphabetically in this chapter.

```
X = USERTEXT('cmsg1',CUSTID,CUSTNAME)
```

This returns the text of "cmsg1" after inserting the values from CUSTID and CUSTNAME into the message.

WEOF Statement

The WEOF statement writes an end-of-file (EOF) mark on the tape unit specified by the most recent T-ATT command.

Syntax

WEOF {THEN stmts} {ELSE stmts}

THEN stmts — statements to be executed after EOF marks are successfully written; the THEN statements may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multiple line IF statement

ELSE stmts — statements to be executed if EOF marks cannot be written; the reason for the problem is returned in SYSTEM(0). The ELSE statements may appear on one line separated by semicolons or on multiple lines terminated by an END, as in the multiple line IF statement.

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

Description

The WEOF statement writes two EOF marks on the magnetic tape on the current unit, then backspaces over the second one. This correctly positions the tape for subsequent WRITET operations. The THEN statements, if any, are then executed.

The tape unit must have previously been attached before issuing this command. If the tape unit has not been attached, the ELSE statements, if any, are executed, and the system function SYSTEM(0) returns a value of 5 (tape off line) or 6 (cartridge not formatted correctly for this operating system revision). (Please refer to the SYSTEM function, listed alphabetically in this chapter.)

If the tape unit is inadvertently set off line, the system detects the condition and allows the user to correct it and proceed.When a subsequent tape instruction is processed, the system displays:

```
Tape drive off line (C)ontinue/(Q)uit:
```

If C is entered, the system returns to the BASIC program and the tape instruction is re-executed. If Q is entered, the BASIC program is aborted and control returns to TCL. Thus, the ELSE statements are not executed in either case, and the BASIC program has no way to detect such adverse action.

Caution! *The tape drive should* ***never*** *be put off line while it is running under the control of any tape operation. By doing so, the tape drive may lose its momentum and the tape read/write head may not be aligned with the current data block on tape. Even though the system allows you to (C)ontinue, it is not guaranteed that valid data is then read or written.*

`WEOF ELSE`
`  GOTO TPERR`
`END`
	Writes two EOF marks, then backspaces over the second one; if EOF marks cannot be written, control is transferred to TPERR routine.

WHILE Statement

The WHILE statement is an optional statement within a FOR/NEXT or LOOP statement sequence.

Please refer to the FOR statement or LOOP statement for information about the entire statement sequence.

WRITE{U} Statement

The WRITE{U} statement is used to update a file item.

Syntax

WRITE{U} expr ON/TO {file.var, item.id {ON ERROR stmts}

U	if specified, locks item; after completing the write operation, WRITEU does not unlock the item lock; if not specified, WRITE unlocks the item if it was initially locked. (The letter U is appended to the statement name to imply **update**, not **unlock**.)
expr	information to be written
ON/TO	ON and TO are equivalent; either may be specified
file.var	variable to which file in which item is to be written was previously OPENed; if omitted, the internal default file variable is used (that is, the file most recently opened without a file variable)
item.id	name of item to be written; if it currently exists, its contents are replaced by expr; if the item does not exist, a new item is created
ON ERROR stmts	statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be written to due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information about remote files, refer to the *UltiNet User's Guide*.) The ON ERROR clause has no effect when releasing local files. The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.

Description

Item locks are intended to be used to prevent simultaneous updates of the same item by more than one program.

If the item is not locked before the WRITEU statement is executed, it is locked afterwards. For more information on item locks, please see the READ, READV, and MATREAD statements, listed alphabetically in this chapter.

Note: *The RELEASE statement can also be used to unlock the item. The DELETE statement also unlocks the item. (Please refer to the DELETE and RELEASE statements, listed alphabetically in this chapter.)*

The user should note that the BASIC program will abort with an appropriate error message if the specified file has not been opened prior to the execution of the WRITE statement. (Refer to run-time error messages in Appendix B.)

UltiNet Considerations

The ON ERROR clause allows the program to retrieve the UltiNet error number and take appropriate action. Such action could, for instance, include printing the associated message text via a PUT statement or STOP statement, and resuming or terminating program execution.

If items in a remote file cannot be written due to network errors and no ON ERROR clause is present, the program terminates with an error message.

```
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".

```
A="123456789"
B="X55"
WRITE A TO FN1,B
```

Equivalent to example above; uses TO instead of ON.

```
WRITE 100*5 ON "EXP"
```

Replaces the current contents of item EXP in the file opened without a file variable with string value "500".

```
WRITEU CUST.NAME ON CUST.FILE,ID
```

Replaces the current contents of the item specified by variable ID (in the file opened and assigned to variable CUST.FILE) with the contents of CUST.NAME. Does not unlock the item.

```
WRITEU CUST.NAME ON CUST.FILE,ID ON ERROR
  ERRNUM = SYSTEM(0)
  GOTO PROCESSERR
  END
```

Writes as above, or branches to local routine to process UltiNet error number.

WRITET{X} Statement

The WRITET{X} statement writes a record to tape. The tape unit and record length (block size) on the tape is as specified by the most recent T-ATT command.

Syntax

WRITET{X} expr THEN/ELSE stmts

X	indicates to convert data from hexadecimal to binary format; this feature is intended to allow characters such as segment marks to be written by BASIC programs.
expr	information to be written to the next record of the current tape unit; if the length of the string is less than the current tape block size, the tape block is padded with trailing blanks; if the length of the string is greater than the current block size, the string is truncated to the current block size and trailing characters in the string are not written; in this case a warning message is printed on the terminal, and SYSTEM(0) returns a value of 4.
THEN stmts	statements to be executed after record is successfully written; the THEN statements may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multiple line IF statement.
ELSE stmts	statements to be executed if expr is null or tape is not attached; the ELSE statements may appear on one line separated by semicolons or on multiple lines terminated by an END, as in the multiple line IF statement. If the value to be written is null (""), SYSTEM(0) returns a value of 3. If the tape has not been attached, SYSTEM(0) returns a value of 5.

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

Description

The WRITETX form assumes all characters in the variable are hexadecimal. If any non-hexadecimal characters are encountered, they are ignored. Data integrity is the responsibility of the programmer.

For information on reading binary data from tape and converting it to hexadecimal, see the description of READT.

The tape unit must have been attached before this command is issued. If the tape unit has not been attached, the ELSE statements, if any, are executed, and the system function SYSTEM(0) returns a value of 5 (tape off line) or 6 (cartridge not formatted correctly for this operating system revision). (Please refer to the SYSTEM function, listed alphabetically in this chapter.)

If the tape unit is inadvertently set off line after the first tape instruction, the system detects the condition and allows the user to correct it and proceed. When a subsequent tape instruction is processed, the system displays:

```
Tape drive off line (C)ontinue/(Q)uit:
```

If C is entered, the system returns to the BASIC program and the tape instruction is re-executed. If Q is entered, the BASIC program is aborted and control returns to TCL. Thus, the ELSE statements are not executed in either case, and the BASIC program has no way to detect such adverse action.

Caution! *The tape drive should* ***never*** *be put off line while it is running under the control of any tape operation. By doing so, the tape drive may lose its momentum and the tape read/write head may not be aligned with the current data block on tape. Even though the system allows you to (C)ontinue, it is not guaranteed that valid data is then read or written.*

```
FOR I=1 TO 5
WRITET A(I) ELSE STOP
NEXT I
```

The values of array elements A(1) through A(5) are written onto five tape records. If one of the array elements is null or if the tape unit is not attached, the program terminates.

```
WRITETX '31323334FF414243' ELSE STOP
```

The following string, where _ represents a segment mark, is written to tape:

1234_ABC

WRITEV{U} Statement

The WRITEV{U} statement is used to write a single attribute value to an item in a file.

Syntax

WRITEV{U} expr ON {file.var,}item.id,attrib.no{ON ERROR stmts}

U	if specified, locks item; after completing the write operation, WRITEVU does not unlock the item lock; if not specified, WRITEV unlocks the item if it was initially locked. (The letter U is appended to the statement name to imply **update**, not **unlock**.)
expr	information to be written
file.var	variable to which file in which item is to be written was previously OPENed; if omitted, the internal default file variable is used (that is, the file most recently opened without a file variable)
item.id	name of item to be written; if it currently exists, its contents are replaced by expr; if the item does not exist, a new item is created
attrib.no	attribute number to write; if attrib.no is 0, the attribute is inserted at the beginning of the item as attribute 1 and all existing attributes in the item are shifted by one; if attrib.no is -1, the attribute is appended to the end of the item and all existing attributes are undisturbed; if attrib.no is greater than the existing number of attributes, a new attribute is created and all attributes between it and the former last attribute will be null
ON ERROR stmts	statements to be executed if the file is a remote file, that is accessed via UltiNet, and it cannot be written to due to a network error condition. In this case, the value of SYSTEM(0) indicates the UltiNet error number. (Refer to the SYSTEM function, listed alphabetically in this chapter; for more information

about remote files, refer to the *UltiNet User's Guide.*) The ON ERROR clause has no effect when releasing local files.

The statements may be on a single line or on multiple lines. If multiple lines are used, the clause must be terminated by an END statement as in the multi-line IF statement.

Description

Item locks are intended to be used to prevent simultaneous updates of the same item by more than one program.

The letter U is appended to the statement name to imply **update**, not **unlock**. It is intended for master file updates when several transactions are being processed and an update of the master file item is made following each transaction update.

If the item is not locked before the WRITEVU statement is executed, it is locked afterwards. For more information on item locks, please see the READ, READV, and MATREAD statements, listed alphabetically in this chapter.

Note: *The RELEASE statement can also be used to unlock the item. (Please refer to the RELEASE statement, listed alphabetically in this chapter.)*

The BASIC program aborts with an appropriate error message if the specified file has not been opened prior to the execution of the WRITEV statement.

UltiNet Considerations

The ON ERROR clause allows the program to retrieve the UltiNet error number and take appropriate action. Such action could, for instance, include printing the associated message text via a PUT statement or STOP statement, and resuming or terminating program execution.

If items in a remote file cannot be written due to network errors and no ON ERROR clause is present, the program terminates with an error message.

```
Y="THIS IS A TEST"
WRITEV Y ON X,"PROG",0
```

The string value "THIS IS A TEST" is inserted prior to the first attribute of item PROG in the file opened and assigned to variable X.

```
WRITEV "XYZ" ON "A7",4
```

Attribute 4 of item A7 in the file opened without a file variable is replaced by string value "XYZ".

```
WRITEV "XYZ" ON "A7",4 ON ERROR
   ERRNUM=SYSTEM(0)
   GOSUB PROCESSERR
   GOTO TOP
END
```

Writes as above, or retrieves error number and performs local subroutine on UltiNet error number.

```
WRITEVU CUST.NAME ON CUST.FILE,ID,3
```

Replaces the third attribute of item ID in the file opened and assigned to variable CUST.FILE with the contents of variable CUST.NAME. Does not unlock the item.

4 BASIC Debugger

The BASIC debugger facilitates the debugging of new BASIC programs and the maintenance of existing BASIC programs.

The BASIC debugger has the following general capabilities:

- controlled stepping through execution of program by way of single or multiple steps
- transferring control to a specified step (line number)
- breaking (temporary halting) of execution on specified line numbers, on the satisfaction of specified logical conditions, on entering or returning from external subroutines
- displaying and/or changing any variables, including dimensioned and dynamic array variables and named COMMON variables
- displaying values of variables in hexadecimal
- tracing variables
- conditional entry to the system debugger
- directing output (terminal/printer)
- displaying of specified (or all) source code lines
- help facility, which lists all BASIC debugger commands
- paging, which stops display at end of each screen page

The BASIC debugger prompt is an asterisk (*).

Entering the Debugger

The BASIC debugger may be entered at execution time as follows:

- pressing the BREAK key from within a program
- using the 'D' (debug) option with the RUN verb
- automatically under some error conditions

The BASIC debugger may be re-entered at execution time as follows:

- satisfying breakpoint conditions
- executing specified number of lines
- entering or returning from external subroutine when specified

When the BASIC debugger is entered, it displays a code that gives the reason for the entry into the debugger, the program name, and the source code line number about to be executed. The following codes may be displayed:

A	run time abort
B	breakpoint encountered
C	CALL statement breakpoint executed
E	execution breakpoint
I	interrupt (BREAK key) detected
R	RETURN statement breakpoint executed

If the S debugger command (display source code) is off when the debugger is entered, the debugger displays the code, the program name, and the source code line number about to be executed. If the S command is on, the debugger displays the current source code line. For example, if the debugger is entered and the S command is off, a line similar to the following is displayed:

```
*I BP PGRM, Line # 23
```

If the debugger is entered and the S command is on, a line similar to the following is displayed:

```
023 INPUT ANS
```

The reason the debugger was entered can be displayed by the O command, which is listed alphabetically in this section.

Compiler Restrictions

Two options available with the BASIC and COMPILE verbs should not be used if the debugger is to be used subsequently with the program. These options are C and S.

Line numbers cannot be referenced in programs compiled with the C option, which inhibits saving end-of-line markers in object code; in this case, the BASIC debugger views the program as one single line.

Variables cannot be referenced in programs compiled with the S option, which inhibits saving the symbol table. The symbol table contains all variable names defined in a program and is used to reference variables. It is automatically stored with the object code when the program is compiled (unless suppressed by the 'S' option). The BASIC debugger can retrieve the value of a variable from the symbol table by use of the / (list) command. (For details, refer to the / Command).

Neither variables nor line numbers can be referenced in a program that has a $NODEBUG directive, which has the same effect as the C and S options together.

Summary of Debugger Commands

Table 4-1 is a summary of the BASIC debugger commands:summary. When specifying a debugger command, there must not be any spaces between any elements. The following sections describe in detail the commands and their use.

SYS2 privileges are required for all commands other than G, END, OFF, and P. This prevents users from making unauthorized changes to data during reporting and data entry.

Table 4-1. BASIC Debugger Commands (1 of 2)

Command	Description
Bvoc{&voc} B$on	Set breakpoint on logical condition where v is variable o is logical operator <,>,=,# c is condition to meet n is line number when preceded by B$o
BYE	End program execution and return to TCL
C	Toggle CALL/RETURN breakpoint mode
D	Display breakpoint table
DE{BUG}	Escape to system debugger
E{n}	Single/multiple step execution
END	End program execution and return to TCL
G{n}	Continue program execution at specified line
H	Display list of debugger commands (help)
HX	Display variable values in hexadecimal
K{n} K{{/}var}	Remove breakpoints
L{{m-}n}	Display specified source code current lines
L*	Display all source code lines
LP	Toggle output between terminal and printer
N	Clear debug entry delay counter
Nn	Bypass 'n' breakpoints/steps before reentering debugger
O	Display current debugger options
OFF	Logoff

Table 4-1. BASIC Debugger Commands (2 of 2)

Command	Description
P	Inhibit/enable output
PC	Printer-close output to spooler
R	Display GOSUB return stack
S	Toggle display of source code lines and line numbers
STOP	End program execution; if executed from PROC, returns to PROC
T	Turn trace table on/off
T{/}v	Trace specified variable 'v'
U{n} U{{/}v}	Remove traces
V	Display current program name and line number; verify object code
Z	Request source (if not in same file/same name)
$,*, or ?	Display current program name and line number; verify object code
/m	Display value of variable or if dimensioned array, entire array
/m(x{,y})	Display value of element in array
/m<a{,v{,s}}>	Display value of element in dynamic array
/*	Display entire symbol table
[x,y]	Display specified substring
[	Display entire string

B Command - Set Breakpoints

The B command sets breakpoints, where breakpoints are conditions that cause a break in program execution. Breakpoints are kept in a breakpoint table.

Syntax

Bvoc{&voc}
B$ operator line.no {&voc}

var	variable or array element to be tested
operator	may be one of the following: < less than > greater than = equal to # not equal to
condition	constant, variable, or array element to test for; string constants must be enclosed in quotes using the same rules that apply to BASIC literals; only individual array elements may be specified.
&	logical connector between expressions (AND)
$	the breakpoint is to be line number condition
line.no	source program line number to be tested

Description

Each program or external subroutine may set up to eight breakpoint conditions, any one of which will, when satisfied, cause a break in execution. Breakpoints 1 through 4 are reserved for specifying conditions for non-COMMON variables, named COMMON variables, and line numbers. Breakpoints 5 through 8 are reserved for specifying conditions for COMMON variables. The debugger assigns the entry numbers.

Breakpoints 5 through 8 are passed between subroutines and calling programs.

Breakpoints set for named COMMON variables are **not** placed in the table for COMMON variables; they are placed in the table along with local variables and are treated as local variables by the debugger.

If the breakpoint has been accepted, the debugger displays a plus sign and the breakpoint number:

```
*BA>0 +1
*BTAX<2 +5
```

If the breakpoint condition is met during program execution, an execution break occurs. The debugger halts the program and, if the S command is off, displays a message similar to the following:

```
*Bn file.name prog.name, Line # m
```

where

n	number of the breakpoint table entry that caused the break
file.name	file that contains program
prog.name	name of program in which break occurred
m	program line number that caused the break

If the S command is on, the debugger displays the source code line, similar to the following:

```
003 TAX = 10
```

For more information on the S command, see the description of the command listed alphabetically in this section.

The current breakpoints can be viewed by using the D command.

`*BTAX=500`	Indicates that an execution break should occur when the value of TAX is equal to 500.
`*B$>15&X=3`	Causes program to break when the line number is greater than 15 and X is equal to 3.
`*BPRICE(3)=24.98`	Sets a break condition to halt execution when the third element of the array PRICE is equal to 24.98.

BYE Command - Return to TCL

The BYE command terminates program execution and returns the user to TCL, even if the program was executed from a PROC.

Syntax

BYE

Description

The END and STOP commands can also be used to exit the debugger. If the program was executed by a PROC, the STOP command returns to the PROC, rather than TCL. For more information, see the STOP command, listed alphabetically in this chapter.

C Command - Toggle CALL/RETURN Breakpoint

The C command is used to set and clear breakpoints for CALL and RETURN statements.

Syntax

C

Description

When the C command is set, an execution break occurs if a subroutine is called or if the subroutine RETURN statement is executed. When the program is called, the debugger is entered at the SUBROUTINE statement. When the subroutine RETURN statement is executed, the debugger is entered at the statement following the CALL statement.

If the C command is already set, entering C again causes it to be cleared. After the command is entered, the debugger displays 'on' or 'off' to indicate the new setting.

The current setting of the the C command can be viewed by the O command, which is listed alphabetically in this section.

```
*C on                          System responds 'on'
  .
  .
*C off                         System responds 'off'
```

D Command - Display Tables

The D command is used to display the trace and breakpoint tables.

Syntax

D

Description

The D command displays all trace and breakpoint tables for the currently executing program or external subroutine.

If there are no entries in the table, the following message is displayed:

```
[B505] Trace and Breakpoint tables are empty.
```

Sample output from a D command:

```
*D
Trace   Variable
   1    NAME
   2    LINE
   3    ADDR

Break   Condition
   1    $ = 356
   2    ADDR > 100
```

DE{BUG} command - Enter System Debugger

The DEBUG command passes control to the system debugger. To return to the BASIC debugger, use the system debugger END, G, or linefeed commands. (For more information about the system debugger, please refer to the *Ultimate Assembly Language Reference Guide*.)

E Command- Set Lines to Execute

The E command specifies the number of program lines to be executed when the program is resumed. After the specified number of lines is executed, an execution break occurs.

Syntax

E{n}

n number of lines to execute

Description

The form E with no parameters turns off the E function so that when the program resumes, it continues until interrupted by the user or another breakpoint, or until the program ends.

When an execution break occurs, the debugger displays a program name and line number, similar to the following:

```
E file.name prog.name, line # n
```

The E command is overridden if any entry in the breakpoint table is satisfied; for example, if E specifies to execute six lines, but a breakpoint is encountered after three lines, the program breaks at the breakpoint.

The current value of E can be displayed by using the O command.

***E** off	Turns the E function off
***E1**	Only one line of the program is executed at a time.
***E4**	Four lines of the program are executed before an execution break returns control to the user.

END Command - Return to TCL

The END command terminates program execution and returns the user to TCL, even if the program was executed from a PROC.

Syntax

END

Description

The BYE command and the STOP command can also be used to exit the debugger. If the program was executed by a PROC, the STOP command returns to the PROC, rather than TCL. For more information, see the STOP command, listed alphabetically in this chapter.

G Command- Resume Execution of Program

The G command is used to resume execution of the program.

Syntax

G{n}

n line number of source program at which to resume execution (go to line n)

Description

The form G with no parameters specifies that execution is to resume with the very next line in the program.

Note: *To resume execution at the next program line, you can press LINEFEED or the down arrow key instead of entering G.*

If the line number specified is greater than the number of lines within the program, the following message is displayed.

```
[B507] Invalid line number.
```

If the program entered the debugger because of a fatal runtime error, the program cannot be resumed if G with no line number is entered. If G is specified, the following message is displayed:

```
[B506] You cannot continue execution of a program
after a fatal abort !
```

The program **may** resume execution if G with a line number is entered; however, the results are not dependable.

`*G`	Program execution is resumed at the very next line in the program.
`*G37`	Execution is resumed at line 37 of the program.

H Command - Help

The H command displays a list and short description of all the BASIC debugger commands, similar to the following:

```
?,$,*, V        - show filename, program name, line#, verify object.
/var{(r,c)}{<a,v,s>}  - display (and alter) var; var = * displays all.
[{m,n}          - set/reset substring range
Bvoc{&voc}      - B{reakpoint set}.
C               - C{ALL/RETURN breakpoint on/off}.
D               - D{isplay trace and breakpoint table}.
DE              - DE{bug enter}.
E{n}            - E{xecution step set/reset}.
G{n}            - G{oto line# or resume processing}.
H               - H{elp}. (This message).
HX              - H{e}X {display on/off}.
K{n}{var}       - K{ill breakpoint entry or var}.
L{{m-}n}{*}     - L{ist source}.
LP              - L{ine }P{rinter on/off}.
N{n}            - N{umber of breakpoints to step through set/reset}.
O               - O{ptions display}.
P               - P{rint output display on/off}.
PC              - P{rinter }C{lose}.
R               - R{eturn stack display}.
S               - S{ource code display on/off}.
T{var}          - T{race var} or T{race on/off}.
U{n}{var}       - U{ntrace entry or var}.
Z               - {Set up source code pointers}.
END, BYE, STOP  - Terminate program (STOP resumes PROC processing
OFF             - Terminate program and log off.
```

HX - Display in Hexadecimal Format

The HX command toggles the display of values between hexadecimal and ASCII character format.

Syntax

HX

Description

When the hexadecimal display is on, values of variables are displayed as hexadecimal characters. All other displays continue to be in ASCII character format.

After the command is entered, the debugger displays 'on' or 'off' to indicate the new setting.

The current setting of the the HX command can be viewed by the O command, which is listed alphabetically in this section.

`*HX on`	Turns on hexadecimal mode
`*/A->4649525354`	Displays value of A in hex
`*HX off`	Turns off hexadecimal mode
`*/A->FIRST`	Displays value of A as ASCII characters

K Command - Breakpoint Table

The K command removes (kills) breakpoints.

Syntax

K{n}
K{/}var

n number of entry in breakpoint table to delete; n must be in the range 1-8; all other breakpoints remain the same

var variable that has breakpoint set; if more than one breakpoint is set for the variable, the first one in the table is removed

Description

The form K with no parameters deletes all breakpoint conditions in both the local and the COMMON breakpoint tables.

A minus sign and the breakpoint number are printed next to the command to indicate that the entry has been removed.

The current breakpoints can be viewed by using the D command.

```
*K2   -2                    Removes the second breakpoint
                            condition.

*KA   -1                    Removes the breakpoint set for
                            variable A.

*K                          Removes all breakpoint conditions.
Breakpoint table cleared.
```

L Command - Displaying Source Code

The L command displays one or more lines of source code from the same file as the object code program.

Syntax

```
L{{m-}n}} {*}
```

m- display line number range specified by m-n

n display line number specified by n

* display all lines in the source program

If the form L is used, only the current line is displayed.

`*L`	Displays current line of source program.
`*L1-10`	Displays first ten lines of source program.
`*L*`	Displays entire program

LP Command - Printer Output

The LP command, which is similar to a PRINTER ON command in BASIC, sends all output to the printer. Subsequent LP commands toggle this function and the words 'Off' or 'On' are printed next to the command.

N Command - Bypass Breakpoints

The N command causes the BASIC debugger to proceed through a specified number of execution breaks before returning control to the user.

Syntax

N{n}

n number of execution breaks to bypass

Description

The form N resets this function so that control is passed to the user on every execution break.

Variables that are being traced are still printed at each breakpoint.

The current value of N can be displayed by using the O command.

***N** off	Returns debugger to the single execution break mode (normal mode).
***N2**	Bypasses two execution breaks before returning to the debugger.

O Command - Display Options

The O command displays the current settings for debugger options.

Syntax

O

Description

The option settings are displayed for the following debugger commands:

E	number of lines to execute
N	number of breaks to skip
[]	subtstring display
HX	hexadecimal display on/off
S	source code display on/off
C	CALL/RETURN breakpoint on/off
code	reason for entry into debugger

```
*O
E Count ..... 1
N Count ..... 0
[] Start,Len ..... 0,0
Hex Display ..... off
Source Display ..... off
CALL/RETURN Trap ... on
Reason Code ..... E
```

OFF Command - Log Off

The OFF command terminates program execution and logs the user off the system.

Syntax

OFF

P Command - Suppress Program Output

The P command suppresses all output from the program to the terminal so that the user may look at only the debugger output. Subsequent P commands toggle this function and the word 'Off' or 'On' is printed next to the command.

Syntax

P

PC Command - Close Printer

The PC command is the same as the PRINTER CLOSE command in BASIC. Normal printer output is held until the program finishes execution, but by using the 'PC' command, the user forces printing of data that is waiting to be output.

Syntax

PC

R Command - Display GOSUB Return Stack

The R command displays the GOSUB return stack, which contains the source code line numbers for the GOSUB statements that are currently active.

Syntax

R

Description

If the S command (display source code) has been specified, the source code line, as well as line number, is displayed. (For more information on the S command, see the description of the command listed alphabetically in this section.)

If no GOSUBs are currently active, the debugger displays the following message:

```
[541]  There are no GOSUBs in the return stack.
```

The current setting of the the S command can be viewed by the O command, which is listed alphabetically in this section.

```
*R                                  Display when S command is off
Return Stack
14
11
7

*S on
*R                                  Display when S command is on
Return Stack
014 GOSUB 30   ;*General open
011 GOSUB 20   ;*Open files
007 GOSUB 10   ;*Initialization
```

S Command - Display Source Code Lines

The S command toggles the display of the source code program name and line numbers, and the source code lines.

Syntax

S

Description

The S command affects the display of the R command and the display when the debugger is re-entered.

After the command is entered, the debugger displays 'on' or 'off' to indicate the new setting.

The current setting of S can be displayed by using the O command.

```
*S off                          Turns off source line display
*G
  .
  .
*D BP PROG, Line # 23           Display when S command is off

*S on                           Turns on source line display
*G
  .
  .
023 A(1) = SZ                   Display when S command is on
```

STOP Command - Exit Debugger

The STOP command exits the debugger and terminates program execution.

Syntax

STOP

Description

If the program was executed by a PROC, the STOP command returns to the PROC, rather than TCL.

To terminate PROCs as well as the BASIC program and return to TCL, see the BYE and END commands, listed alphabetically in this chapter.

T Command - Set Trace Table

The T command specifies a variable for the trace table.

Syntax

T{{/}}var

/ not required, provided for compatibility with system debugger

var variable whose value is to be printed out at each execution break; if var is dimensioned, all elements of array are displayed

Description

The trace table is used for the automatic printout of a specified variable or variables after a break has occurred. Each program and external subroutine has its own trace and breakpoint tables. This allows the programmer to set up different break points and/or variable traces for different subroutines.

The values of the variables are printed whenever one of the following conditions exists:

- the BREAK key is pressed
- a breakpoint set by the B command is encountered
- n statements as specified by the E command have been executed

The form T with no parameters turns display of trace off if it was on, or turns display of trace on if it was off. The word 'ON' or 'OFF' is displayed to indicate the current status of the trace display.

If a valid T command is specified, the debugger displays a plus sign and the trace table entry number next to the command. If the variable does not exist or the wrong symbol table is assigned, the following message is displayed:

```
[B510] Symbol not in SYMBOL TABLE.
```

Up to six variables may be entered in the trace table associated with each program or external subroutine.

The current trace table entries can be viewed by using the D command.

***TNAME** +1	Sets a trace for variable 'NAME'.
***TDA(2)** +2	Sets a trace for the second element of the array DA.
***T** off	If on, turns trace off.
***T** on	If off, turns trace on.

U Command - Delete Traces

The U command is used to delete variables from the trace table.

Syntax

U{{/}var}
U{n}

/ not required, provided for compatibility with system debugger

var variable to be deleted from the trace table

n trace table entry number

Description

The form U with no parameters deletes the entire trace table.

A minus sign is printed next to the command to indicate that an entry has been removed.

The current trace table entries can be viewed by using the D command.

***UNAME** -1	Deletes the variable NAME from the trace table.
***U3** -3	Deletes the third entry from the trace table.
***U** Trace table cleared.	Deletes all variables from the trace table.

V Command - Verify Object Code

The V command performs a checksum calculation to verify the integrity of the object code for the current source program line.

Syntax

V

Description

This command is similar to the ?, *, and the $ commands, which are described in this chapter following the alphabetical listings. $

The V commands displays information similar to the following:

*V file.name prog.name, Line # m, Object verifies

where

file.name	file that contains program
prog.name	name of program currently being executed
m	program line number that is being verified

```
*V  BP PROG, Line # 3,  Object verifies
```

Z Command - Displaying Source Code

The Z command allows the user to display the source code to the BASIC program when the source is located outside the object code program file or is stored under a different program name.

Syntax

Z {file.name prog.name}

file.name file that contains program with source program

prog.name name of program

Description

If the file name and program name are not specified with the Z command, the BASIC debugger displays the following message:

File/prog name?

Enter the file name and program name (item.id), separated by a blank. If the file and program are found, the BASIC debugger returns with a prompt (*).

If the file is not found, the following message is displayed:

```
[503] Not a valid file name.
```

If the program is not found, the following message is displayed:

```
[504] Not a valid program name.
```

If either the file name or program name is not specified, or the filename is invalid, the "File/prog name?" prompt is repeated until a valid file name and program name are entered, or RETURN is pressed, aborting the Z command.

/ Command - Displaying and Changing Variables

The / (list) command can be used to display or change variables and arrays during program execution.

Syntax

/var
/*

var name of variable to display or change

* display all variables; no changing of the values is permitted

Description

The list command can be used as follows:

- display or change a simple variable, dimensioned or dynamic array, explicitly stated array element, or a COMMON variable
- if a single element of a dimensioned array is listed, subsequent elements can be displayed by pressing the line-feed key (down arrow key on some terminals)
- change an element to a null value by pressing <CTRL-_>
- list named COMMON and variable dimensioned array elements; however, they must be initialized by the runtime processor before they can be listed in the debugger
- display unprintable characters (that is, characters with ASCII values less than 32) as a dot (.)

If the variable is found, the value is displayed and the user is prompted with an equals (=) sign. A new value may then be entered. Values are entered as strings, but without surrounding quotes. A carriage return with no input causes the variable to retain its current value. To change an element to a null value, press <CTRL-_>.

If an array is specified, each element is displayed until all elements are exhausted or until the BREAK key is pressed.

If the specified variable is unassigned, the message 'var unassigned' is displayed immediately following the command.

If the variable does not exist, the following message is displayed:

```
[510] Symbol not in SYMBOL TABLE.
```

The listing stops at the end of each screen page. To continue, press any key. To return to the debugger prompt, press <CTRL-x>.

***/NAME**->Jones=	Displays the value of the variable NAME.
***/Y(3)**->7879=	Displays the value of the third element of the array Y.
***/X<2,3>**->16=	Displays the contents of the second attribute, third value of the dynamic array X .
***/Z(2)**->27=	Displays third element of array Z; to display the value of each subsequent element of the array Z, press the LINE-FEED or down arrow key.
***/DC**(var unassigned)->0=	Value has not been assigned; zero is assumed.
/	Displays the values of all variables in the program. No changing of these values is permitted. The display pauses at the bottom of each screen; press any key to continue.

?, *, and $ Command - Verify Object Code

The ?, *, and the $ commands all display the current program or external subroutine name and current line number, and perform a checksum calculation to verify the integrity of the object code.

Syntax

```
?
*
$
```

Description

These commands are equivalent to the V command, which is listed alphabetically in this chapter.

```
*?   BP COUNT, Line # 7, Object verifies
```

[] Command- Specify Substring to Display

The [] (substring) command specifies the substring to which all variables printed by the BASIC debugger are limited.

Syntax

[{start,len]}

start starting character position

len number of characters.

Description

The [(left bracket only) command resets the effect of a previous [] command, causing values of variables to be printed in their entirety.

Setting len to 0 has the same effect as entering a [(left bracket).

The current values of [] can be displayed by using the O command.

`*[1,5]`	Displays first five characters of all strings.
`*[ off`	Turns off string window.

Example of Using the BASIC Debugger

This section shows a sample of using the BASIC debugger.

The following sample program called TEST3 is used:

```
001       A=123.7891
002       B='THIS IS A STRING'
003       DIM X(3)
004       X(1)=123456
005       X(2)='HELLO THERE'
006       X(3)=0
007       PRINT A,B
008       PRINT X(1),X(2),X(3)
009       END
```

In the following dialogue, the text in boldface is entered by the user; <CR> is indicated only when it is the only entry; however, <CR> must be pressed after every entry.

Dialogue	Explanation
:RUN BP TEST3 (D)	Run program with 'D' option to break before first line is executed.
D BP TEST3, line # 1	Indicates execution halted before line 1.
*	Debugger prompt
***/X**	Display array X
X(1)(var unassigned)->0= **<CR>** X(2)(var unassigned)->0= **<CR>** X(3)(var unassigned)->0= **<CR>** *	Values are unassigned because no lines have been executed; the = indicates value can be changed; press <CR> to leave as is. After all array elements have been displayed, returns to prompt.
***B$=5** +1	Breakpoint 1 set, break if line number is 5.
***G**	Go
B1 BP TEST3, Line # 5	Break condition 1 satisfied; about to execute line 5.
***/X(1)**123456=**<CR>**	Display X(1). Leave unchanged.
***TX(2)** +1	Trace X(2). Print at each break.
***E1**	Set single step. Break at each statement.
***G**	Go
E BP TEST3, Line # 6	Execution break caused by E; about to execute line 6.
X(2)->HELLO THERE	X(2) displayed by trace.
***G**	Go

Dialogue, cont.	Explanation
`E BP TEST3, Line # 7`	Execution break caused by E1; about to execute line 7.
`X(2)->HELLO THERE`	X(2) displayed by trace.
`*E off`	Set execution to normal mode. E off.
`*$ BP TEST3 Line # 7` `Object verifies`	Displays file and program name, verifies object, about to execute line 7
`*/A->123.7891=<CR>`	Display variable A. Leave unchanged.
`*P on`	Turn terminal print on.
`*B$=10 +`	Break when line number is 10.

`*D`
`Trace Variable`
`1 X(2)`
 	
`Break Condition`
`1 $ = 5`
`2 $ = 10`
	Display trace and break tables

`*K1 -1`	Kill first break condition ($=5).
`*/A 123.7891=356.71`	Display variable A; change to 345.71.
`*/A 356.71=<CR>`	Display variable A. Leave unchanged.
`*END`	End execution of program and return to TCL.

5 Programmer's Reference

This chapter contains hints and recommendations for programmers using the Ultimate BASIC language. The information includes

- Understanding the Ultimate System File Structure
- Programming Techniques for Handling I/O
- Programming Considerations about I/O for Network Users
- Programming Techniques for Handling File Items
- Techniques for Cursor Positioning
- Programming for Maximum System Performance
- Programming Examples

Understanding the Ultimate System File Structure

The Ultimate system's file structure is unlike that of conventional indexed or sequential files. An Ultimate file is divided into two main parts: the dictionary and the data portion, both of which can be accessed by a BASIC program. The dictionary section defines the attributes (fields) of the file. The data section contains one item (record) for each instance of data (for example, each customer in a Customer file). In both portions, individual records (items) are retrieved directly by record key (item identifier, or "item.id").

Many BASIC programs work with Ultimate data files to access and retrieve, or maintain and update, information.

A typical data file contains items having a common format. In a Customer File, for example, each item may contain a customer name in the first attribute, a corresponding customer address in the second attribute, one or more invoice numbers in the third attribute, and so on. This file structure may be explicitly defined by a set of items (attribute definition items) in the dictionary of the Customer File, but this definition is not required.

The dictionary is a reference tool and does not need to be read by a BASIC program unless needed. If the relative position of the data within items (the structure of the file) is already known, the program can directly access data (such as customer names) by attribute number. If only the attribute name is known, the attribute definition item in the dictionary corresponding to that attribute name can be retrieved and the associated number can be extracted for use in the program.

Attribute definition items in dictionaries are almost always created in a standard format for use with system software such as Recall and Update. For more information on the use of dictionaries, please refer to the *Ultimate Recall and Update User Guide*.

Each item in a file is identified by its item.id, which is the name of the item as well as the value of its key field. Within an item, information is stored in fields (attributes). The attributes can contain subfields called values; multiple values can be stored in an attribute. The values can themselves contain multiple subvalues. A program can access and update any level of data: items, attributes, values, and subvalues.

The format of data stored in an item is referred to as a "dynamic array". Dynamic arrays are described in Chapter 2, Data Representation.

System Delimiters

All data in an item, including the item.id, is in character (string) format; each attribute, value, and subvalue is variable in length and is delimited by special characters known as system delimiters. The format of an item is also called a dynamic array.

The Ultimate system uses characters called attribute marks[1], value marks, and subvalue marks to delimit data. The delimiter mark is inserted in the file at the end of the data it marks. For the user's ease of reading, these special characters are usually shown as special symbols. The symbols displayed by BASIC are different from the symbols displayed by other system functions, such as the editor. Table 5-1 lists the delimiters and the characters used to display them.

Table 5-1. System Delimiters

Delimiter	Editor Symbol	BASIC Symbol	Example
attribute mark	^	~	NAME^ADDRESS
value mark	]	}	100]200
subvalue mark	\	|	1\2

Note: *Throughout this manual, system delimiters are shown using the characters as displayed in the editor.*

For efficiency and minimization of errors, the system delimiters should be defined once in the initialization portion of a program with = (assignment) or EQUATE statements, then referenced by variable name. (EQUATE statements are more efficient then assignment statements.)

[1] may also be called field marks in some versions

The delimiters have also been preassigned to standard names that can be used whenever a system delimiter is needed.

attribute mark

assignment statement:	AM = CHAR(254)
EQUATE statement:	EQUATE AM TO CHAR(254)
predefined variable:	@FM (field mark)

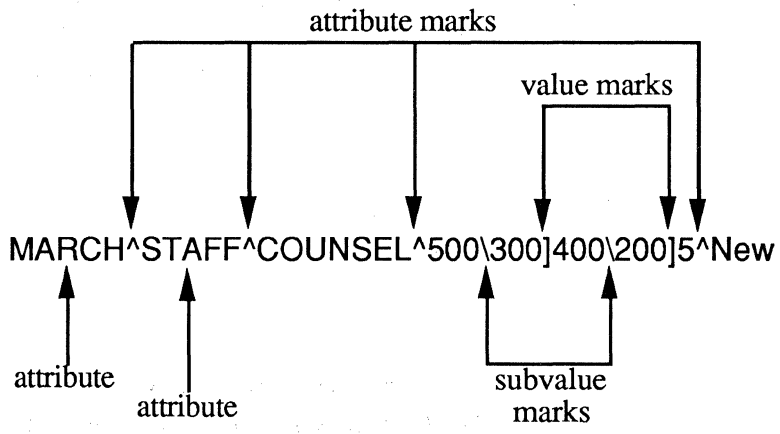
value mark

assignment statement:	VM = CHAR(253)
EQUATE statement:	EQUATE VM TO CHAR(253)
predefined variable:	@VM

subvalue mark

assignment statement:	SVM = CHAR(252)
EQUATE statement:	EQUATE SVM TO CHAR(252)
predefined variable:	@SM

The following example shows a sample item with five attributes:



Segment Marks

In addition to the system delimiters, a fourth delimiter, called a segment mark and abbreviated SM, is used internally by the system to mark the end of every string referenced by a BASIC program. Consequently, BASIC programs should not refer to segment marks or use segment mark characters within strings.

The value of a segment mark is CHAR(255); it is sometimes shown as an underscore (_) as in XYZ_. However, when data that is understood to be delimited by a segment mark is discussed, the SM is not usually shown.

Programming Techniques for Handling I/O

The Ultimate BASIC statements for file access and update (I/O) reflect and accommodate the Ultimate file structure.

The following BASIC statements are used for I/O:

Statement	Description
CLEARFILE	deletes all data items in a file
CLOSE	closes a file; recommended for UltiNet users
DELETE	deletes an item with a specified item.id
MATREAD(U)	reads an item specified by an item.id (key) into a dimensioned array
MATWRITE(U)	writes an item with a specified item.id from a dimensioned array
OPEN	opens a file; required for any I/O functions
READ(U)	reads an item specified by an item.id (key) into a variable
READNEXT	sequentially retrieves item.ids from a select list so that the actual item can be read from file
READT	reads the next record on magnetic tape; note that tape can only be read sequentially, record by record.
READV(U)	reads a specified attribute from an item specified by an item.id (key) into a variable; this statement should only be used when a single attribute is to be accessed from an item.
RELEASE	unlocks item locks set by the program

Statement	Description
SELECT	creates a select list containing the item.ids of all records in a specified file or the attributes of a specified dynamic array
WRITE(U)	writes an item specified by an item.id (key) from a variable
WRITET	writes the next record to tape
WRITEV(U)	writes an attribute from a variable to a specified attribute in an item specified by an item.id (key); this statement should only be used when a single attribute is to be written to an item.

OPEN

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.

If the program executes other programs that use the same files, the file variables can be specified in a named COMMON area and passed to the executed programs. If the program CALLs other programs that use the same files, the files can be specified in either COMMON or named COMMON areas and passed to the CALLed programs.

I/O Considerations for Network Users

Network users should consider the UltiNet system properties when creating programs that may involve remote file access.

For single Ultimate systems, all file access and other I/O tasks involve only "local" files that are directly connected to the system. For network users, however, files may be shared between Ultimate systems via the UltiNet network equipment. This means that files are passed across physical cables and/or modems and telephone lines, with the attendant possibility of errors in the file transfer process.

Please note that the performance of I/O functions over the network is slower than comparable functions from the disk. Network I/O is

measured in hundreds of bytes per second; disk I/O is measured in millions of bytes per second.

When working with remote files, the files should be closed when it is no longer needed in a program; this frees the corresponding remote open-file table entry. Since the number of entries in this table is limited, freeing unused connections could allow greater use of the network. It is also possible to reduce the telephone line charges by closing files, because if all UltiNet files are closed, the link is disconnected. On the other hand, excessive opening and closing of remote files would merely increase network traffic and decrease program efficiency.

The UltiNet equipment can identify a wide variety of error conditions, and is set up to notify the system that requests a file whenever a network error prevents a successful file transfer operation. However, it is the responsibility of the application (that is, the BASIC program) to retrieve the specific error information and act upon it. If no provision is made for processing network errors, the program may abort to the debugger after displaying the error message.

Ultimate BASIC allows programs to identify network errors and specify the actions to be taken. The BASIC statements that involve disk I/O functions all have an optional clause called ON ERROR that allows the program to specify what actions should be taken in case of a network error. The BASIC SYSTEM(0) function has been set up to return the error number generated by the UltiNet equipment as soon as the ON ERROR routine is entered. The ABORT, PUT, and STOP statements allow a program to print a message associated with a specified error number, assuming the error number has been retrieved.

Network users, then, have a number of options when programming applications that may involve remote file access. The following are recommendations:

- Close files when they are no longer needed
- Include an ON ERROR clause in all I/O statements
- Use the SYSTEM(0) function to retrieve the current error number in the ON ERROR statements
- Use statements such as PUT or STOP to display the error number and associated text before either resuming the program or terminating its execution.

Accessing Items

To access an item (record) in a file, a BASIC program must specify the item.id (key). A program cannot sequentially read through items in a file without specifying the item.ids. However, the program can create a list of item.ids, then use READNEXT statements together with READ statements to access items sequentially in the order specified by the list. The item.id list is called a select list, and may be either generated within a program or set up ahead of time before the program is run.

Select lists can be created by a SELECT statement in BASIC or a SELECT or SSELECT command in Recall. The BASIC SELECT statement does not use any selection criteria; all items/elements are included on the list. The Recall commands SELECT and SSELECT, however, are very flexible; they can select and sort, if desired, items according to a conditions that a specified attribute must meet. The list usually contains only item.ids, but it can contain attributes and values as well.

Items can be selected and sorted according to user specifications before running the program, creating a select list of just the item.ids needed for processing. The Recall SELECT commands can also be used in an EXECUTE statement to generate a select list from within the program, instead of prior to running the program.

The Recall SELECT and SSELECT commands allow sophisticated preprocessing of a file using selection criteria in order to retrieve only those items meeting certain conditions. The SSELECT command, moreover, allows access to items in a sorted sequence. For example, the following Recall statement selects items in the CUST file representing customers in Los Angeles, then sorts them into date sequence (ascending order); items that have the same date are sorted into amount sequence (ascending order):

```
SSELECT CUST BY DATE BY AMOUNT WITH CITY="LOS ANGELES"
```

If the SELECT command is outside the program, the same program could be used to produce different results, depending on the selection criteria and, consequently, the items selected. The program itself would not need to be modified.

A simple example of this versatility would be a program to calculate and report information based on dates, company departments, months, quarters, product lines, etc. Only the relevant data for each report would be handed to the program for processing.

More than one select list can be created within a BASIC program. For example:

```
SELECT CUST TO NAMES
SELECT INV TO ORDERS
```

Both statements could be in one program. The CUST select list is assigned to variable NAMES, the INV list to ORDERS. The select lists can be accessed by using these READNEXT statements:

```
READNEXT CUST FROM NAMES ELSE GOTO 100
READNEXT INV FROM ORDERS ELSE GOTO 200
```

The following example opens a file to the default file variable, creates a select list from its item ids, then reads each item in turn. When all items have been read, the program stops.

```
   OPEN 'INV' TO INV ELSE STOP 201,'INV'
   SELECT
10 READNEXT REC ELSE STOP
   READ A FROM INV,REC ELSE STOP 202,REC
   .
   .
   WRITE A ON INV,REC
   GOTO 10
```

Read Locks

Read locks are used by the system to maintain the integrity of data if several processes attempt to update the same item at the same time. A read lock set by one process inhibits another process from updating an item in the locked group, but it does not inhibit other processes from reading an item in that group.

The BASIC statements used to access items set read locks; however, these locks have no impact on user programs.

All statements that read items from a file read-lock the group the item is in. The item is then copied to workspace and the read lock is released. If item locking is specified by the statement, the item lock is set after the read lock is released.

Read locks allow other processes to read items in the group with the read lock, but not to update them. This means, for example, that if a statement with a LOCKED clause tries to access an item in a group with a read lock, the item can still be read and the LOCKED clause is not executed.

All statements that write or delete items set write locks for the group as before. A write or delete statement of an item in a group that has a read lock set does not finish until the read lock is released and until the write or delete is completed, no additional read locks can be set.

Recall

The Recall commands SELECT and SSELECT set a read lock for the group from which the item ids are currently being retrieved. All other Recall commands set read locks only if the output is directed to the spooler or if there is a Tfile conversion. In all cases, if the WITHIN connective is used, read locks are not set.

If there is a Tfile conversion in any Recall statement, a read lock is set for the item being retrieved, the item is copied to workspace, and the read lock is released.

While a read lock is set in RECALL, the BREAK key is inhibited.

Caution! *If Recall sets a read lock and there is a call to a BASIC subroutine, the read lock may cause some problems while the BASIC subroutine is executing. For example, if an INPUT statement in the subroutine expects input from the terminal and no operator responds to the input request, other processes that need to write to that group may appear to hang. Another problem could occur if two Recall processes use the W option to write to files and each process needs to write to the file that the other process has read locked; this would cause a deadly embrace. A RELEASE statement in the subroutine would clear the read locks, but this could mean, possibly, a loss of data integrity and possibly, the occurrence of soft GFEs.*

Accessing Data in Items

When an item has been read into a variable or array, any level of data can be retrieved, changed, and updated in the file. An item may contain up to three levels of data: attributes, values, and subvalues.

The method of accessing specific values in an item depends on how the item has been stored. For example, assume that the file called CUST has one item called 1234 with these attributes:

Attribute Number	Attribute Name	Data
00 (item.id)	NUMBER	1234
01	LAST-NAME	STERN
02	FIRST-NAME	JEFF
03	ADDRESS	125 MORNINGSIDE
04	CITY	ORANGE
05	ZIP	92667
06	ORDERS	10]12

Assume further that the following BASIC statements have been executed:

```
OPEN 'CUST' TO FVAR ELSE STOP 201,'CUST'
SELECT FVAR
READNEXT REC ELSE PRINT 'DONE';STOP
```

At this point, REC = 1234. The item can be read into the program in either dynamic array format or dimensioned array format, as explained below.

Dynamic Array Format

The READ statement is used to read the item in dynamic array format:

```
READ A FROM FVAR,REC ELSE GOTO 9999
```

After the READ, the item is stored in variable A in dynamic array format:

```
STERN^JEFF^125 MORNINGSIDE^ORANGE^92667^10]12
```

Several methods are available to access the attributes in this variable. Angle brackets to specify the desired attribute and an assignment statement could be used:

```
LAST.NAME = A<1>
```

If a value is to be assigned, the angle brackets would enclose the attribute number and value number; for example, the following statement assigns the second value of the sixth attribute to the variable VAL:

```
VAL = A<6,2>
```

The assignment statement could use an intrinsic function to do the same operation:

```
LAST.NAME = EXTRACT(A,1)
```

A number of functions are available for dynamic array processing. These functions include

DELETE
EXTRACT
INSERT
REMOVE
REPLACE
REUSE

A string can be built from two or more attributes:

```
NAME = A<2>:' ':A<1>
```

To write an updated item from a dynamic array, the WRITE statement is used:

```
WRITE A ON FVAR,REC
```

Dimensioned Arrays

Another way of handling a file item is to read it into a dimensioned array. This allows the system to assign each attribute into its own addressable variable for updating.

A DIM or COMMON statement is used to dimension an array; it must precede the associated I/O statements MATREAD and MATWRITE.

Assume again that REC = 1234. The following statements dimension an array A to the number of attributes in the item that is read:

```
DIM A(0)
MATREAD A FROM FVAR,REC ELSE GOTO 9999
```

The INMAT function can be used to determine the number of elements:

```
ARRAY.SIZE = INMAT()
```

In the example, array A has the following six elements:

```
A(1) = STERN
A(2) = JEFF
A(3) = 125 MORNINGSIDE
A(4) = ORANGE
A(5) = 92667
A(6) = 10]12
```

Each attribute can be accessed by its corresponding array element:

```
LAST.NAME = A(1)
```

To write an updated item from a dimensioned array, the MATWRITE statement is used:

```
MATWRITE A ON FVAR,REC
```

Determining the Number of Values

The DCOUNT function may be used to determine the number of values (including null values) in an attribute. For example, the following lines of a program can be used to determine the number of orders in attribute 6 of the previous example:

```
EQUATE VM TO CHAR(253) ; *or VM=CHAR(253)
ORDCOUNT=DCOUNT(A(6),VM)
FOR I=1 TO ORDCOUNT
   PRINT A(6)<1,I>
NEXT
```

If the item is in dynamic array format, the print statement would look similar to the following:

```
PRINT A<6,I>
```

Choosing Between Dynamic and Dimensioned Arrays

There are several things to consider in choosing between dynamic and dimensioned arrays:

An element in a dimensioned array can be accessed more quickly than can an element in a similar size dynamic array.

All the elements in a dimensioned array defined with a literal count towards the total number of variables allowed in a program (currently 3223). If the array is defined with a variable, only one element is counted towards the total number of variables.

Accessing elements in a dimensioned array that is defined with a size of 0 then redimensioned by MATREADing an item into it is slower than accessing elements in an array defined by a literal or a variable, but it is still faster than accessing that element from a dynamic array.

Clearing Variables

The CLEAR statement assigns all variables the value of zero. When not used, all variables initially have an unassigned value. During debugging, this message can be useful in determining potential problems; therefore, it is recommended that CLEAR not be used while debugging is in progress.

Guidelines for Cursor Positioning

Cursor positioning should be controlled by the following PRINT statements using the @ functions. This ensures that the correct control characteristics are sent to the terminal regardless of terminal type (terminal type is specified by the system command TERM).

The @ function is described in Chapter 3.

Code	Description
PRINT @(-1)	Clear screen and position cursor at 'home'
PRINT @(-2)	Position cursor at 'home' (upper left corner).
PRINT @(-3)	Clear from cursor to end of screen
PRINT @(-4)	Clear from cursor to the of line
PRINT @(-5)	Start blink
PRINT @(-6)	Stop blink
PRINT @(-7)	Initiate 'protect' field
PRINT @(-8)	Stop protect field
PRINT @(-9)	Backspace one character
PRINT @(-10)	Move cursor up one line
PRINT @(-11)	Move cursor down one line
PRINT @(-12)	Move cursor right one character
PRINT @(-13)	Enable auxiliary (slave) port
PRINT @(-14)	Disable auxiliary (slave) port
PRINT @(-15)	Enable auxiliary (slave) port in transparent mode
PRINT @(-16)	Initiate slave local print
PRINT @(-17)	Start underline
PRINT @(-18)	Stop underline
PRINT @(-19)	Start inverse video
PRINT @(-20)	Stop inverse video
PRINT @(-21)	Delete line
PRINT @(-22)	Insert line
PRINT @(-23)	Scroll screen display up one line
PRINT @(-24)	Start boldface type
PRINT @(-25)	Stop boldface type
PRINT @(-26)	Delete one character
PRINT @(-27)	Insert one blank character
PRINT @(-28)	Start insert character mode
PRINT @(-29)	Stop insert character mode

Programming for Maximum System Performance

The size of programs can be reduced, with a corresponding increase in overall system performance, by reducing the amount of literal storage. The allocation of variables can also affect system performance. Operations should be predefined rather than repetitively performed.

Minimizing Program Size

An example of inefficient literal storage is the following:

```
200 PRINT 'RESULT IS ':A+B
210 PRINT 'RESULT IS ':A-B
220 PRINT 'RESULT IS ':A*B
230 PRINT 'RESULT IS ':A/B
```

A more efficient way to write these statements is as follows:

```
190 MSG = 'RESULT IS '
200 PRINT MSG:A+B
210 PRINT MSG:A-B
220 PRINT MSG:A*B
230 PRINT MSG:A/B
```

Variable Allocation

Variables are allocated space in the descriptor table as they are defined in a program. The most frequently used variables and COMMON variables should be defined at the beginning of a program. To prevent needless wasted storage space, it is recommended that standard variable names be agreed upon within your user group.

Repetitive Operations

The following statement is an example of an inefficient, repetitive operation:

```
X=SPACE(9-LEN(OCONV(COST,'MCA'))):OCONV(COST,'MCA')
```

It could have been written more efficiently as follows:

```
E=OCONV(COST,'MCA')
X=SPACE(9-LEN(E)):E
```

The following statements are another example of a repetitive operation:

```
FOR I=1 TO X*Y+Z(20)
   .
   .
NEXT I
```

They could have been written as follows:

```
TEMP=X*Y+Z(20)
FOR I=1 TO TEMP
   .
   .
NEXT I
```

Programming Examples

PRIME.NUMBER

This program finds prime numbers.

```
*
*  TEST A NUMBER TO SEE IF IT IS PRIME.
*  IF IT IS NOT, FIND THE SMALLEST PRIME NUMBER
*  GREATER THAN THE ORIGINAL NUMBER.
*
   PRINT
   PRINT 'Enter number to test ':
   INPUT NUM
   PRINT
10 NULL
   IF REM(NUM,2) = 0 THEN
      PRINT NUM: ' is even!'
      NUM=NUM+1
   END
20 NULL
   TEST.NUM = SQRT(NUM)
   FOR N=3 TO TEST.NUM STEP 2
      IF REM(NUM,N) = 0 THEN
         PRINT NUM: ' is divisible by ':N
         NUM = NUM+2
         GOTO 20
      END
   NEXT N
   PRINT NUM: ' is prime!'
   STOP
   END

Sample Run:

Enter number to test ?44

44 is even!
45 is divisible by 3
47 is prime!
```

P0000 (File Update)

This program uses terminal input to update a master file.

```
*
*  UPDATE PROM MASTER FILE
*
   DIM P(5)
   EQU BELL TO CHAR(7), FPMSK TO '4N'
   CLRL=@(-4)
   CLR=@(-1)
   L15='L(#15)'
   TL=CLR:@(12,1):'** CCARM Corp Master Update **'
   TL = TL:@(70,1):'P0000'
   TL = TL:@(4,4):'Enter Prom Part Number: '
   PROMPT ""
   PS=@(0,12):CLRL:BELL
   PRMPT=@(0,12):CLRL:"Enter Line # to change/'D' to
    delete"
   SCR = 'COMPANY:'L15:'PROM WIDTH:'L15
   SCR := 'PROM DEPTH:'L15:'F/P CODE:'L15
   SCR := 'FILL CHAR:'L15
   *
   OPEN 'PROMMASTER' TO PM ELSE
    STOP 201,'PROMMASTER'
    END
10 FLG1=0
   PRINT TL:
   INPUT PID
   IF PID = "END" OR PID="" THEN STOP
   MATREAD P FROM PM,PID ELSE MAT P=""; FLG1=1
*  FLG1 = ITEM NOT ON FILE
   FOR W=1 TO 5
    PRINT @(0,W+5):W'R(##. )':SCR[(W-1)*15+1,15]:P(W)
   NEXT W
   IF FLG1 THEN GO 30
20 PRINT PRMPT:; INPUT ANS
   IF ANS="" THEN MATWRITE P ON PM,PID; GO 10
   IF ANS='D' THEN DELETE PM,PID; GO 10
   IF ANS>0 AND ANS<6 THEN W=ANS; GO 40
   GO 20
```

```
*
30 * ADD NEW ITEM *
   FOR W=1 TO 4
40   PRINT @(19,W+5):CLRL:; INPUT P(W)
     BEGIN CASE
       CASE P(W)='B'
         W=W-1; IF W=0 THEN GO 10 ELSE GO 40
       CASE W=2
         IF P(2)#4 & P(2)#8 THEN
           PRINT PS:'Must be a 4 or 8 in this field'
           GO 40
         END
       CASE W=3
         IF NOT(NUM(P(3))) THEN
50         PRINT PS:'Invalid response, must be decimal/K
             units'
           GO 40
         END
         IF REM(P(3),32)#0 THEN GO 50
       CASE W=4
         IF NOT(P(4) MATCH FPMSK) THEN
           PRINT PS:'Must be 4 decimal digits'; GO 40
         END
       CASE W=5
         IF P(5)#0 & P(5)#"F" THEN
           PRINT PS:'Must be "0" or "F" '; GO 40
         END
     END CASE
     IF NOT(FLG1) THEN GO 20
   NEXT W
   FLG1=0 ; * ITEM NOW EXISTS
   GO 20
```

Sample Terminal Output

```
***CCARM Corp  Master Update***                    P0000

Enter Prom Part Number: 222

     1.    Company:     SMITH
     2.    Prom Width:  4
     3.    Prom Depth:  256
     4.    F/P Code:    1112
     5.    Fill Char:

Enter Line # to change/'D' to delete item/-NL- to
update _
```

ITEMS.BY.CODE (Use of Job Control)

This program illustrates the use of the EXECUTE statement to create a job control application. The operator enters a dictionary code for searching the current account's master dictionary. The application displays a sorted and numbered list of master dictionary items that have the specified dictionary code. The application can then be re-run or ended.

```
*** PROGRAM USING THE EXECUTE STATEMENT***
*
   OPEN "DICT","MD" TO MD ELSE STOP 201,'MD'
   CLEAR = @(-1)
   CES = @(-3)
   CEL = @(-4)
   PRINT CLEAR:@(0,22):
10 PRINT "Enter the dictionary code for the search or
'END '":CEL:
   INPUT CODE
   IF CODE = "" OR CODE = 'END' THEN STOP
   XXX = ""
   ID = ""
*  SELECT THE FILE
*    PUT SELECT LIST IN VARIABLE ID
*    ERROR MSG IN VARIABLE XXX
*
   EXECUTE 'SSELECT MD WITH D/CODE = "':CODE:'"',
RTNLIST ID, RETURNING XXX
   IF XXX<1,1> = "401" THEN
     PRINTERR "No dictionary items for code = ":CODE
     PRINT @(0,22):
     GOTO 10
   END
   PRINT CLEAR:
   PRINT "Master Dictionary items with a dictionary
code of ":
   PRINT CODE
   I = 1
   X = 0 ; Y = 2
   LONGEST = 0
*  PRINT THE ITEM ID'S WITH SEQUENCE NUMBERS
```

```
    LOOP WHILE ID<I> # "" DO
      IF Y = 21 THEN
        X = X + 5 + LONGEST
        Y = 2
        LONGEST = 0
      END
      IF X + LEN(ID<I>) + 3 > 79 THEN
        PRINT @(0,22):"I need to clear the screen ":CES:
        PRINT "to display the remaining items, "
        PRINT "Press <RETURN> to continue or (C)ancel ":
        INPUT ANS:
        IF ANS[1,1] = "C" THEN PRINT @(0,22): ; GOTO 10
        PRINT @(0,2):CES:
        X = 0
        Y = 2
        LONGEST = 0
      END
      PRINT @(X,Y):I 'R(###)':" ":ID<I>
      IF LEN(ID<I>) > LONGEST THEN
        LONGEST = LEN(ID<I>)
      END
      I = I + 1
      Y = Y + 1
    REPEAT
    PRINT @(0,22):
    GOTO 10
STOP
END
```

Sample Terminal Output:

```
Enter the dictionary code for the search or 'END'?Q

Master Dictionary items with a dictionary code of Q

  1   ACC              20   QFILE
  2   ALPHA            21   SYSLIB
  3   AREA             22   WORDS
  4   BARB             23   ZIP
  5   BBP
  6   BLOCK
  7   CHANNEL
  8   COMMS
  9   ERRMSG
 10   INV.A
 11   INV.B
 12   INVENTORY
 13   INV-PROSPECT
 14   LEADS
 15   MAIL.FILE
 16   NEXT
 17   PROCLIB
 18   PROSP
 19   PUB

Enter the dictionary code for the search or 'END'?_
```

SUMMARY.REPORT (Menu/Report Generator)

The BASIC program listing below contains the coding for a "Summary Prospect Report". An abbreviated listing of a subroutine called GET.CRITERIA, which is called by SUMMARY.REPORT, is included following the main program.

This program illustrates sample coding from a menu-driven set of report generation programs. The actual process,which is not included, is set up so that the operator can select the desired report option from a "D&B Prospect Selector" menu produced by a PROC. The PROC calls the appropriate BASIC program to print the selected report. The program prints the report and returns to the PROC, which redisplays the menu.

```
*** PROGRAM TO PROMPT OPERATOR FOR STATE CODES,
*** COUNTY CODES,SIC CODES, AND SALES VOLUME.
*** PROGRAM WILL ALLOW UP TO TEN DIFFERENT
*** REPORT CRITERIA TO BE SET UP BEFORE THE
*** REPORTS ARE GENERATED
***
   COMMON STATES,COUNTIES,SALES,SICCODES,SIC.SELECT,
SORT.BY, TITLE, NAME, FLAG
   PROMPT ""
   DIM RPTS(11)
   MAT RPTS = ' '
   RPT = 1
***
10 PRINT @(-1):@(10,0):"Selective Prospect Summary
Report ":
   PRINT "                        Report #":RPT:
*  Call Subroutine to prompt for selection criteria
   CALL GET.CRITERIA
   IF FLAG = 'X' THEN GO 10
   IF FLAG = '-' THEN GO 90
***
80 * BUILD RPT RECALL STATEMENT
   RPTS(RPT) = "SORT USC.PROSPECT "
   RPTS(RPT) = RPTS(RPT): " WITH STATE "
   IF STATES = "ALL" THEN GO 81
   X = 1
```

```
      LOOP
        STATE = STATES<X>
      UNTIL STATE = ' ' DO
        RPTS(RPT) =RPTS(RPT):' = ':STATE
        X = X + 1
      REPEAT
   81 RPTS(RPT) =RPTS(RPT):" AND WITH COUNTY-CODE "
      IF COUNTIES = "ALL" THEN GO 82
      X = 1
      LOOP
        COUNTY = COUNTIES<X>
      UNTIL COUNTY = "" DO
        RPTS(RPT)=RPTS(RPT):' = ':COUNTY
        X = X + 1
      REPEAT
   82 RPTS(RPT) =RPTS(RPT):' AND WITH SALES >= "'
      RPTS(RPT) =RPTS(RPT):SALES: '"'
   83 RPTS(RPT) =RPTS(RPT):" AND WITH "
      IF SIC.SELECT = "P" THEN
        RPTS(RPT) =RPTS(RPT):" PRIMARY-SIC "
      END ELSE
        RPTS(RPT)=RPTS(RPT):" SIC-CODES "
      END
      IF SIC.CODES = 'ALL' THEN GO 84
      X = 1
      LOOP
        FROMSIC = SIC.CODES<X,1>
        TOSIC   = SIC.CODES<X,2>
      UNTIL FROMSIC = "" DO
        IF X > 1 THEN RPTS(RPT) = RPTS(RPT):" OR "
        RPTS(RPT) =RPTS(RPT):' >="':FROMSIC:'"'
        RPTS(RPT) =RPTS(RPT):' AND <= "':TOSIC:'"'
        X = X + 1
      REPEAT
   84 ***
      BEGIN CASE
        CASE SORT.BY = 1
          RPTS(RPT) =RPTS(RPT):" BY ZIP BY COMPANY "
        CASE SORT.BY = 2
          RPTS(RPT) =RPTS(RPT):" BY COMPANY "
        CASE SORT.BY = 3
```

```
      RPTS(RPT) =RPTS(RPT):" BY PRIMARY-SIC BY"
      RPTS(RPT) =RPTS(RPT):" COMPANY "
   END CASE
   RPTS(RPT) =RPTS(RPT):" COMPANY OFFICER "
   RPTS(RPT) =RPTS(RPT):"TELEPHONE DMI-LINE SALES"
   RPTS(RPT) =RPTS(RPT):' HEADING " ':" 'T'"
   RPTS(RPT) =RPTS(RPT)                              D&B "
   RPTS(RPT) =RPTS(RPT):"Prospect Report for - ":NAME
   RPTS(RPT) =RPTS(RPT):"                Page 'PC' 'L'"
   RPTS(RPT) =RPTS(RPT):"'L' ":TITLE:" 'C' 'LL' ":'"'
   RPTS(RPT) =RPTS(RPT):"DBL-SPC ID-SUPP LPTR "
***
90 *** BUILD ANOTHER RPT?
   RPT = RPT + 1
   IF RPT > 10 THEN GO 1000
91 PRINT @(5,23):"Do you wish to generate another
report":
   PRINT "(Y/N)? #  ":@(51,23):; INPUT RSP,1:_
   PRINT @(51,23):SPACE(2):
   PRINT @(51,23):RSP:
   IF RSP = 'X' THEN
     RPT = RPT - 1
     IF RPT < 1 THEN RPT = 1
     GO 10
   END
   IF RSP = 'Y' THEN GO 10
   IF RSP # 'N' THEN GO 91
***
1000 *** EXECUTE RPTS
   PRINT @(-1):@(10,0):"Now processing reports....."
   RPT = 1
   LOOP
     STATEMENT = RPTS(RPT)
   UNTIL STATEMENT = ' ' DO
     PRINT;PRINT
     PRINT STATEMENT
     EXECUTE STATEMENT
     RPT = RPT + 1
   REPEAT
   STOP
END
```

The following is the subroutine called by SUMMARY.REPORT:

```
SUBROUTINE GET.CRITERIA
   COMMON STATES, COUNTIES, SALES, SIC.CODES,
      SIC.SELECT, SORT.BY, TITLE, NAME, FLAG
   FLAG = ' '
***
20 *** GET STATES
   X = 1
   STATES = ' '
   PRINT @(5,2):"Enter State code - ":
22 PRINT @(21+(X*3),2):"## ":@(21+(X*3),2):
   INPUT STATE,3:_
   PRINT @(21+(X*3),2):SPACE(3):
   IF STATE = 'X' OR STATE = 'END' THEN
     IF X = 1 THEN STOP
     FLAG = '-'
     GO 99
   END
   IF STATE = ' ' THEN GO 30
   IF STATE = '-' AND X > 1 THEN
     PRINT @(21+(X*3),2):SPACE(3):
     STATES = DELETE(STATES,X,0,0)
     STATES = DELETE(STATES,X-1,0,0)
     X = X - 1
     GO 22
   END
   PRINT @(21+(X*3),2):STATE 'L(#3)':
   IF STATE = 'ALL' THEN STATES = 'ALL'; GO 30
   IF NOT(STATE MATCHES '2A') THEN GO 22
   STATES = REPLACE(STATES,X,0,0,STATE)
   X = X + 1
   IF X > 12 THEN GO 30
   GO 22
***
30 *** GET COUNTY CODE
   .
   .
40 *** GET MINIMUM SALES VOLUME
   .
   .
```

```
50 *** GET SIC CODE RANGES
   .
   .
60 *** SELECT ON PRIMARY OR ALL SIC CODES
   .
   .
70 *** GET FREE FORM HEADING
   .
   .
73 *** GET OPERATORS NAME
   .
   .
75 *** GET SORT CRITERIA
   PRINT @(5,21):
   PRINT "Sort by 1)Zip 2)Company name 3)SIC code # ":
   PRINT @(51,21):
   INPUT SORT.BY,1:_
   PRINT @(51,21):SPACE(1):
   IF SORT.BY = 'X' OR SORT.BY = 'END' THEN
     FLAG = 'X'
     GO 99
   END
   IF SORT.BY = ' ' THEN GO 75
   IF SORT.BY = '-' THEN GO 73
   IF SORT.BY < 1 OR SORT.BY > 3 THEN GO 75
   PRINT @(51,21):SORT.BY 'R(#)':
***
99 *** RETURN
   RETURN
```

Sample menu produced by PROC

```
        D&B Prospect Selector
       **********************

1) Detailed Prospect Report

2) Summary Prospect Report

3) Prospect Label Print

4) Detailed, Summary, and Label Print

5) User instructions

88) Exit to 'TCL'

99) Logoff


Enter option - _
```

QOH (Use of LOCATE with Dynamic Arrays)

This program prints a report showing total cost * quantity on hand by product group.

```
**Print cost * quantity on hand by product group.
*
DIM STOCKITEM(20)                    ;* ID = Part No
  EQU COST    TO STOCKITEM(3)        ;* Cost
  EQU QOH     TO STOCKITEM(8)        ;* Quantity on Hand
  EQU PRODGRP TO STOCKITEM(9)        ;* Product Group
*
  EQU AM TO CHAR(254)
  EQU VM TO CHAR(253)
*
  RESULTS=''
  OPEN '','STOCK' TO STOCK ELSE STOP 201,'STOCK'
  SELECT STOCK
1 *
  READNEXT ID ELSE
    HEADING "Product Total Value     Average'L' Group"
:SPACE(22):"Value'L'"
  VMC=DCOUNT(RESULTS<1>,VM)
  FOR I=1 TO VMC
    PRINT RESULTS<1,I> "L(#7 )":
    PRINT OCONV(RESULTS<2,I>,'MR2,$') "R(#12)":
    IF RESULTS<3,I>=0 THEN
      AVE=0
    END ELSE
      AVE=RESULTS<2,I>/RESULTS<3,I>
      END
    PRINT OCONV(AVE,'MR2,$') "R(#12)"
  NEXT I
  STOP
  END
  MATREAD STOCKITEM FROM STOCK,ID ELSE GOTO 1
  LOCATE PRODGRP IN RESULTS<1>,1 BY 'AL' SETTING POS
ELSE
    INS PRODGRP BEFORE RESULTS<1,POS> ;* product group
    INS ''      BEFORE RESULTS<2,POS> ;* total value
    INS 0       BEFORE RESULTS<3,POS> ;* total quantity
```

```
    END
  RESULTS<2,POS>=RESULTS<2,POS> + (COST*QOH)
  RESULTS<3,POS>=RESULTS<3,POS> + QOH
  GOTO 1
  END
```

Sample report:

```
Product  Total Value     Average
Group                      Value

A              $36.00       $3.00
B              $52.00       $4.00
C              $70.00       $5.00
D              $90.00       $6.00
```

A BASIC Compiler Messages

This appendix presents a list of messages that may occur as a result of compiling a BASIC program. The messages are stored in the ERRMSG file. In the following descriptions, the error number and message are printed in **boldfaced** type. The cause and explanation are in normal type.

[B100] Line A, 'B' Compilation aborted; no object code produced

Compilation errors present.

[B101] Warning - end of compilation before end of item, at line number below:

An END statement, unassociated with a multi-line THEN or ELSE clause, has been encountered before the physical end of the item

[B102] Line A Object code exceeds 57,534 bytes.

[B103] Line A Label 'B' is missing

Label indicated by GOTO or GOSUB was not found or MATREAD statement uses simple variable instead of dimensioned array.

[B104] Line A Label 'B' is doubly defined

[B105] Line A 'B' has not been dimensioned

[B106] Line A 'B' has been dimensioned and used without subscripts

[B107] Line A LOOP statements nested too deep

[B108] Line A NEXT statement missing

[B109] Line A Variable missing in NEXT statement

B110 **Symbol table is A% full**

B111 **Last variable is at A**

--------- V A R I A B L E S ---------

B112 **----------- L A B E L S -----------**

```
B113          ---------- E Q U A T E S ----------
```

[B114] Line A Maximum number of variables exceeded

More than 3223 variables (including array elements) used.

[B115] Line A Label 'B' is used before the EQUATE stmt.

The symbol is referenced before it has been defined.

[B116] Line A Label 'B' is used before the COMMON stmt.

A COMMON variable is referenced before it is declared as COMMON.

[B117] Line A Label 'B' is missing a subscript list.

[B118] Line A Label 'B' is the object of an EQUATE statement and is missing.

Variable after TO clause in EQUATE statement has not been defined.

[B119] Line A Warning - precision value out of range - ignored

A precision not in the range of 0-9 was specified.

[B120] Line A Warning - multiple precision statements - ignored

Only the first precision statement is applied.

[B121] Line A Label 'B' is a constant and cannot be written into.

An EQUATEd symbol has been specified in an assignment statement.

[B122] Line A The label 'B' is used incorrectly.

Expression after TO in EQUATE is illegal.

[B123] Line A Label B has been dimensioned with non-zero subscript in COMMON and cannot be re-dimensioned

[B124] Line A Label 'B' has literal subscripts out of range.

Array subscript less than 1 or greater than value in DIM statement.

[B125] No source statements found; no object code produced

[B126] Line A ELSE clause missing

[B127] Line A NEXT missing

[B128] Line A Item 'B' not found

Object of $INCLUDE or $CHAIN directive is missing.

[B129] Illegal: program name same as dictionary item name

Compiler cannot write object code in dictionary because non-object code dictionary item already exists with the same name as the source program.

[B130] Line A Symbol Table overflow - compilation aborted

B131 **Symbol table is A% full**

B132 **Last variable is at A**
--------- V A R I A B L E S ---------

B133 **------------ L A B E L S ------------**

B134 **----------- E Q U A T E S -----------**

Note: *In the following messages, the B represents a caret (^) that the compiler uses to point to the word that is causing the problem. For example, error message B135 is used as follows:*

```
004 OPEN 'LEDGER' TO GLE ELSE STOP 201,'LEDGER'
***                      ^ Reserved word used
```

B135 ***B Reserved word used**

B136 ***B Illegal assignment statement**

B137 ***B END CASE statement missing**

B138 ***B Variable name expected**

B139 ***B Keyword 'C' expected**

B140 ***B Illegal expression

B141 ***B END statement missing

B142 ***B Illegal syntax

B143 ***B Illegal library function name

B144 ***B End of statement expected

B145 ***B THEN or ELSE clause missing

B146 ***B EXIT used outside LOOP-REPEAT construct

B147 ***B REPEAT missing in LOOP construct

B148 ***B Ambiguous ELSE clause

[B150] Warning - end of compilation before end of item

[B151] Line A Scope of RETURN TO/FOR statement exceeds 28,767 bytes of object.

B152 ***B Illegal System variable (@var)

[B153] Line A Object size plus symbol table size exceeds 128 frames ! Try using (S) option and recompile.

B155 *** Reserved word used B

B156 *** Illegal assignment statement B

B157 *** END CASE statement missing B

B158 *** Variable name expected B

B159 *** Keyword 'C' expected B

B160 *** Illegal expression B

B161 *** END statement missing B

B162 *** Illegal syntax B

B163 *** Illegal library function name B

B164 *** End of statement expected B

B165 *** THEN or ELSE clause missing B

B166*** EXIT used outside LOOP-REPEAT constr B

B167 *** REPEAT missing in LOOP construct B

B168 *** Ambiguous ELSE clause B

B172 *** Illegal System variable (@var) B

[B173] Line A IN. Variable exceeds 1600 bytes.

[B180] Line A The item 'B' was included before. Multiple includes of the same item are illegal.

B199 Source file must have separate DICT and DATA sections

[B241] Line A, 'B' successfully compiled; C frames used.

Notes

B BASIC Run-Time Messages

This appendix presents a list of the error messages that may occur as a result of executing a BASIC program. The messages are stored in the ERRMSG file.

Warning messages indicate that illegal conditions have been smoothed over (by making an appropriate assumption), and do not result in program termination. Fatal error messages result in program termination.

In the following descriptions, the error number and message are printed in **boldfaced** type. The cause and explanation are in normal type.

[B1] Run-time abort at line A
Caused by BASIC statement ABORT.

[B2] Line A Illegal file specification

[B5] Line A Incorrect number of subroutine parameters
Number of parameters in CALL statement is greater than number of parameters in SUBROUTINE statement

[B6] Line A PROCWRITE string length exceeds 350 characters

[B10] Line A Variable has not been assigned a value; zero used
An unassigned variable was referenced; a value of 0 is assumed.

[B11] Line A Tape record truncated to tape record length
An attempt was made to write a tape record greater than the tape record length. (The record is truncated to tape record length.)

[B12] Line A File has not been opened
File indicated in I/O statement has not been opened via an OPEN statement or file variable has been modified since the file was opened

```
[B13] Line A  Null conversion code is illegal;
no conversion done
```

```
[B14] Line A  Bad stack descriptor: B
```
Number of parameters in CALL statement less than number of parameters in SUBROUTINE statement

```
[B15] Line A  Illegal opcode:  B
```
The program object code has been corrupted; recompile program

```
[B16] Line A  Non-numeric data when numeric
required; zero used
```

```
B17] Line A  Array subscript out of range: B
```

```
[B18] Line A  Attribute number less than -1 is
illegal.
```
Attribute less than 1 is specified in READV or an attribute number less than -1 is specified in WRITEV statement.

```
[B19] Line A  Illegal pattern
```
Illegal pattern used with MATCHES operator or in MATCHFIELD function.

```
[B20] Line A  COL1 or COL2 used prior to
executing a FIELD stmt; zero used
```

```
[B22] Line A  Illegal value for STORAGE
statement
```
STORAGE parameter less than 10 or not a multiple of 10.

```
[B23]  Program 'B' must be recompiled.
```
Object code not compatible with current operating system.

```
[B24] Line A  Divide by zero illegal; zero used
```

```
[B25]  Program 'B' has not been cataloged.
```
A subroutine specified in a CALL statement was not found in the program file and was not cataloged or if cataloged, not found in the file specified in the MD.

```
[B26] Line A  'UNLOCK C' attempted before LOCK
```

```
[B27] Line A  RETURN executed with no GOSUB
```

[B28] Line A Not enough work space on Reg B
Not enough user work space to hold all variables.

[B29] Line A USERMSG file has not been opened by the SET-LANGUAGE verb

[B30] Line A Array size mismatch
Array sizes do not match in MAT Copy statement, or in CALL and SUBROUTINE statements.

[B31] Line A Workspace underflow, register B
The program has attempted to CALL too many nested subroutines; the number of subroutines that can be nested depends on the number of variables used as well as the number of CALLs

[B32] Line A Page heading exceeds maximum of 1400 characters

[B33] Line A Precision declared in subprogram 'C' is different from that declared in the mainline program.

[B35] Line A BASIC operation not allowed when called via RECALL dictionary
A subroutine called by the B processing code contains one of the disallowed BASIC statements; see the *Ultimate Recall and Update User Guide* for a list of operations that are not allowed.

[B36] Line A Arrays in calling program and subroutine must both be either fixed dimensions, or both variable dimensions

[B37] Line A Variable dimensioned array element referenced before array was initialized by a DIM statement

[B38] Line A Subroutine argument was incorrectly passed twice

[B39] Line A Incorrect PRECISION in /B/ COMMON block.
PRECISION has been changed after named COMMON area has been defined

[B40] Line A Incorrect number of variables in /B/ COMMON block.

Number of variables in current named COMMON statement is different from number in named COMMON statement that first defined the area

[B41] Line A Lock number is greater than 47.

[B42] Line A COMMON block /B/ has not been initialized or has been already released.

[B43] Line A attempt to create more than 50 named COMMON blocks.

[B44] Line A named COMMON block table Full.

[B45] Program 'B' is not a subroutine!

Program named in CALL statement is not a subroutine.

[B107] Line A LOOP statements nested too deep

LOOP statements can be nested to 50 levels.

[B209] Line A File is update protected.

An attempt was made to update a file that has an update lock.

[B210] Line A File is access protected.

An attempt was made to read a file that has a retrieval lock

C BASIC Debugger Messages

The following is a list of messages that can be displayed by the BASIC debugger. In the following descriptions, the message that is displayed is printed in **boldfaced** type. The cause and explanation are in normal type.

[B501] Invalid BASIC DEBUGGER command.
This message is displayed when the user enters a command that cannot be understood by the BASIC debugger.

[B503] Not a valid file name.
This message is displayed when the BASIC Debugger is unable to open a filename containing source code. This may be because the file does not exist in this account or retrieval/update codes prevent access.

[B504] Not a valid program name.
This message is displayed when the BASIC debugger cannot locate a source item. On an 'L' command or display of source code after a breakpoint, the BASIC debugger attempts to locate the source code in the file that contains the mainline program. For example, if the debugger is entered in a subroutine that is not in the same file as the mainline program, this message is displayed.

After displaying this message, the debugger prompts for the File/Program Name:

```
File/prg name?
```

Enter the filename and the program name.

[B505] Trace and Breakpoint tables are empty.
This message is displayed in response to a 'D' command when both the Breakpoint and Trace tables are empty.

[B506] You cannot continue execution of a program after a fatal abort !
This message is displayed when a user types 'G<CR>' or '<LF>' in BASIC debugger after BASIC runtime detected a fatal error. Execution

cannot continue because the state of the BASIC runtime stack is in an indeterminable state at entry to BASIC debugger. The user is not restrained from entering 'Gline#' in this situation, but it cannot be assumed that execution will continue in this situation either.

[B507] Invalid line number.
This message is displayed when a user attempts to 'G' to a line# past the end of executable code.

[B508] Zero is not an acceptable value.
This message is displayed when the user attempts to supply a zero as a numeric value when zero is unacceptable; for example, display dimensioned element zero. /ARRAY(0).

[B510] Symbol not in SYMBOL TABLE.
The symbol entered by the user cannot be located in the symbol table.

[B511] Non-numeric data.
The user has typed non-numeric data as a command parameter; for example, GTOP.

[B514] Variables must start with an alphabetic character.
The user has entered a variable name that does not start with an alphabetic character; for example, /123.

[B515] Subscript out of range.
The user entered subscripts for a dimensioned array that are greater than the maximum defined in the DIM statement.

[B516] Dimensions illegal for a simple variable.
The user entered subscripts for a non-dimensioned variable.

[B517] NAMED COMMON/VARIABLE dimensioned array has not been initialized.
The user can only display, trace and set breakpoints for named COMMON and variable dimensioned arrays after they have been initialized by the BASIC Runtime package.

[B518] The SYMBOL TABLE is not in the correct format or does not exist. This program must be recompiled if BASIC DEBUGGER is to be used on it.
Either the program was not compiled under the latest version of the BASIC Compiler, or the symbol table was not generated because the

program was compiled with the S option or it contains a $NODEBUG statement.

B520 (not displayed)

```
E Count          A
N Count          B
Window  Start,Len          C,D
Hex  Display          E
Source  Display       F
CALL/RETURN  Trap          G
Reason  Code          H
```

This is the display is generated by the O{ptions} command.

[B521] Trace table Full.

This message is displayed when the user attempts to T{race} a variable and the trace table is full.

[B522] Trace entry not found.

This message is displayed when the user attempts to U{ntrace} a variable and the entry cannot be found. This may be because the entry# entered does not contain an entry or if a variable name is entered, the is no entry for that variable in the trace table.

B524 **Trace table cleared.**

This message is generated if the user types U{ntrace}] with no entry or variable name and the whole table is cleared.

B529 **Trace Variable**

This is the 'heading' for the D{isplay} command and Trace table entries exist.

[B531] Breakpoint table full.

This message is displayed when a user attempts to create a Breakpoint condition and there are no free entries in the Breakpoint Table to hold the condition.

[B532] Breakpoint entry not found.

This message is displayed when the user attempts to K{ill} a Breakpoint entry and the entry # supplied does not contain a Breakpoint condition or if a variable name is supplied, no Breakpoint condition can be found for that variable.

[B533] No logical operator found in B{reakpoint} command.

The BASIC debugger could not locate a '=, #, < or >' symbol in the Breakpoint condition entered by the user.

B534 **Breakpoint table cleared.**

This message is generated when a user enters a K{ill} command with no parameters and the BASIC debugger clears all Breakpoint conditions.

[B535] Non numeric line number.

The user entered B$ and the constant after the operator was not numeric; for example, B$=ABC.

[B536] Only one '&' allowed in a B{reakpoint} command.

The Breakpoint condition entered by the user contains more than one & operator.

[B537] Breakpoints can only be set on single array elements.

The use has attempted to set a Breakpoint condition using a dimensioned array with no dimensions.

B539 **Break Condition**

This message is a 'heading' for the D{isplay} command when Breakpoint Tables entries exists.

B540 **Return Stack**

This message is a 'heading' for the R{eturn Stack Display} command when return entries are found in the BASIC Runtime stack.

[B541] There are no GOSUBs in the return stack.

This message is displayed when the R{eturn Stack Display} command is entered an there are no return entries found in the stack.

B555 Help Message (not displayed)

The following message is generated in response to the H{elp} command :-

```
?, $, *, V      - show file, prog name, line# and verify
                  object
/var{(r,c)}{<a,v,s>} - display (and alter) a variable.
                  var = * displays all
[{m,n}          - set/reset display window
Bvoc{&voc}      - B{reakpoint set}
C               - C{ALL/RETURN breakpoint on/off}
D               - D{isplay trace and breakpoint tables}
DE              - DE{bug enter}
E{n}            - E{xecution step set/reset}
G{n}            - G{oto line# or resume processing}
H               - H{elp}
HX              - H{e}X{display on/off}
K{n}{var}       - K{ill breakpoint entry or var}
L{n,{m}}{*}     - L{ist source}
LP              - L{ine} P{rinter on/off}
N{n}            - N{umber of breakpoints to step through
O               - O{ptions Display}
P               - P{rinted output on/off}
PC              - P{rinter} C{lose}
R               - R{eturn stack display}
S               - S{ource code display on breakpoint
                  on/off}
T{var}          - T{race var} or T{race on/off}
U{n}{var}       - U{ntrace entry or var}
Z               - {set up source code pointers}
END,BYE,STOP    - Terminate program (STOP resumes PROC
                  processing
OFF             - Terminate program and log off.
```

Notes

D List of ASCII Codes

This appendix presents a list of ASCII codes for decimal number values from 0 through 255. The hexadecimal equivalent value and ASCII character generated are also given.

Decimal values 0-31 are assigned as non-printable functions; these codes may be specified by control key sequences as input to a BASIC program. In the listing, the control key is indicated by a caret (^) in the first position in the Key column.

Decimal values greater than 127 (x'7F') are not defined in the ASCII character set. The functions or characters assigned to these values are dependent on the terminal being used. However, special file structure functions and control key sequences have been assigned to decimal values 251 through 255 (x'FB' through x'FF').

Decimal	Key	Hexadecimal	Name
0	^@	00	NUL
1	^A	01	SOH
2	^B	02	STX
3	^C	03	ETX
4	^D	04	EOT
5	^E	05	ENQ
6	^F	06	ACK
7	^G	07	BEL
8	^H	08	BS
9	^I	09	HT
10	^J	0A	LF
11	^K	0B	VT
12	^L	0C	FF
13	^M	0D	CR
14	^N	0E	SO
15	^O	0F	SI
16	^P	10	DLE
17	^Q	11	DC1
18	^R	12	DC2
19	^S	13	DC3
20	^T	14	DC4
21	^U	15	NAK
22	^V	16	SYN
23	^W	17	ETB
24	^X	18	CAN
25	^Y	19	EM
26	^Z	1A	SUB
27	^[	1B	ESC
28	^\	1C	FS
29	^]	1D	GS
30	^^	1E	RS
31	^_	1F	US

Decimal	Key	Hex	Decimal	Key	Hex
32		20	80	P	50
33	!	21	81	Q	51
34	"	22	82	R	52
35	#	23	83	S	53
36	$	24	84	T	54
37	%	25	85	U	55
38	&	26	86	V	56
39	'	27	87	W	57
40	(	28	88	X	58
41	)	29	89	Y	59
42	*	2A	90	Z	5A
43	+	2B	91	[	5B
44	,	2C	92	\	5C
45	-	2D	93	]	5D
46	.	2E	94	^	5E
47	/	2F	95	_	5F
48	0	30	96	`	60
49	1	31	97	a	61
50	2	32	98	b	62
51	3	33	99	c	63
52	4	34	100	d	64
53	5	35	101	e	65
54	6	36	102	f	66
55	7	37	103	g	67
56	8	38	104	h	68
57	9	39	105	i	69
58	:	3A	106	j	6A
59	;	3B	107	k	6B
60	<	3C	108	l	6C
61	=	3D	109	m	6D
62	>	3E	110	n	6E
63	?	3F	111	o	6F
64	@	40	112	p	70
65	A	41	113	q	71
66	B	42	114	r	72
67	C	43	115	s	73
68	D	44	116	t	74
69	E	45	117	u	75
70	F	46	118	v	76
71	G	47	119	w	77
72	H	48	120	x	78
73	I	49	121	y	79
74	J	4A	122	z	7A
75	K	4B	123	{	7B
76	L	4C	124	|	7C
77	M	4D	125	}	7D
78	N	4E	126	~	7E
79	O	4F	127	DEL	7F

Decimal	Hexadecimal	Symbol	Name
128 (x'80') thru 250 (x'FA')		not used	
251	FB	SB	Start buffer
252	FC	SVM	Subvalue Mark
253	FD	VM	Value Mark
254	FE	AM	Attribute Mark
255	FF	SM	Segment Mark

E User Exits

The following user routines supplied with an Ultimate system may be used as user exits from BASIC in the ICONV and OCONV functions.

User Exit	Description
U307A	Puts terminal to sleep until a specified time; equivalent to RQM and SLEEP statements
U407A	Puts terminal to sleep for a specified period of time; equivalent to RQM and SLEEP statements
U50BB	Returns current user's line number and account; line number can also be returned by @USERNO or SYSTEM(19); account can also be returned by @WHO or SYSTEM(26)
U90ED	Initializes tape reel number to 1
U0159	Extended math functions in decimal floating point; equivalent to string number functions SADD, SDIV, SMUL, and SSUB
U018D	Inhibits BREAK key; equivalent to BREAK OFF or BREAK 0
U118D	Enables BREAK key; equivalent to BREAK ON or BREAK 1

Notes

F USERMSG File

The USERMSG file is designed as a multiple data level file with a level for each language translation on the system. The USERMSG file enables system users to create custom messages for their applications, which can then be translated by the UltiKit multi-lingual process.

The USERMSG file can be used with the BASIC USERTEXT function, which is described in Chapter 3, BASIC Statements and Functions. Items can also be printed using the system command PRINT-ERR.

USERMSG Item Format

The format of item.ids in the USERMSG file is up to the user.

Each line in a USERMSG item must conform to a general format:

code{text}

The valid codes are as follows:

Code	Meaning
A	inserts the next parameter from the list of parameters passed by USERTEXT
A(n)	inserts the next parameter as above, but left justified in a field of 'n' blanks
AM	inserts attribute mark
D	inserts the current date
E	inserts the item.id enclosed in brackets
H	inserts the text following the H; does not include a <CR> or line feed
H+	used at the end of the USERMSG item only to suppress final <CR> and line feed that is normally output.

L	inserts <CR> and line feed.
L(n)	inserts n-1 blank lines.
R(n)	inserts the next parameter as A (above), but right justified in a field of 'n' blanks
S(n)	Sets the output buffer pointer to location 'n'.
T	inserts the current time
X	Skips a parameter in the list of parameters passed by USERTEXT

Sample USERMSG item

```
item.id  Welcome
   001   E
   002   H,
   003   A
   004   L
   005   HIt is
   006   D
```

```
X = USERTEXT('Welcome',CUSTNAME)
```

result:
This returns the text of "Welcome" after inserting the value of CUSTNAME into the message:

[Welcome], *custname*
It is 21 Jun 1989

G Revision 200 New Features

Revision 200 of the Ultimate Operating System, Release 10, includes the following new and enhanced features for the Ultimate BASIC language:

- new reserved words
- addition of predefined variables and system variables
- change to file variables
- extension of arithmetic operators to dynamic arrays
- extension of comments to CALL, COMMON, DIM, and EQUATE statements
- new compiler directives:
  $COMPATIBILITY
  INCLUDE
  $INSERT
- new and enhanced BASIC statements and functions
- BASIC compiler changes
- new BASIC debugger

For more information, see the *O/S Revision 200 New Features* guide.

Because of these changes, **all BASIC programs from previous revisions of the operating system must be recompiled** after upgrading to Revision 200 or later. When recompiling BASIC programs, be sure to compile using the same options, such as S (suppress symbol table), that were used in the original compilation. For information on upgrading, see the Ultimate Upgrade Document for your system and revision.

Statements and Functions

The following statements and functions were added or revised for Revision 200 of the Ultimate Operating System. If you are using a revision prior to 200, the commands that are marked New are not available at all. The commands that are marked enhanced are available in prior revisions, but not all features are available.

assignment statement	Enhanced
BREAK{KEY} statement	Enhanced
CLEARDATA statement	New
CLEARSELECT statement	New
COMMON statement	Enhanced
CONVERT statement	New
CRT statement	New
DIM statement	Enhanced
ECHO statement	Enhanced
EQU{ATE} statement	Enhanced
ERRTEXT function	New
EXECUTE statement	Enhanced
FIELD function	Enhanced
FMT function	New
FOR/NEXT statement	Enhanced
INMAT function	New
INPUT statement	Enhanced
MATCH{ES} operator	Enhanced
MATCHFIELD function	New
MATPARSE statement	New
MATREAD statement	Enhanced
OPEN statement	Enhanced
PAGING statement	New
READT statement	Enhanced
RELEASE statement	Enhanced
REMOVE statement	New
REUSE function	New
RQM statement	Enhanced
SELECT statement	Enhanced
SLEEP statement	New
SORT function	New
SOUNDEX function	New
SUBROUTINE statement	New

SUM function	New
SYSTEM function	Enhanced
TRAP ON THEN CALL statement	New
TRIM function	Enhanced
USERTEXT function	New
WRITE statement	Enhanced
WRITET statement	Enhanced

Compiler Changes

The following changes have been made to the BASIC compiler:

- Compilation is faster and more efficient; for example, large programs generally compile at least twice as fast as previously
- Error messages produced by the BASIC compiler have been changed and now give more information when an error is encountered; the compiler also indicates where on the line it was scanning when it noted the error.
- BASIC object code size has been increased to 57,534 bytes.
- The BASIC and COMPILE verbs, which are used to compile BASIC programs, have been changed as follows:
  - The A option, which provided a listing of assembled code, is no longer available
  - The E option, which provided an errors only listing, has been removed; the errors only listing is the default
  - The F option can be used with the M option to list internal variables and labels, including those created by IF/THEN and FOR/ NEXT loops; internal variables and labels are displayed preceded by an asterisk.
  - An I option has been added, which, if the L option is also specified, displays lines from $INCLUDEd programs as part of the listing
  - The format and information displayed by the M option has changed

BASIC Debugger

The following lists the new and enhanced BASIC debugger commands.

Bvoc{&voc}	enhanced
BYE	new
C	new
D	enhanced
G{n}	enhanced
H	new
HX	new
K{{/}var}	enhanced
O	new
R	new
S	new
U{n}	new
V	enhanced
*	new
/m<a{,v{,s}}>	new

Notes

Index

- D -

- E -

-G/H-

- I -

- L -

- M -

- N -

- V -

- W -

Notes

Reader Comment Form

Ultimate welcomes your comments. If you find a problem or error in this manual, or can suggest an improvement, please complete this form. Please attach additional sheets, if necessary.

Name of Manual: **Ultimate BASIC Language Reference Guide**

Document No.: **6929-3** Date: _____

Comments

FROM:

Name: _____ System Number: _____

Company: _____

Address: _____

City: _____ State: _____ Zip: _____

Fold and tape. Please do not staple.

Place
Stamp
Here

THE ULTIMATE CORP.
717 Ridgedale Avenue
East Hanover, NJ 07936
Attn: Documentation Manager

Fold and tape. Please do not staple.

Ultimate
THE ULTIMATE CORP

Problem Identification Form QC#

Name Date

System No. Phone # Release Affected

Area Affected (circle one): Async, BASIC, Bisync, Docu, Editor, OS, Passthru, Proc/PROVERB, Recall/RETRIEVE, RPL, Runoff, Security, Spooler, System, Tape, TCL, UltiCalc, UltiKit, UltiLink, UltiMation, UltiNet, UltiPlot, UltiProc, UltiWord, UltiWriter, Update/REVISE, Upgrade

Hardware Platform (circle one): 1400, Bull, Bull DPX/2, HP, IBM, LSI, RS/6000, RT, SEQ, VAX

Is problem reproducible? (Y/N) _____
Example supplied? (Y/N) _____
Tape supplied? (Y/N) _____

If problem is reproducible, please supply example, T-Dump of program or proc, etc. to enable immediate processing.

Detailed description and/or steps used to recreate problem. (Please print clearly.)

System Model _____ CRT Models _____
Memory Size _____ Printer Models _____
# of Ports _____ # of Parallel Printers _____ # of Serial Printers _____
Kernel Rev. _____

For Ultimate Use Only

Date Verified _____ Priority _____
Verified by _____
Category: 1400, Bull, Comm, Docu, IBM, LSI, OS, SEQ, SIG, ULT/ix, Upgrade, Util, VAX

Response/Status

FROM:

Company: _____

Address: _____

City: _____ State: _____ Zip: _____

Fold and tape. Please do not staple.

Place
Stamp
Here

THE ULTIMATE CORP.
717 Ridgedale Avenue
East Hanover, NJ 07936
Attn: Technical Support

Fold and tape. Please do not staple.

Ultimate Technical Support Suggestion Form

Priority _____ *Verified By* _____ *Date Received* _____ *QC Number* _____

For Ultimate Use Only

Category Affected: Systems, Applications, Other _____

Release Affected _____ Date Submitted _____

Contact Name _____

Your System No. _____ Telephone No. _____

Detailed Description

System Configuration (memory size, number of ports, types of terminals and printers, etc.)

Response/Status Date _____

FROM:

Company: _____

Address: _____

City: _____ State: _____ Zip: _____

Fold and tape. Please do not staple.

Place
Stamp
Here

THE ULTIMATE CORP.
717 Ridgedale Avenue
East Hanover, NJ 07936
Attn: Technical Support

Fold and tape. Please do not staple.

Ultimate
THE ULTIMATE CORP.

717 Ridgedale Avenue
East Hanover
New Jersey 07936
201/ 887 9222
FAX 201/ 887 9546