ULT

BASIC

THE ULTIMATE CORP.

ULTIMATE BASIC LANGUAGE

REFERENCE GUIDE

HOW TO ORDER THIS GUIDE

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

For additional copies, please call your dealer or the ULTIMATE Corporation.

Operating System Release 10 - Rev 140
Document No. BAS-01

HOW TO USE THIS MANUAL

This manual is intended as a reference for programmers using the ULTIMATE BASIC programming language. Although not a tutorial, it covers all aspects of using BASIC with the ULTIMATE system file structure and operating system. The material is presented in a structured format, with text and graphics integrated into single-topic units.

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 directives, 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 extended arithmetic package (floating point and string).

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

Chapter 4 explains the testing and debugging procedure and each command in the BASIC 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 special applications.

The appendixes list error messages, ASCII codes, debugger commands and messages.

How the manual is formatted

This manual is presented in a structured format. Each individual topic is an independent unit with its own headline, summary, text, and one or more exhibits.

With a structured format the reader can easily locate the specific topic(s) needed, and all pertinent information is included within the unit.

All topics are numbered within their section, except for the BASIC statements and functions in Chapter 3. The statements and functions are in alphabetical order for easy reference. Each statement/function name is centered as the topic heading without a topic number.

Each topic typically has one or more exhibits. Figure A is always the first exhibit, Figure B is the second exhibit, and so on.

Conventions used

This manual presents general formats for each of the BASIC statements and intrinsic functions. In presenting and explaining these general forms, the following conventions apply:

Example	Meaning
READ	Words printed in capital letters are required and must appear exactly as shown.
expr	Words printed in lower-case letters are parameters to be supplied by the user (i.e., variables, expressions, etc.).
expr	Parameters are underlined for easy reference in the text explanation below the general form.
{expr}	Braces enclosing a word and/or a parameter indicates that the word and/or parameter is optional and may be included or omitted at the user's option. If an ellipsis (i.e., three dots...) follows the terminating bracket, then the word and/or parameter may be omitted or repeated an arbitrary number of times.
RND(expr) COL1()	All functions require a set of parentheses, which usually enclose a parameter. No space is allowed between the function name and the left parenthesis.

The figures on the opposite page illustrate the general figure identifications and content for the topics describing the BASIC Statements and Intrinsic Functions.

Other conventions used throughout the manual are:

BOLD	Bold face type is used for section and unit headings. It is also used in exhibits to indicate user input as opposed to system-displayed data, and in the Appendix message listings.
<CR>	The <CR> symbol indicates a physical carriage return pressed at the keyboard.

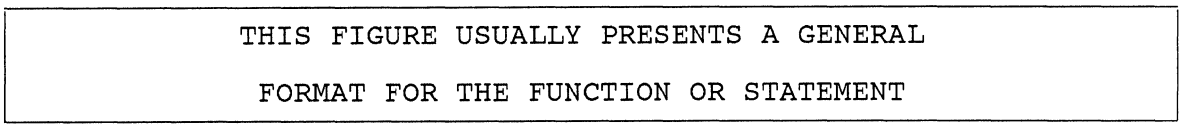


Figure A. General Format

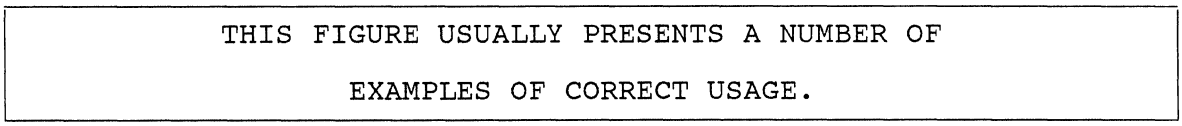


Figure B. Examples of Correct Usage

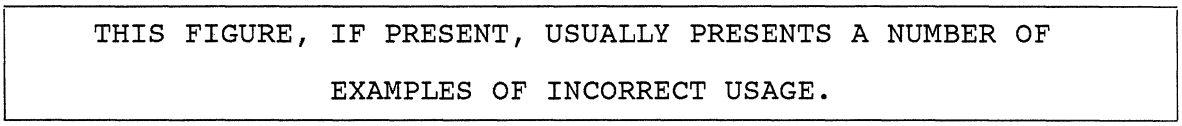


Figure C. Examples of Incorrect Usage

THE ULTIMATE BASIC REFERENCE MANUAL

TABLE OF CONTENTS

CHAPTER 1

OVERVIEW OF PROGRAMMING WITH THE BASIC LANGUAGE

1.1 AN OVERVIEW OF THE BASIC LANGUAGE

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

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

- Optional alphanumeric or numeric statement labels
- Statement labels of any length
- Multiple statements on one line
- Single statements on multiple lines
- Computed GOTO statements
- Complex and multi-line IF statements
- Priority case statement selection
- String handling with variable length strings up to 32,267 characters
- External subroutine calls
- Direct and indirect calls
- Magnetic tape input and output
- Fixed point arithmetic with up to 15 digit precision
- 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
- Shared source code between programs
- Linked programs

Figure A lists the BASIC statements. The BASIC intrinsic functions are listed in Figure B. Figures C and D list the BASIC compiler directives and redirection variables, respectively. All terms listed are BASIC "keywords" and cannot be used as variable names.

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

Figure A. BASIC Statements

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

Figure B. BASIC Intrinsic Functions

	$CHAIN	$INCLUDE	$NODEBUG	$*

Figure C. BASIC Directives

ARG.	MSG.	SELECT.	IN.	OUT.

Figure D. BASIC Redirection Variables

1.2 The File Structure of BASIC Source Programs

BASIC source programs are stored as items in disk files. Object code is referenced through pointer items in file dictionaries.

BASIC source programs are stored as items in the data section of a disk file. The compiler generates pointers to object code in the dictionary section of the file. In order to compile programs, the data and dictionary sections must be distinct files.

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, please refer to Chapter 4, Testing and Debugging BASIC Programs.)

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

Attribute	Contents
0 (item-id)	Program name
1	CC
2	Starting FID of object code
3	Number of frames of object code
4	(null)
5	Time and date of compilation

The term "FID" stands for "frame-id", or frame number. Attributes 0 through 4 are protected by the system against alterations by the Editor or any other file-updating program.

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 using the T-DUMP verb 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.

1.3 The Components of a BASIC Program

A BASIC program is comprised of BASIC statements. A program may also include directives that are interpreted and used by the compiler.

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.)

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

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

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.

Any BASIC statement may begin with an optional statement label. A statement label is used so that the statement may be referenced from other parts of the program. A statement label may be either alphanumeric or numeric. Numeric statement labels may be any constant whole 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. When an alphanumeric label is used, it must be followed by a colon before the statement which it labels. (The colon is optional with numeric labels.) The following subroutine has a statement label of INPUTLOOP and references two other labels:

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

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

```
TOP:

     GOSUB DOITAGAIN
```

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 detailed in Chapter 3.)

A BASIC program can also include compiler directives. Directives always begin with "$". They appear similar to BASIC statements, but they affect the way a program is compiled, not the way it runs. (For details, see the next topic.)

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

A simple BASIC program is illustrated in Figure A to show overall program format. Figure B illustrates the same program with a number of Remark statements and a null line added for clarity.

The user should note that a BASIC program, when stored, constitutes a file item, and is referenced by its item-id. The item-id is the name given to the program when it is created via the EDITOR; refer to Section 1.5, entitled "The Process of Creating and Compiling BASIC Programs". An individual line within a BASIC program constitutes an attribute.

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

Figure A. 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; * START OVER
     END
```

Figure B. BASIC Program With Remark Statements

1.4 Compiler Directives within a BASIC Program

Compiler directives can be included in programs just like BASIC statements. Any line in a BASIC source program which begins with "$" is interpreted as a compiler directive and not a BASIC statement.

Compiler directives appear similar to BASIC statements, but they affect the way a program is compiled, not the way it runs. Each type of directive is detailed below.

$INCLUDE Directive - Sharing Source Code Among Programs

The $INCLUDE directive may be used to include source code stored in one program file item as part of another. The general format of the $INCLUDE directive is:

```
$INCLUDE {filename} itemname
```

If filename is omitted, the file is assumed to be the one containing the program currently being compiled. The itemname specifies the name under which the program item is stored. $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, should not exceed 32768 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.

$CHAIN directive - Linking program file items

The $CHAIN directive can be used to link program file items together at compilation. The general format of the $CHAIN directive is:

```
$CHAIN {filename} itemname
```

If filename is omitted, the file is assumed to be the one containing the program currently being compiled. The $CHAIN directive continues compilation with the specified program itemname. Since any source code appearing after the $CHAIN directive is ignored, the directive should be the last line in the source code.

Note that the final object code size should not exceed 32768 bytes.

$NODEBUG directive - Omitting test capabilities

The $NODEBUG directive may be used after a program has been debugged. It directs the compiler to discard information used during program testing. The general format is:

```
$NODEBUG
```

The $NODEBUG directive causes the compiler to not save the EOL opcodes and the symbol table as part of the object code. (This has the same effect as specifying the "C" and "S" options on the COMPILE or BASIC verb.)

$* directive - Inserting specified text

The $* directive can be used to embed text (such as a copyright notice) in a program's object code. The general format is:

```
$* text
```

The text is specified immediately after the asterisk (*), to the end of the line. The text appears in the object code in a code sequence not generated by any BASIC statement.

1.5 The Process of Creating and Compiling BASIC Programs

A BASIC program is created via the Editor as any other data-file item. Once this source code item has been filed, it is compiled by issuing a COMPILE command (or a BASIC command) at the TCL level.

BASIC programs are created via the ULTIMATE system Editor. To enter the Editor, issue the following command at the TCL level:

```
ED{IT} filename item-id
EEDIT filename item-id
```

The system will then enter the Editor, and you may begin entering the BASIC program. The EEDIT command 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 the TCL level or within the Editor, as shown in Figure B. (See the System Command Guide for further discussion of the EDIT command; see the Editor manual for details about using the Editor.)

The program will be stored in the file specified by filename under the name specified by item-id.

Once the BASIC program has been entered and filed, it may be compiled at the TCL level. Two TCL verbs are available to create the object code: COMPILE and BASIC; either verb may be used since they perform the same operation. The EBASIC form of BASIC must be used to compile programs created with an EEDIT command. EBASIC expands the item to include any spaces that were compressed by EEDIT.

Compiling a program creates object code that can be executed with the RUN verb and can be cataloged. The symbol table is also included with the object code (unless suppressed by the "S" option). The general compile command formats are:

```
COMPILE filename item-list {(options)}
BASIC filename item-list {(options)}
EBASIC filename item-list {(options)}
```

The 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. The options parameter is optional; if used, it must be enclosed in parentheses. An option is specified as an alphabetic character; multiple options used in a single command should be separated by commas.

The valid options are listed below. For detailed descriptions of each, see the next two section topics.

A	Assembled code option
C	Suppress End Of Line (EOL) opcodes from object code.
E	List error lines only.
L	List BASIC program.
M	List map of BASIC program
N	No page
P	Print compilation output on line printer
S	Suppress generation of symbol table
X	Cross reference all variables

The BASIC compiler stores a compiler version number in each program's object code. The run-time system program checks this number each time before running a program to see if it is compatible with the current compiler version. If it is not, the program is not allowed to run; the system issues an error message (B23). The message indicates that the program must be recompiled before it can be run.

Note that compiling does not create an item in the user's Master Dictionary. Master Dictionary items are created by cataloging the compiled program or by using the compile-and-go format in the BASIC source program.

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

```
EEDIT filename item-id

ED{IT} filename item-id

COMPILE filename item-list {(options)}

BASIC filename item-list {(options)}

EBASIC filename item-list {(options)}
```

Figure A. General Forms for Editing and Compiling a BASIC Program

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

>ED BP COUNT <CR>    <------------ User edits item 'COUNT'
                                    in file 'BP' (Basic Programs)
New Item
Top
.I <CR>              <------------ User enters input mode and
                                    begins to enter program

001* PROGRAM COUNTS FROM 1-10 <CR>
002    FOR I = 1 TO 10 <CR> <----- Entered with CTL/I (or TAB key)
003      PRINT I <CR>  <-------     pressed once for indentation
004    NEXT I <CR>                 to first tab stop.
005 END <CR>                       |
006 <CR>                           ----- CTL/I (or TAB key) pressed
TOP                                       twice for second tab stop
                                          indentation
.FI <CR>          <-----------
                              |
                              -------- User files item
'COUNT' Filed

>COMPILE BP COUNT <CR> <---------- User issues compile command
 *****
Successful compile; 1 frames used.
```

Figure B. BASIC Program "COUNT" Created (edited), Filed and Compiled

1.6 BASIC Compiler Options: A, C, E, L, N, and P Options

Nine options are available with the BASIC compile statement. Six are described below: They are the "A" for assembled code, the "C" for suppression of end of line opcode, "E" for the listing of error lines only, the "L" for the listing of the program during compilation, the "P" for routing output to the printer, and the "N" option for no paging. The next topic describes the remaining three compiler options.

The general forms of the BASIC compile command are:

```
BASIC filename item-list {(options)}
COMPILE filename item-list {(options)}
EBASIC filename item-list {(options)}
```

Multiple options are separated by commas. The options are:

A The Assembled code option. The "A" option generates a listing of the source code line numbers, the labels and the BASIC opcodes used by the program. This is a 'pseudo' assembly code listing which allows the user to see what BASIC opcodes his program has generated. The hexadecimal numbers on the left of the listing are the BASIC opcodes and the mnemonics are listed on the right. The assembled code listing of the BASIC program "COUNT" (from previous section) is shown, as an example, in Figure B.

C The Compress option. The Compress option suppresses the end-of-line (EOL) opcodes from the object code. The EOL opcodes are used to count lines for error messages. This eliminates 1 byte from the run time object code for every line in the source code. This option is designed to be used with debugged cataloged programs. Any run time error message will specify a line number of 1.

E The 'list error lines only' option. The "E" option generates a listing of the error lines encountered during the compilation of the program. The listing indicates the line number in the source code item, the source line itself and a description of the error associated with the line.

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

N No page. Inhibits automatic paging on terminal when using the "L" and/or "M" options.

P The printer option. The "P" option routes all output generated by the compilation to the Spooler.

OPTIONS	MEANING
A	Assembled Code listing
C	Compress -- EOL Opcodes suppressed from object code item
E	Error lines only listing
L	Listing of source code
M	Map (variable and statement)
N	No page
P	Route compilation output to printer
S	Suppress symbol table
X	Cross reference

Figure A. General Description of Compiler Options

```
SOURCE          BASIC       PSEUDO
CODE            OBJECT      ASSEMBLY
LINE NO.        CODE        CODE

001             01          EOL
002             03          LOADA      I
002             FD          LOAD.      1
002             20          ONE
002             2D          SUBTRACT
002             5F          STORE
002                1001
002             05          LOADN      10
002             03          LOADA      I
002             20          ONE
002             28          FORTEST    2001
002             01          EOL
003             5D          LOAD       I
003             50          PRINTCRLF
003             01          EOL
004             06          BRANCH     1001
004                2001
004             01          EOL
005             01          EOL
006             45          EXIT

[B0] LINE 6 COMPILATION COMPLETED
```

Figure B. Sample Code Conversions During Compilation

1.7 BASIC Compiler Options: M, S, & X Options

This section describes the remaining three options available when issuing the BASIC compile statement. They are the "M" for map, the "S" for suppressing generation of the symbol table, and the "X" for cross reference.

The options are:

M The map option. The "M" option generates a variable map and a statement map, both of which are printed out after compilation. These maps show where the program data has been stored in the user's workspace. The variable map lists the offset in decimal (from the beginning of the seventh frame of the IS buffer) of every BASIC variable in the program. For example, the form:

```
20 xxx 30 yyy
```

shows that the descriptor of variable 'xxx' starts on byte 20 and the descriptor of variable 'yyy' starts on byte 30 of the seventh frame of the IS buffer. Descriptors are 10 bytes in length.

The statement map shows which statements of the BASIC program are contained in which object code frames. Frame 01 is the starting FID stored in the object pointer item. The statement map may be used to determine if frequently executed loops cross frame boundaries.

S The suppress symbol table option. The "S" option 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.

X The cross reference option. The "X" option creates a cross reference of all the labels and variables used in a BASIC program and stores this information in the BSYM file. NOTE: A BSYM file must exist (a modulo and separation of 1,1 should be sufficient). 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 item-id is the variable or label name. The attributes contain the line numbers of where the variable or label is referenced. An asterisk will precede the line number where a label is defined, or where the value of the variable is changed.

No output is generated by this option. An attribute definition item should be placed in the dictionary of the "BSYM" file which allows a cross reference listing of the program to be generated by the command:

```
>SORT BSYM BY LINE-NUMBER LINE-NUMBER
```

1.8 Cataloging BASIC Programs: CATALOG and DECATALOG Verbs

Compiled BASIC programs can be cataloged and used as commands at the TCL level. They can also be decataloged. The CATALOG and DECATALOG verbs are used to create and delete TCL commands for compiled BASIC programs.

The general form of the CATALOG command is:

```
CATALOG filename item-list {(L)}
```

The filename specifies the file containing programs to be cataloged. Item-list consists of one or more program names (item-ids), or "*" to indicate all programs in the file. The (L) option indicates that the program is not to be executed automatically at logon time. When the (L) option is not present, if a program name is the same as an account name, that program will be automatically run whenever the account logs on. (For details, please refer to Section 1.9, "Executing Compiled BASIC Programs".)

Programs to be cataloged must first be compiled. The program may not have the same name as an existing item in the user's Master Dictionary unless that item is also a cataloged program verb. For example, if the BASIC source program "A" is in the Master Dictionary, another program cannot be cataloged with the name "A". If a conflicting item already exists in the user's Master Dictionary, the system will respond with:

```
[415] (item-id) exists on file
```

and the program will not be cataloged.

For each program successfully cataloged, the system responds with

```
[244] (item-id) cataloged
```

Once a program is cataloged, it may be run simply by typing its name at the TCL prompt. CATALOG adds the program name as a verb in the user's Master Dictionary (when not already present) with the following form:

```
1)  PC
2)  E6
3)
4)
5)  filename item-id
```

When the "(L)" option is used with CATALOG, line one of each verb in the Master Dictionary will be "P" instead of "PC". This will inhibit automatic execution of the program at logon time if the program name is the same as the account name.

The DECATALOG verb has the primary purpose of removing the object code from the system. The general form of the DECATALOG command is:

```
DECATALOG filename item-list
```

DECATALOG removes the object programs specified by item-list in the file filename 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 available pool ("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.

External subroutines used with the BASIC CALL statement may also be cataloged, though it is unnecessary when both the subroutine and the calling routine are in the same program file. The CALL statement will first search the Master Dictionary for a catalog verb in order to locate a subroutine's object code. If not found, it will then look for an object pointer in the dictionary of the program file for the calling routine.

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

1.9 Executing Compiled BASIC Programs

All execution of BASIC programs is performed at the TCL level. TCL can interpret a RUN command, a PROC name, and/or a BASIC program name (both source and compiled versions). A compiled BASIC program can be executed by issuing a RUN command. If the program has been cataloged, it can be executed by issuing only the program name. Programs with the same name as an account name can be automatically executed at logon time.

The general format of the RUN command is:

```
RUN filename item-id {argument list} {(options)}
```

The filename and item-id specify the compiled BASIC program to be executed. The optional argument list specifies any parameters that must be passed to the program. If used, the options must be enclosed in parentheses. Multiple options may be separated by commas. Valid options are as follows:

A Abort option. The "A" option inhibits entry to the Basic Debugger under all error conditions; instead, the program will print a message and terminate 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. The "E" option forces the program to enter the Basic Debugger whenever an error condition occurs. The use of this option will force the operator to either accept the error by using the Debugger, or exit to TCL.

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

N Nopage option. The "N" 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.

Issuing a Program Name directly from TCL

A compiled and cataloged program can be executed directly from the TCL level using the following general format:

```
>progname {argument list}
```

The progname 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.

Executing BASIC programs from a PROC or other BASIC program

PROCs (procedures) may be used to perform various tasks from a single integrated "procedure". A TCL command, and special PROC commands are stored within the PROC. The following example illustrates the use of a BASIC program in conjunction with a Recall SSELECT (Sort Select) command.

A PROC named LISTBT is as follows:

```
PQ
HSSELECT BASIC/TEST
STON
HRUN BASIC/TEST LISTIDS
P
```

A BASIC program named LISTIDS is as follows:

```
    OPEN 'BASIC/TEST' ELSE PRINT 'FILE MISSING'; STOP
10  N = 0
20  READNEXT ID ELSE STOP
    PRINT ID 'L#18':
    N = N + 1
    IF N>= 4 THEN PRINT; GO TO 10
    GO TO 20
    END
```

By typing in LISTBT at the TCL level, the PROC LISTBT selects the item-ids contained in file BASIC/TEST and invokes the BASIC program LISTIDS to list the item-ids selected, four to a line, left justified in a field of 18 blanks.

A PROC can be executed automatically at logon time if the PROC name is the same as the logon account name. For further information about PROCs, refer to the ULTIMATE PROC Manual.

As an alternative to using PROCs for job control tasks, users can execute BASIC programs, PROCs, and TCL verbs within a "controlling" BASIC program. The controlling program can use an EXECUTE statement(s), as well as other supporting statements (PUT, GET, SEEK) and a function (EOF) to implement the job control tasks.

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

Executing Programs at Logon Time

When a user logs on, the system will attempt to execute a program in the user's Master Dictionary with the same name as the logon account name. This program may be a PROC, or a compile-and-go BASIC program, or a cataloged BASIC program.

This feature is useful to run a standard job control sequence or present a custom tailored menu of choices to the user.

In some cases, users may need to catalog a BASIC program with the same name as the account name but NOT to run it automatically at logon time. To avoid automatic execution, the program can be cataloged with the L option. For example, if INVENTORY were an account name, the CATALOG command:

```
CATALOG BP INVENTORY (L)
```

would catalog the program, but would not link it for automatic execution whenever a user logs on the INVENTORY account.

For details on cataloging programs, refer to Section 1.8, "Cataloging BASIC Programs: CATALOG and DECATALOG Verbs".

```
TCL COMMAND:      >RUN filename item-id {argument list} {(options)}

PROC:             >PROC1

CATALOGED or
SOURCE PROG:      >progname {argument list}

LOGON:            Logon please:ACCOUNTNAME {argument list}
```

Figure A. Options at TCL Level for Executing BASIC Programs

```
NOTE:  In each example below, the same "RUN PROGRAMS TESTING"
command is interpreted at the TCL level, regardless of the
point of origin of the command.

TCL COMMAND:      >RUN PROGRAMS TESTING  <CR>

PROC:             ...
                  HRUN PROGRAMS TESTING
                  P
                  ...

PROGRAM:          ...
                  EXECUTE "RUN PROGRAMS TESTING"
                  ...

In the example below, the program name "TESTING" is executed if an
account name of "TESTING" has been established.

LOGON:            Logon please:TESTING
```

Figure B. Alternative Ways to Execute a BASIC Program

1.10 Executing BASIC Source (Compile-and-go) Programs

BASIC source programs may be entered as items in Master Dictionaries and treated as compile-and-go verbs or PROCs. Programs and PROCs may be executed automatically at logon time. BASIC source programs can also be executed from within other BASIC programs via the EXECUTE statement.

Compile-and-go

A BASIC source program can be executed from the TCL level without previous compilation. This option, called "compile-and-go", requires only that the source program be entered as an item in a Master Dictionary. These BASIC programs must 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:

```
>progname {argument list}
```

For example:

```
>HELLO
```

would compile and execute the BASIC source program named "HELLO".

The effect of compile-and-go is that of writing a PROC in BASIC, 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.

CHAPTER 2

REPRESENTING DATA: CONSTANTS, VARIABLES, AND EXPRESSIONS

2.1 Representing Data Values: Numbers and Strings

In ULTIMATE BASIC, there are two types of data: numeric and string. Numeric data consists of a series of digits and represents an amount (e.g., 255). String data consists of a set of ASCII characters which may be alphabetic, numeric, and/or keyboard symbols.

Numbers and Numeric Data

A number may contain up to 15 digits, including the digits following the decimal point. In a BASIC program, the PRECISION statement establishes the number of fractional digits. The default PRECISION is 4, so that numbers must be in the range:

```
-14,073,748,835.0000 to 14,073,748,835.0000
```

when a program uses the default PRECISION. To change the acceptable form and range of numbers, the PRECISION statement must be used.

Since a number can have a maximum of "n" fractional digits, where "n" is the PRECISION, the value:

```
1234.567
```

is a legal number if the PRECISION is 3 or 4, but is not a legal number if the PRECISION is 0, 1, or 2. By changing the PRECISION to a value less than 4, the range of the allowable whole numbers is increased accordingly. (For details, see the PRECISION statement, listed alphabetically in Chapter 3.)

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

```
-17000000
-14.3375
```

String Data

A string is represented by a set of characters enclosed in single quotes, double quotes, or backslashes. For example:

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

If a string value contains one string delimiter (', ", or \), then another delimiter must be used to delimit that string. For example:

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

A string may contain from 0 to 32,267 characters (i.e., maximum length of an ULTIMATE file item). Internally, a string is delimited by a Segment Mark (SM), which is a character having a decimal value of 255. No string, therefore, may contain a Segment Mark. Figure A presents a number of valid and invalid string constants.

A string having the format of an ULTIMATE file item is called a "dynamic array". Since values within a file item may change, dynamic array strings usually contain variable, rather than constant, data. Dynamic array strings are explained in the next topic.

Data consisting of only digits may be defined as numeric (e.g., 2.5) or string (e.g., "2.5"); either data type is legal. The choice normally depends on the type of operations and expressions in which the value will be used. For arithmetic usage, the numeric data type is more efficient; for joining (i.e., concatenation), the string data type is more efficient. Either type, however, will be processed with accurate results without programmer intervention.

VALID STRING	INVALID STRING
"ABC%123#*4AB"	ABC123 (i.e., quotes are missing)
'1Q2Z....'	'ABC%QQR" (i.e., either two single quotes or two double quotes must be used)
"A 'LITERAL' STRING"	
'A "LITERAL" STRING'	
'' (i.e., the empty string)	"12345678910 (i.e., terminating double quote missing)
\DEF\	

Figure A. Correct and Incorrect Usage of Strings

2.2 Multi-valued Strings: Dynamic Arrays

A string having the format of an ULTIMATE file item is called a dynamic array. The string 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.

Dynamic arrays are significant in ULTIMATE BASIC because they 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.

Review of ULTIMATE File Structure

An ULTIMATE data file consists of a set of file items. Each item consists of a string that is in item format. Such a string is called a dynamic array.

A dynamic array consists of one or more attributes; multiple attributes are separated by attribute marks (i.e., an attribute mark has an ASCII equivalent of 254, shown as "^"). An attribute, in turn, may consist of one or more values; multiple values in an attribute are separated by value marks (i.e, a value mark has an ASCII equivalent of 253, shown as "]"). Finally, a value may consist of one or more subvalues; multiple subvalues in a value are separated by subvalue marks (i.e., a subvalue mark has an ASCII equivalent of 252, shown as "\"). This structure is summarized in Figure A.

An example of a dynamic array is as follows:

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

where "55", "ABCD", "73XYZ", and "100000.33" are attributes.

The following illustrates a more complex dynamic array:

```
"Q5^AAAA^952]ABC]12345^A^B^C]TEST\12I\9\99.3]2^555"
```

where "Q5", "AAAA", "952]ABC]12345", "A", "B", "C]TEST\12I\9\99.3]2" and "555" are attributes; "952", "ABC", "12345", "C", "TEST\12I\9\99.3", and "2" are values; and "TEST", "12I", "9", and "99.3" are subvalues.

The maximum length of a dynamic array (including attribute marks, value marks, and subvalue marks) is 32,266 characters.

Additional examples of correctly formed dynamic arrays are illustrated in Figure B. For complete details on the ULTIMATE file structure, please see the ULTIMATE system documentation.

Accessing Elements of a Dynamic Array

Individual elements of a dynamic array can be referenced by specifying the appropriate subscript position (attribute, value, and subvalue number) within the string. Attributes, values, and subvalues are numbered beginning with 1. Subscripts are normally written in angle brackets following the string, with the first subscript specifying an attribute, the second subscript (if present) specifying a value within the selected attribute, and the third subscript (if present) specifying a 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, or "ABCD". If Y represents the second dynamic array above, then Y<3,2> = "ABC" and Y<6,2,1> = "TEST".

If a program attempts to access a non-existent attribute, value, or subvalue, the system returns a null string. Each of the elements in the arrays below, for example, would be returned as a null string:

Element	Array
<3>	"Q5^AAAA"
<3,2>	"Q5^AAAA^3"
<3,2,2>	"Q5^AAAA^3]2"

Dynamic arrays may also be referenced with BASIC functions and statements such as EXTRACT, DELETE, INSERT, REPLACE, and LOCATE. For details, please refer to the appropriate function or statement, listed alphabetically in Chapter 3.

ULTIMATE file items are stored as variable data in a BASIC program. A dynamic array, however, is simply any string expression (not necessarily a variable) treated as having the format of a disk file item. This string may be stored in a simple variable or in element(s) of a dimensioned array. See the following topics for more information on variables and dimensioned arrays.

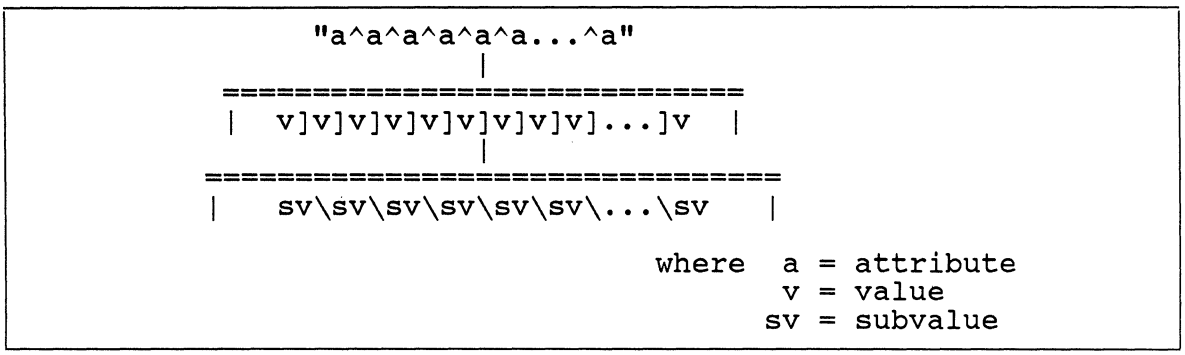


Figure A. General Form of Dynamic Array

ARRAY	EXPLANATION
123^456^789]ABC]DEF	"123", "456", "789]ABC]DEF" are attributes; "789", "ABC" and "DEF" are values.
1234567890	"1234567890" is an attribute.
Q56^3.22]3.56\88\B]2^99	"Q56", "3.22]3.56\88\B]C", and "99" are attributes; "3.22", "3.56\88\B", and "C" are values; "3.56", "88", and "B" are subvalues.
A]B]C]D^E]F]G]H^I]J	"A]B]C]D", "E]F]G]H", and "I]J" are attributes; "A", "B", "C", "D", "E", "F", "G", "H", "I", and "J" are values.

Figure B. Examples of Correct Usage of Dynamic Arrays

2.3 Defining Data Values as Constants or Variables

Within a BASIC program, a numeric or string data value may be represented as either a variable or a constant. A constant is a value that may have an associated name. A variable is a name for a storage location that may have a changing value.

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" when used in a BASIC statement. A constant may also be a named value. In this case, a symbolic name would be equated with a constant value; for example, the name "AM" could be equated to CHAR(254). To improve a program's readability, the name would be used instead of the value in BASIC statements.

The EQUATE statement can be used to associate a name with a value. For details, see the EQUATE statement, listed alphabetically in Chapter 3.

Variables

A "variable" has both a name and a value (or may identify a file). The value of a variable may be either numeric or string, and may change dynamically during the execution of the program. A variable may contain one or more value elements, as in the case of a dynamic array assigned as the value of a variable.

A simple variable is associated with a single storage location, and has only one value at any given time. By contrast, a dimensioned array variable is associated with multiple storage locations, each of which has a separate value and, in general, can function as a simple variable. A particular location (or element) within a dimensioned array is specified by following the array name with subscripts (numbers or other arithmetic expressions) in parentheses. For example, A(10) refers to the tenth element of the one-dimensional array A.

Subscripts in angle brackets are also used to refer to elements of dynamic arrays. If variable X contains a dynamic array, for example, X<3> specifies the third attribute of the dynamic array. However, 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.

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.

2.4 Representing Changing Data Values: Variables

Data values that may change in a BASIC program are defined as variables. The name of a variable refers to a particular data storage area. The value(s) of a variable refer to the current contents of the storage area. Values may be either numeric or string, and may change dynamically throughout the execution of a BASIC program.

Naming Variables

The name of a variable identifies the variable; 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 ending with a period are reserved for ULTIMATE pre-defined variables. Variable names may be of any length.

The following terms would all be valid variable names:

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

BASIC keywords (i.e., words that define BASIC statements, functions, and system variables) may not be used as variable names. The BASIC keywords are listed in the figures of Section 1.1, entitled "An Overview of the BASIC Language".

The name of a variable and its storage location are assigned by the first BASIC statement in a program that uses the name. This is typically an assignment, INPUT, or READ statement, which assigns the variable a value.

Assigning and Accessing Values in a Variable

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 then 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:

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

would print the number "12".

When a variable contains a dynamic array string, each element of the dynamic array can be addressed by specifying its position within angle brackets. The angle brackets "<" and ">" enclose the element identifier, as in <2>. For example:

```
A = B<2>
```

assigns the second attribute of variable B to variable A. And:

```
B<2,6>
```

would access the 6th value of attribute 2 of variable B.

Multi-valued variables are called "dimensioned arrays", or simply "arrays". Dimensioned arrays are variables with a pre-defined number of storage locations assigned by a DIM or COMMON statement. Dimensioned arrays should not be confused with dynamic arrays, which are simply strings in file item format. A dynamic array may be stored in any variable, including an element of a dimensioned array. See the next topic for more information about dimensioned arrays.

The following statements pertain to assigning variables. (For details, see the appropriate statement, listed alphabetically in Chapter 3.)

Name	Purpose
=	(Assignment). Names and assigns a value, and a storage location if needed, to a variable.
EQUATE	Allows one variable to be defined as the equivalent of another variable.
COMMON	Allows certain variables to be allocated storage space before any other variables in the program; also allows for the passing of values between programs.
STORAGE	Allows a program to change the buffer size for storing variables in a program.
DIM	(Dimension). Names and assigns a specified number of storage locations to a multi-valued array variable.

VALID VARIABLE NAME	INVALID VARIABLE NAME
A5	ABC 123 (i.e., no space allowed)
ABCDEFGHI	5AB (i.e., must begin with letter)
QUANTITY.ON.HAND	Z.,$ (i.e., comma not allowed)
R$$$$P$	A-B (i.e., "-" not allowed)
J1B2Z	
INTEGER	
THIS.IS.A.NAME	

Figure A. Correct and Incorrect Usage of Variable Names

2.5 Multi-valued Variables: Dimensioned Arrays

Dimensioned arrays are variables that have been dimensioned by a DIM or COMMON statement and contain a pre-defined number of elements.

Before a dimensioned array may be used in a BASIC program, the maximum dimension(s) of the array must be specified. The DIM or COMMON statement reserves the array's variable name and the number of storage locations. (Please refer to the DIM or COMMON statement, listed alphabetically in Chapter 3.)

A BASIC program can address any element of a dimensioned array as a separate variable. It can assign values to any/all elements in a single statement. When an array is dimensioned, values can be stored in each separate "slot" or element in the array.

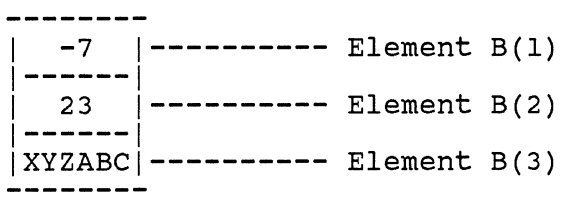
A dimensioned array contains one value per element. For example, Array A has been dimensioned as A(4):

```
-------
|  3  |---- The first element of A has value 3
-------
|  8  |---- The second element of A has value 8
-------
|-20.3|---- The third element of A has value -20.3
-------
| ABC |---- The fourth element of A has string value "ABC"
-------
```

The above example illustrates a one-dimensional array (called a vector). A two-dimensional array (called a matrix) is characterized by having rows and columns. For example, Array Z has been dimensioned as Z(3,4):

```
          COL.1  COL.2  COL.3  COL.4
          ---------------------------
Row 1 |   3   | XYZ |    A   | -8.2 |
      |-------------------------------|
Row 2 |   8   | 3.1 |   500  | .333 |
      |-------------------------------|
Row 3 |   2   |  -5 | Q123 |   84   |
      ---------------------------
```

Any array element may be accessed by specifying its position in the array. This position is like an offset from the beginning of the array. In specifying an element, the user must have one offset or subscript for each dimension of the array. For example, this is Array B:



In this example element B(1) has a value of -7, while element B(3) has a string value of "XYZABC". For a two-dimensional array (matrix) the first subscript specifies the row, while the second specifies the column. For example, in array Z above:

```
Element Z(1,1)    has a value of 3
Element Z(2,3)    has a value of 500.
```

When reading from ULTIMATE disk files into dimensioned arrays, the MATREAD or MATREADU statement may be used to assign each attribute of an item to an individual vector element. Conversely, the MATWRITE or MATWRITEU statement may be used to construct an item from a vector when writing to a file. (For details, see the appropriate statement, listed alphabetically in Chapter 3.)

2.6 Arithmetic Expressions: Standard Arithmetic

Expressions are formed by combining operators with variables, constants, or BASIC intrinsic 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, i.e., the adjacent constants, variables, or intrinsic functions.

Arithmetic expressions are formed by using the arithmetic operators listed in Figure A. 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.

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 will be performed first. Figure A shows the precedence of the arithmetic operators. 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: -R/A+B*C. The division and multiplication operators have the same (high) precedence; since the division operator is leftmost, it is evaluated first (i.e., R/A = result 1). The expression then becomes: -(result 1)+B*C. The multiplication operation is performed next (i.e., B*C = result 2). The expression then becomes: -(result 1)+(result 2). The negation and addition operators have the same precedence; since the negation operator is leftmost, it is evaluated first (i.e., -(result 1) = result 3). The expression then becomes: (result 3)+(result 2). The addition is then performed, yielding the final result.

Using some figures in the above expression illustrates, for example, that the expression -50/5+3*2 evaluates to -4.

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

Arithmetic operators may not appear adjacent to one another. This means, for example, that 2*-3 is not a valid expression, though 2*(-3) is.

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 will be printed (refer to APPENDIX B - BASIC RUN-TIME ERROR MESSAGES) and zero will be assumed for the string value.

The following expression, for example, evaluates to 123:

```
123 + "ABC"
```

OPERATOR SYMBOL	OPERATION	PRECEDENCE
^	exponentiation	1 (high)
*	multiplication	2
/	division	2
+	addition or identity	3
-	subtraction or negation	3 (low)

Figure A. Arithmetic Operators

CORRECT USE	EXPLANATION
2+6+8/2+6	Evaluates to 18
12/2*3	Evaluates to 18
12/(2*3)	Evaluates to 2
A+75/25	Evaluates to 3 plus the current value of variable A.
-5+2	Evaluates to -3
-(5+2)	Evaluates to -7
8*(-2)	Evaluates to -16
5 * "3"	Evaluates to 15

Figure B. Examples of Correct Usage of Arithmetic Expressions

2.7 Extended (Floating Point and String) Arithmetic

For some applications, the precision or magnitude of regular arithmetic is not sufficient to represent the data or functions needed. For these cases, ULTIMATE BASIC supports two types of extended arithmetic: floating point and string.

For both floating point arithmetic and string arithmetic, the standard operations of add, subtract, multiply, divide, and compare have been implemented as intrinsic functions within BASIC. These arithmetic functions are not constrained by the PRECISION statement that affects regular numeric values and arithmetic. They can handle values of a much greater precision and magnitude.

Values to be used in floating point or string arithmetic must be specially-formatted strings. Intrinsic functions are provided which convert regular numeric or string values of constants or variables to strings of the required format. Another set of intrinsic functions may be used after string or floating point operations to convert the results back to regular numeric or string values.

"String arithmetic" means any of the supported string-based arithmetic functions. Whereas regular arithmetic operates on binary numeric values, converting numeric strings to binary as necessary, string arithmetic operates directly on numeric strings, converting binary constants or variables to internal string equivalents if required.

Once numbers are defined as "strings", they can be used in string arithmetic without any conversions. Numeric values (non-strings) are, in most cases, handled successfully; the system converts them to their equivalent string values before performing the arithmetic.

A string number can have virtually any magnitude and any precision, including the entire range of numbers that are valid with the standard arithmetic, plus much more. The following examples are all valid string numbers:

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

The add, subtract, compare, and multiply functions can handle any magnitudes and precisions of string numbers. The result of a divide function is limited to 14 significant digits.

"Floating point arithmetic" means any of the supported floating point-based arithmetic functions. Before a floating point function can be used, numeric arguments must be converted to floating point string representation. Special float and fix functions are used to allow conversion of numbers to floating point string and back to regular string

format after the arithmetic function(s) are completed.

Floating point numbers have a considerably different format from that of standard 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. The following examples show the floating point string representation of various numbers:

FLOATING POINT STRING VALUE	NUMBER
"0E0"	0
"1E0"	1
"1E3"	1000
"1E-20"	.00000000000000000001
"-1234567890123E-5"	-12345678.90123
"98765432109876E-13"	9.98765432109876
"-28855E-2"	-288.55

Guidelines for Using Extended Arithmetic Functions

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.

ULTIMATE BASIC provides twelve intrinsic functions to handle floating point and string arithmetic operations:

Operation	String Function	Floating Point Function
Addition	SADD	FADD
Subtraction	SSUB	FSUB
Multiplication	SMUL	FMUL
Division	SDIV	FDIV
Comparison	SCMP	FCMP
Convert (Float)		FFLT
Convert (Fix)		FFIX

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

2.8 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: a string constant, a variable with a string value, a sub-string, or a concatenation of string expressions. String expressions may be combined with arithmetic expressions.

A sub-string is a set of characters which makes up part of a whole string. For example, "SO.", "123", and "ST." are sub-strings of the string "1234 SO. MAIN ST." Sub-strings are specified by a starting character position and a sub-string length, separated by a comma and enclosed in square brackets (see Figure A). For example, if the current value of variable S is the string "ABCDEFG", then the current value of S[3,2] is the sub-string "CD" (i.e., the two character sub-string starting at character position 3 of string S). Furthermore, the value of S[1,1] would be "A", and the value of S[2,6] would be "BCDEFG".

If the "starting character" specification is past the end of the string value, then an empty sub-string value is selected (e.g., if A has a value of 'XYZ', then A[4,1] will have a value of ''). If the "starting character" specification is negative or zero, then the first character is assumed (e.g., if X has a value of 'JOHN', then X[-5,1] will have a value of 'J').

If the "sub-string length" specification exceeds the remaining number of characters in the string, then the remaining string is selected (e.g., if B has a value of '123ABC', then B[5,10] will have a value of 'BC'). If the "sub-string length" specification is negative or zero, then an empty sub-string is selected (e.g., B[5,-2] and B[5,0] both have a value of '').

Concatenation operations may be performed on strings. Concatenation is specified by a colon (:) or CAT operator. The concatenation of two strings (or sub-strings) is the appending of the characters of the second operand onto the end of the first. For example:

```
"AN EXAMPLE OF " CAT "CONCATENATION"
```

evaluates to:

```
"AN EXAMPLE OF CONCATENATION"
```

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

The precedence of the sub-string operator (square brackets), however, is higher than that of the arithmetic operators. So in an expression like A+B[7,3], a sub-string of B will be converted to a numeric value and then added to the value of A.

The concatenation and sub-string operators both consider their operands to be string values. If numeric values are used, the system converts them into equivalent string values before performing the operation. For example:

```
56:"ABC" concatenates to "56ABC"
```

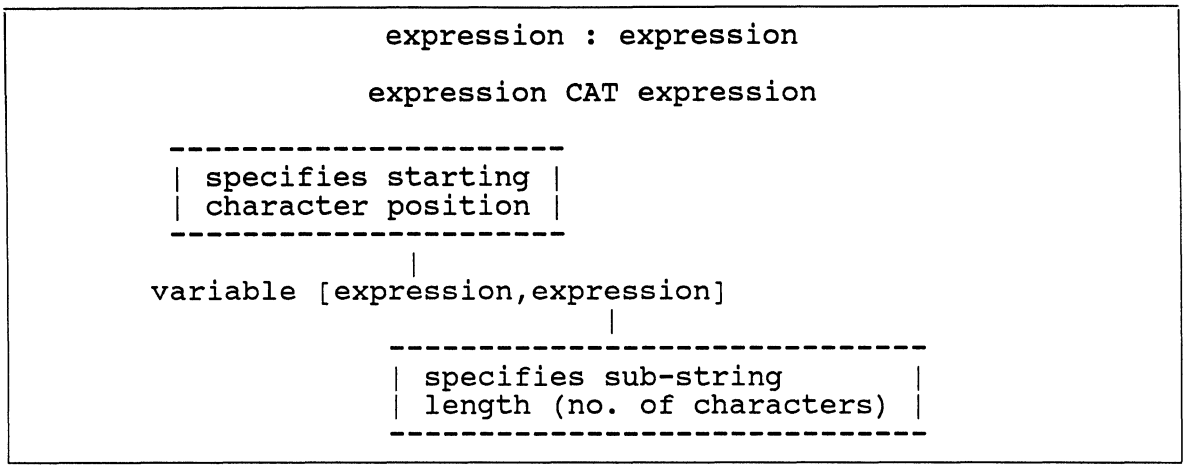


Figure A. General Form of Concatenation and Sub-strings

NOTE: For the following examples:
A = ABC123
Z = EXAMPLE

CORRECT USE	EXPLANATION
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 "93" (i.e., 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".

Figure B. Examples of String Expressions and Arithmetic

2.9 Format Strings: Numeric Mask and Format Mask Codes

Expressions may be formatted by the use of format strings. A format string immediately following a variable name or expression specifies that the value will be formatted as specified by the characters within the format string.

A format string may contain a numeric mask of up to 7 characters and/or a format mask. It is virtually identical to the Recall Mask Conversion Code, and may be used to format both numeric and non-numeric strings. The format string has the following general form:

"{j}{n{m}}{Z}{,}{c}{$}{(format mask)}"

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 is enclosed in parentheses.

The format string may be used as a literal, or it may be assigned to a variable. In either case the format string or variable immediately follows the expression it is to format. The resultant formatted value may be used anywhere an expression is permitted, including an assignment statement which stores a variable's formatted value back into the same variable or to a new variable, and in PRINT statements of the form: PRINT X "format string". Formatting has higher precedence than concatenation, but lower than sub-string and arithmetic operations.

Figure A gives an explanation of the numeric mask and format mask codes. The numeric mask is represented by the symbols: j, n, m, Z, ,, c and $, which control justification, precision, scaling, and credit indication. The format mask controls field length and fill characters. It may consist of any combination of field specifications and literal data. Each field specification consists of a format character optionally followed by a numeric field length specification, such as "#3" or "%5". The format characters are "#", "*" and "%". Field lengths must not exceed 99. Any other character in the format field, including parentheses, may be used as a literal character.

> NOTE: If a dollar sign is placed outside of the format mask, it will be output just prior to the value, regardless of the filled mask. If a dollar sign is used within the format mask it will be output in the leftmost position regardless of the filled field.

Figure B shows the general form and a summary of the codes. Figures C and D show correct and incorrect format strings.

NUMERIC MASK CODES:

j specifies justification. May specify "R" for right justification or "L" for left justification. Default justification is left.

n is a single numeric digit defining the number of digits to print out following the decimal point (with rounding). If n = 0, the decimal point will not be output following the value.

m is an optional 'scaling factor' specified by a single numeric digit which 'descales' the converted number by the 'mth' power of 10. Because BASIC assumes 4 decimal places (unless otherwise specified by a PRECISION statement), to descale a number by 10 m should be set to 5, to descale a number by 100, m should be set to 6, etc.

Z is an optional parameter specifying the suppression of leading zeros.

, is an optional parameter for output which inserts commas between every thousands position of the value.

c The following five symbols are Credit Indicators which are optional parameters of the form:

- 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 to follow positive or zero values.
- E Causes negative values to be enclosed with a "<.....>" sequence; a blank follows positive or zero values.
- N Causes the minus sign of negative values to be suppressed.

$ Is an optional parameter for output which appends a dollar sign to the leftmost position of the value, prior to conversion.

FORMAT MASK CODES:

#n specifies that the data is to be filled on a field of 'n' blanks.

*n specifies that the data is to be filled on a field of 'n' asterisks.

%n specifies that the data is to be filled on a field of 'n' zeros and to force leading zeros into a fixed field.

NOTE: Any other character, including parentheses may be used as a field fill.

Figure A. Explanation of the Format String Codes

GENERAL FORM:

"{j}{n{m}}{Z}{,}{c}{$}{(format mask)}"

NUMERIC MASK

MASK CODE	VALID CODE VALUES	MEANING
j	R or L	Right or Left justification (default is left justification).
n	single numeric	# of decimal places.
m	single numeric	'Descaling' factor.
Z	Z	Suppress leading zeros.
,	,	Insert commas every thousands position.
c	C,D,M or E	Credit indicators.
$	$	Outputs dollar sign prior to value.

FORMAT MASK (enclosed in parentheses)

MASK CODE	EXAMPLE	MEANING
$	$	Outputs a dollar sign in the leftmost position of field.
#n	#10	Fills data on a field of 10 blanks.
%n	%10	Fills data on a field of 10 zeros.
*n	*10	Fills data on a field of 10 asterisks, or on a field of any other specified character.

NOTE: If a dollar sign is placed outside of the format mask, it will be output just prior to the value, regardless of the filled field. If a dollar sign is used within the format mask it will be output in the leftmost position regardless of the filled field.

Figure B. General Form and Summary of Format String Codes

UNCONVERTED STRING (X)	FORMAT STRING	RESULT
X = 1000	V = X"R26"	10.00
X = 1234567	V = X"R27,"	1,234.57
X = -1234567	V = X"R27,E$"	$<1234.57>
X = 38.16	V = "1"	38.2
X = -1234	V = X"R25$,M(*10#)"	***$123.40-
X = -1234	V = X"R25,M($*10#)"	$****123.40-
X = -1234	V = X"R25,M($*10)"	$***123.40-
X = 072458699	V = X"L(###-##-####)"	072-45-5866
X = 072458699	V = X"L(#3-#2-#4)"	072-45-5866
X = SMITH, JOHANNSEN	V = X"L((#13))"	(SMITH, JOHANN)
X = 12.25	Y = "1"; PRINT X Y	12.3
X = 12345	PRINT X "R2"	12345.00
X = 1	INPUT @(2,4):X "R(%%)"	01

Figure C. Examples of Correct Usage of Format String

INCORRECT USAGE	EXPLANATION
V = X"MR26"	MR and ML are the codes for RECALL Mask Conversions. In BASIC use simply R or L.
V = X"RL26"	Both right and left justification cannot be used.
V = X"R212"	The descaling factor may only be a single numeric digit. If necessary, the Precision may be set to zero (i.e. no decimal places) so that a descaling factor of 2 will descale by 100, etc.
V = X"L(#100)"	Fill field should not exceed 99 characters.
V = X"R(9%)"	Format code characters must precede the numeric.
V = X"L*32"	Format string must be enclosed in parentheses.

Figure D. Examples of Incorrect Format Strings

2.10 Relational Expressions

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

The relational operators are listed in Figure A. Note that the MATCH(ES) operator is discussed in a separate topic; all others are discussed below. A relational operation evaluates to 1 if the relation is true, and evaluates to 0 if the relation is false. 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.

For purposes of clarification, relational expressions may be divided into two types: arithmetic relations and string relations. An arithmetic relation is a pair of arithmetic expressions separated by any one of the relational operators. For example:

```
3 < 4       (3 is less than 4) = (true) = 1

3 = 4       (3 is equal to 4) = (false) = 0

3 GT 3      (3 is greater than 3) = (false) = 0

3 >= 3      (3 is greater than or equal to 3) = (true) =1

5+1 > 4/2   (5 plus 1 is greater than 4 divided by 2) =
            (true)=1
```

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 (i.e., if a relational operator encounters one numeric operand and one string operand, it treats both operands as strings). To resolve a string relation, character pairs (one from each string) are compared one at a time from leftmost characters to rightmost. If no unequal character pairs are found, the strings are considered to be 'equal'. If an unequal pair of characters are found, the characters are ranked according to their numeric ASCII code equivalents (refer to the LIST OF ASCII CODES in APPENDIX C of this manual). The string contributing the higher numeric ASCII code equivalent is considered to be "greater" than the other string. Consider the following relation:

```
"AAB" > "AAA"
```

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

If the two strings are not the same length, but the shorter

string is otherwise identical to the beginning of the longer string, then the longer string is considered "greater" than the shorter string. The following relation, for example, is true and evaluates to 1:

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

OPERATOR SYMBOLS	OPERATION
< or LT	Less than
> or GT	Greater than
<= or =< or LE	Less than or equal to
= or EQ	Equal to
# or <> or >< or NE	Not equal to
>= or => or GE	Greater or equal
MATCH or MATCHES	Pattern matching

Figure A. Relational Operators

CORRECT USE	EXPLANATION
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'), so expression evaluates to 1.
"Q" LT 5	ASCII equivalent of Q (X'51') is not less than ASCII equivalent of 5 (X'35'), so expression evaluates to 0.
6+5 = 11	Evaluates to 1.
Q EQ 5	Evaluates to 1, if current value of variable Q is 5; evaluates to 0 otherwise.
"ABC" GE "ABB"	Evaluates to 1 (i.e., C is "greater" than B).
"XXX" LE "XX"	Evaluates to 0.

Figure B. Examples of Correct Usage of Relational Expressions

2.11 Relational Expressions: 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.

The general form of the pattern matching relation is shown in Figure A. The MATCH or MATCHES relational operator 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). The pattern may consist of any combination of the following:

- An integer number followed by the letter N (which tests for that number of numeric characters).

- An integer number followed by the letter A (which tests for that number of alphabetic characters).

- An integer number followed by the letter X (which tests for that number of any characters).

- A literal string enclosed in quotes (which tests for that literal string of characters).

Consider the following expression:

```
DATA MATCHES "4N"
```

This relation evaluates to 1 if the current string value of variable DATA consists of four numeric characters.

If the integer number used in the pattern is 0, then the relation will evaluate to 1 only if all the characters in the string conform with the "specification letter" (i.e., N,A, or X). For example:

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

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

As a further example, consider the following expression:

```
A MATCHES "1A4N"
```

This relation evaluates to 1 if the current string value of variable A consists of an alphabetic character followed by four numeric characters.

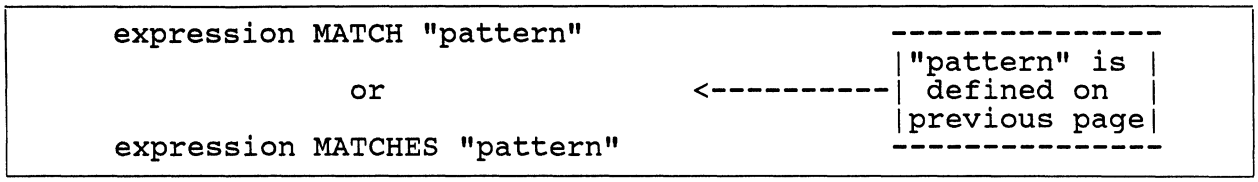


Figure A. General Form of Pattern Matching Relation

CORRECT USE	EXPLANATION
Z MATCHES '9N'	Evaluates to 1 if current string value of variable Z consists of 9 numeric characters; evaluates to 0 otherwise.
Q MATCHES "0N"	Evaluates to 1 if current value of Q is any unsigned integer evaluates to 0 otherwise.
B MATCH '3N"-"2N"-"4N'	Evaluates to 1 if current value of B is, for example, any social security number; evaluates to 0 otherwise.
B="4N1A2N" C MATCHES B	Evaluates to 1 if current string value of C consists of four numeric characters followed by one alphabetic character followed by two numeric characters; evaluates to 0 otherwise.
A MATCHES "0N'.'0N"	Evaluates to 1 if current value of A is any number containing a decimal point; evaluates to 0 otherwise.
"ABC" MATCHES "#N"	Evaluates to 0.
"XYZ" MATCHES "3A"	Evaluates to 1.
"XYZ1" MATCH "4X"	Evaluates to 1.
X MATCHES ''	Evaluates to 1 if current string value of X is the empty string; evaluates to 0 otherwise.

Figure B. Examples of Correct Usage of Pattern Matching Relation

2.12 Logical Expressions

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

The logical operators are listed in Figure A. Logical operators operate on the true or false results of relational or arithmetic expressions. (Relational expressions are considered false when equal to zero, and are considered true when equal to one; arithmetic expressions are considered false when equal to zero, and are considered true when not equal to zero.) 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.

Logical operators act on their associated operands as follows:

A OR B — is true (evaluates to 1) if A is true or B is true; 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; is false (evaluates to 0) if A is false or B is false or both are false.

Consider, for example, the following logical expression:

```
A*2-5>B AND 7>J
```

The multiplication operation has highest precedence, so it is evaluated first (i.e., A*2 = result 1). The expression then becomes:

```
result 1 - 5>B AND 7>J
```

The subtraction operation is next (i.e., result 1 - 5=result 2). The expression then becomes:

```
result 2 > B AND 7>J
```

The two relational operators are of equal precedence, so the leftmost is evaluated first (i.e., result 2 > B=result 3, where result 3 has a value of 1 indicating true, or a value of 0 indicating false). The expression then becomes:

```
result 3 AND 7>J
```

The remaining relational operation is then performed (i.e., 7>J = result 4, where result 4 equals 1 or 0). The final expression therefore becomes:

```
result 3 AND result 4
```

which is evaluated as true (1) if both result 3 and result 4 are true, and is evaluated as false (0) otherwise.

The NOT function may be used in logical expressions to negate (invert) the expression or sub-expression. For details, please refer to the description of the NOT Intrinsic Function, listed alphabetically in Chapter 3.

OPERATOR SYMBOL	OPERATION
AND or &	Logical AND operation
OR or !	Logical OR operation

Figure A. Logical Operators

CORRECT USE	EXPLANATION
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; evaluates to 0 otherwise.
"XYZ1" MATCH "4X" AND X	Evaluates to 1 if the current value of variable X is non-zero; evaluates to 0 if current value of X is 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 either or all variables is 0.

Figure B. Examples of Correct Usage of Logical Expressions

2.13 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. Parenthesized expressions are evaluated before using the results as operands in other expressions.

The precedence of the operators is shown below. Operators with higher precedence are processed first; a series of operators with equal precedence is processed left to right.

OPERATOR SYMBOL	OPERATION	PRECEDENCE
<...>	Dynamic array subscripting	1 (high)
[...]	Sub-string specification	1
^	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
OR or !	Logical OR	8

As an example, the expression:

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

would be evaluated as follows (r1...r8 are results of prior operations):

```
A + B : C[D,(E^F*G)] H MATCH I AND J
A + B : C[D,((r1)*G)] H MATCH I AND J
A + B : C[D,(r2)] H MATCH I AND J
A + B : (r3) H MATCH I AND J
 (r4) : (r3) H MATCH I AND J
 (r4) : (r5) MATCH I AND J
 (r6) MATCH I AND J
 (r7) AND J
 (r8)
```

2.14 How Variables are Structured and Allocated

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 descriptor table contains 'n' entries of 10 bytes each where 'n' is the number of variables (including array elements) in the program. The number of descriptors is limited to 3224. 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 terminated by a SM	for string values of eight characters or less
6-byte pointer to the free space area	for string values with more than eight characters
base (4 bytes), modulo (2 bytes), separation (2 bytes)	for file variables
6-byte pointer to external subroutine code	for external subroutines

Free storage

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

Buffer allocation

Strings longer than eight bytes are placed in storage buffers located in the free storage space. These fixed-length 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 seven (7) bytes per buffer for internal usage. The maximum length for strings in 50-byte buffers, then, is 43 bytes.

When a string requires a new buffer, the system looks in a table of abandoned buffers for a buffer of the appropriate size. If one can't be found, a buffer size is calculated, and a buffer of this size is then allocated to the variable in question. The effect of allocating free storage in this manner is that a buffer is made somewhat larger than the

string it will contain. This allows for larger strings to be stored in the same buffer. This is important because of the allocation procedure.

Initially, free storage is one contiguous block of space. Buffers are allocated from the beginning of the free storage area. When a string is assigned to a variable which exceeds the variable's current buffer size, the buffer is abandoned and a new buffer is allocated from the remaining contiguous portion of free storage. If there is not enough contiguous space for the new buffer, a procedure called 'garbage collection' takes place. Garbage collection collects the abandoned buffer space and forms a single block of contiguous space. If, after garbage collection takes place, there is still not enough contiguous space, the program is aborted with the message:

```
NOT ENOUGH WORK SPACE
```

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.)

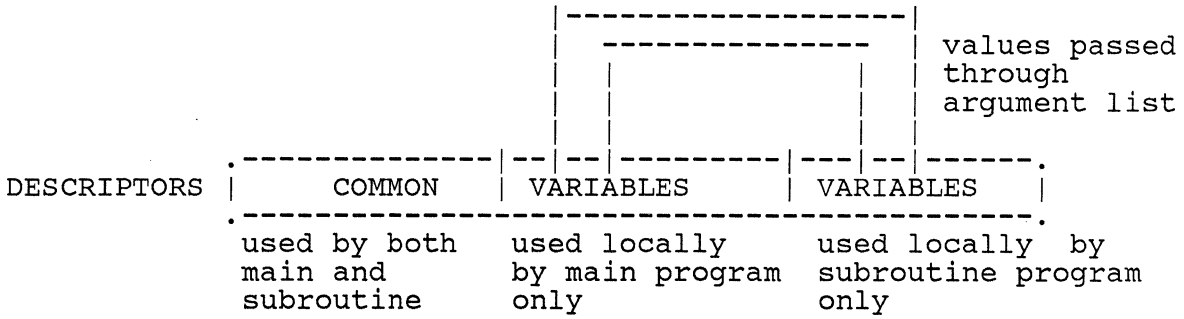
Variable allocation

Variables are allocated descriptors in the following order:

- Common variables
- Simple variables
- Dimensioned variables

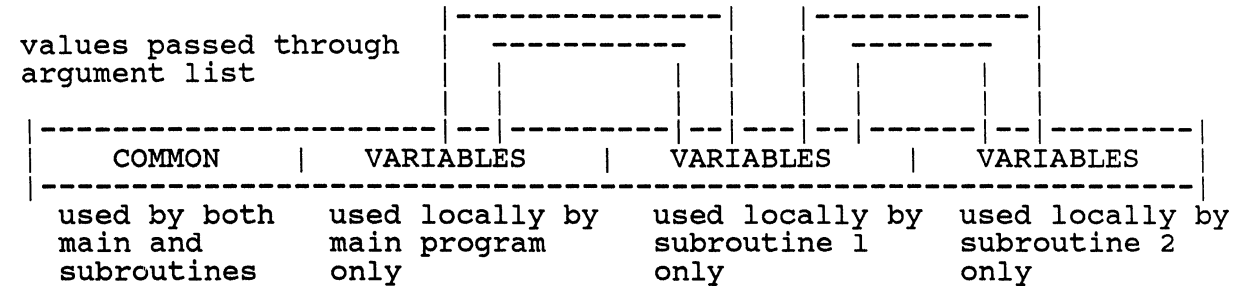
Passing values - subroutines

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



Variables declared as COMMON in both the main program and the subroutine refer to the same locations. There is a one to one correspondence between the variables in both COMMON statements. When values are passed through the argument list 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.

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

Passing values - CHAIN and ENTER programs

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 permissable to CHAIN or ENTER a program that calls a subroutine but it is illegal to CHAIN or ENTER from a subroutine.

NOTES

CHAPTER 3

BASIC STATEMENTS AND FUNCTIONS

3.1 A Summary of the Statements and Functions

3.2 Alphabetical Listing of Statements and Functions

The BASIC Statements:

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

The BASIC Intrinsic Functions:

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

3.1 A Summary of the Statements and Functions

Figure A lists the BASIC statements. The BASIC intrinsic functions are listed in Figure B. Figure C lists the BASIC compiler directives, which are discussed in Section 1.4. Figure D lists the redirection variables used with certain BASIC statements.

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

Figure A. BASIC Statements

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

Figure B. BASIC Intrinsic Functions

$CHAIN	$INCLUDE	$NODEBUG	$*

Figure C. BASIC Compiler Directives

ARG.	MSG.	SELECT.	IN.	OUT.

Figure D. BASIC Redirection Variables

3.2 Alphabetical Listing of Statements and Functions

Each statement and function is described in detail in its own separate topic. The topics are presented in alphabetical order, according to the statement or function name. All statements and functions have been integrated into one alphabetical listing.

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:

```
CALL SUBR1
```

A BASIC intrinsic 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()
```

Each topic about a statement or function begins on a new page. Topics may be presented on one or more pages, as necessary. In general, the text description is covered on the first page, followed by a page of figures. The figures review the general form, which is also covered in the text, and give examples of usage with explanation.

For one-page topics, the text is in the upper portion of the page and the figures are below the text. For multi-page topics, the text precedes the figures. Some complex statements are divided into two separate topics to explain the special cases.

The statements and functions identified by symbols, such as the Assignment (=) statement and the @ function, are listed before the statements and functions with alphabetical names. Thus, the topics begin with:

```
! and * Statement
= Statement
@ Function
ABORT Statement
ABS Function
...
```

and end with:

```
WRITEVU Statement
```

! and *
Statements

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

A Remark statement can be specified in one of three ways: by the characters "REM", by the asterisk character (*), or by the exclamation point (!). Thus, there are three general forms of the Remark statement:

```
REM text ...
! text ...
* text ...
```

REM, !, or * must be placed at the beginning of the statement, but may appear anywhere on a line (e.g., 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. The text may be any arbitrary characters, up to the end of the line.

Remarks are useful, when writing BASIC programs, to summarize, introduce, explain or document the program instructions and routines. A Remark statement allows programmers to place comments anywhere in the program without affecting program execution.

For example:

```
REM THE TEXT FOLLOWING THESE STATEMENTS
!   DOES NOT AFFECT
*   PROGRAM EXECUTION
```

Note that there are extra blank spaces in some of the statements above. These blank spaces appearing in the program line (which are not part of a data item) will be ignored. Thus, blanks may be used freely within the program to enhance the appearance and readability of a program and its comments.

Figure B shows a sample program with Remark statements.

```
REM text              ! text              * text
```

Figure A. General Forms of Remark Statement

```
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
```

Figure B. Sample Program With Remark Statements

= (Assignment) Statement

The = (Assignment) statement is used to assign a value to a variable, or to an element of a dynamic array stored in a variable. The variable may be either a simple variable or an element of a dimensioned array.

The general forms of the Assignment statement are:

```
variable = expression
variable <attr# {,value# {,subval#}}> = expression
```

In the first form, 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. For example:

```
ABC = 500

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

The first statement will assign the value of 500 to the variable ABC. The second statement will asign the value 300 to the variable X2 (i.e., X2 = (ABC+100)/2 = (500+100)/2 = 600/2 = 300).

String values may also be assigned. For example:

```
VALUE = "THIS IS A STRING"

SUB = VALUE [6,2]
```

The first statement above assigns the string "THIS IS A STRING" to variable VALUE. The second statement assigns the string "IS" to variable SUB (i.e., assigns to SUB the two-character sub-string starting at character position 6 of VALUE).

In the second form of the Assignment statement, the dynamic array element in variable specified by attr#, value#, and subval# is replaced by the value of the expression. The values of attr#, value#, and subval#, determine whether the data being assigned is an attribute, a value, or a subvalue. If value# and subval# both have a value of 0 or are both absent, then an entire attribute is assigned. If subval# has a value of 0 or is absent but value# is present, then a value is assigned. If attr#, value#, and subval# are all non-zero, then a subvalue is assigned. If the last (or only) index specified (attr#, value#, or subval#) has a value of -1, then expression is inserted after the last attribute, value, or subvalue.

Arrays must be declared using DIM or COMMON statements before their elements can be addressed in an assignment statement.

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

NOTES:

1. 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. (Please refer to the EQUATE statement, listed alphabetically in this chapter.)

2. All elements in a dimensioned array can be assigned a value(s) by the MAT = assignment statement. (Please refer to the MAT = statement, listed alphabetically in this chapter.)

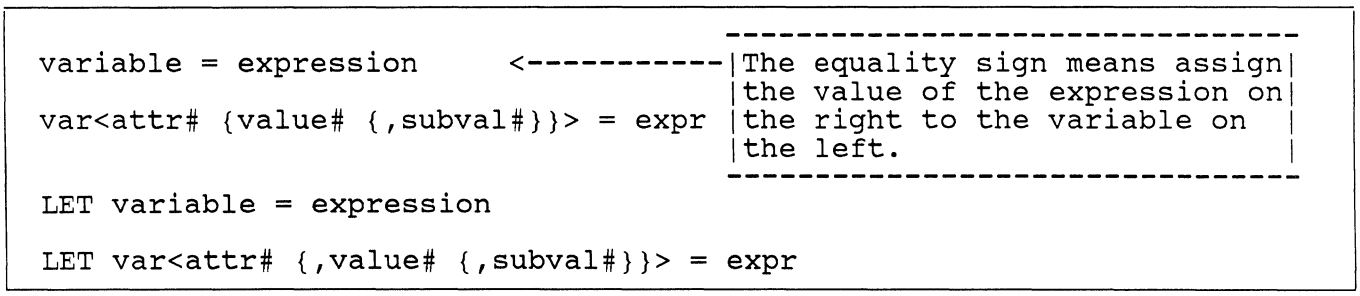


Figure A. General Forms of the Assignment Statement

CORRECT USE	EXPLANATION
X=5	Assigns 5 to X.
X=X+1	Increments X by 1.
ST="STRING"	Assigns the character string to ST.
ST1=ST[3,1]	Assigns sub-string "R" to ST1.
TABLE(I,J)=A(3)	Assigns matrix element from vector element.
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.

Figure B. Correct Examples of Assignment Statement

@
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 PRINT statement.

The general form of the @ function is:

```
@(expression1 {,expression2})
```

If both expression1 and expression2 are present, expression1 specifies the column to which the cursor is to be positioned, and expression2 specifies the row, or line. Columns and rows are numbered starting with zero (0), left to right and top to bottom on the screen. If only expression1 is present, and its value is non-negative, then it is a column specification for the cursor, as before, and the cursor is assumed to remain on the current line. Note, however, that not all terminals support column-only cursor positioning, so the results are not guaranteed. For this reason, both column and row specifications should be used when positioning the cursor. For example:

```
PRINT @(30): "HELLO"
```

This statement prints the message "HELLO" on the current line position of the cursor, starting at column position 30. Another example:

```
PRINT @(10,15): "GOOD-BYE"
```

This statement prints the message "GOOD-BYE" on line 15, starting at column position 10.

If only expression1 is present and its value is negative, then the @ function returns a terminal or printer control string as determined by the table in Figure C.

When positioning the cursor, the values of the expression(s) used in the @ function must be within the row and column limits of the terminal screen.

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. The printer type is shown and changed with the PRINTER command. For more information about these commands, please refer to the ULTIMATE System Commands manual.

Note that not all terminals (or printers attached to terminal auxiliary ports) will 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 normally returned.

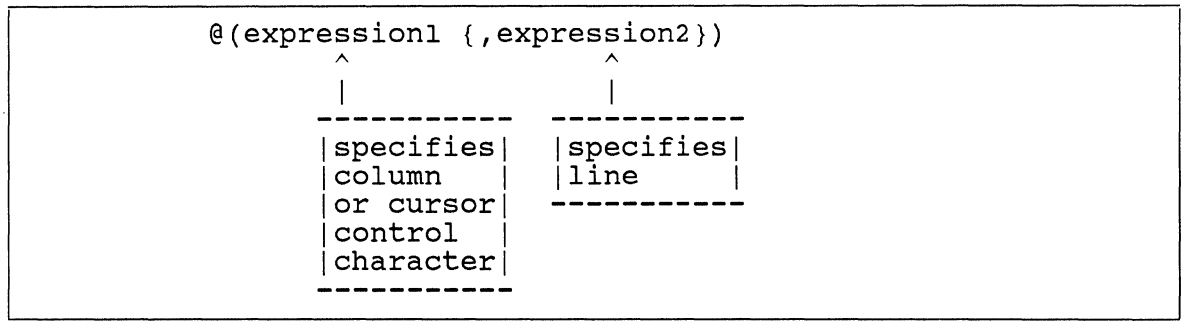


Figure A. General Form of @ Function

CORRECT USE	EXPLANATION
X = 7 Y = 3 PRINT @(X,Y): Z	Prints the current value of variable Z at column position 7 of line 3.
Q = @(3): "HI" PRINT Q	Prints "HI" at column position 3 of current line.
A = 5 PRINT @(A,A+5):A	Prints the value 5 at column position 5 of line 10.
PRINT @(-1)	Clears the screen and positions the cursor at 'home' position.

Figure B. Examples of Correct Usage of @ Function

CODE	EXPLANATION
@(-1)	Generates the clear-screen character; clears the screen and positions the cursor at 'home' (upper left corner of the screen).
@(-2)	Positions the cursor at 'home' (upper left corner).
@(-3)	Clears from cursor positon to the end of the screen.
@(-4)	Clears from cursor position to the end of the line.
@(-5)	Starts blinking on subsequently printed data.
@(-6)	Stops blinking.
@(-7)	Initiates 'protect' field. All printed data will be 'protected', that is, it cannot be written over.
@(-8)	Stops protect field.
@(-9)	Backspaces the cursor one character.
@(-10)	Moves the cursor up one line.
@(-11)	Moves the cursor down one line.
@(-12)	Moves the cursor right one column.
@(-13)	Enables auxiliary (slave) port.
@(-14)	Disables auxiliary (slave) port.
@(-15)	Enables auxiliary (slave) port in transparent mode.
@(-16)	Initiates slave local print.
@(-17)	Starts underlining.
@(-18)	Stops underlining.
@(-19)	Starts inverse video.
@(-20)	Stops inverse video.
@(-21)	Deletes a line.
@(-22)	Inserts a line.
@(-23)	Scrolls the screen display up one line.
@(-24)	Starts boldface type.
@(-25)	Stops boldface type.
@(-26)	Deletes one character.
@(-27)	Inserts one blank character.
@(-28)	Starts insert character mode.
@(-29)	Stops insert character mode.

The following @ function values affect ULTIMATE-supported letter-quality printers:

@(-101,p)	Sets VMI (Vertical Motion Index) to p.
@(-102,l)	Sets HMI (Horizontal Motion Index) to l.
@(-103)	Sets alternate font.
@(-104)	Sets standard font.
@(-105)	Generates a half line-feed.
@(-106)	Generates a negative half line-feed.
@(-107)	Generates a negative line-feed.
@(-108)	Prints black ink.
@(-109)	Prints red ink.
@(-110)	Loads cut sheet feeder.
@(-111)	Selects feeder1.
@(-112)	Selects feeder2.
@(-113)	Selects standard thimble.
@(-114)	Selects proportional space thimble.

Figure C. Explanation of @ Function Negative Values

ABORT
Statement

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

The general form of the ABORT statement is:

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

An ABORT statement may optionally be followed by an error message name, and error message parameters separated by commas. The error message name errnum is a reference to an item in the ERRMSG file. The param parameters are variables or literals to be used within the error message format.

An ABORT statement may be placed anywhere within the BASIC program to indicate the end of one of several alternative paths of logic.

Upon the execution of an ABORT statement, the BASIC program will terminate.

A sample BASIC program illustrating the correct use of the ABORT statement is presented in Figure B. 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"--will be printed, and the program is then terminated.

NOTE: The STOP statement can also be used for program termination. (Refer to the STOP statement, listed alphabetically in this chapter.)

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

Figure A. General Form of ABORT Statement

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

Figure B. Sample Program Using the ABORT Statement

ABS
Function

The ABS function returns an absolute value.

The general form of the ABS function is:

```
ABS(expression)
```

The ABS function generates the absolute numeric value of the expression. For example:

```
A = 100
B = 25
C = ABS(B-A)
```

These statements assign the value 75 to variable C.

```
ABS(expression)
```

Figure A. General Form of ABS Function

CORRECT USE	EXPLANATION
A = ABS(Q)	Assigns the absolute value of variable Q to variable A.
A = 600 B = ABS(A-1000)	Assigns the value 400 to variable B.

Figure B. Examples of Correct Usage of ABS Function

INCORRECT USE	EXPLANATION
Y = "ABCD" Z = ABS(Y)	Expression in ABS functions must be numeric.

Figure C. Example of Incorrect Usage of ABS Function

ALPHA
Function

The ALPHA function returns a value of true (1) if the given expression evaluates to an alphabetic character or string.

The general form of the ALPHA function is:

```
ALPHA(expression)
```

The ALPHA function tests the specified expression for an alphabetic value. If the expression evaluates to a letter or alphabetic string, the function will return a value of true (a value of 1). Otherwise, the ALPHA function will return a value of false (0).

Consider the following example:

```
IF ALPHA(ADAB) THEN PRINT "ALPHABETIC DATA"
```

This statement will print the text "ALPHABETIC DATA" if the current value of variable ADAB is a letter or an alphabetic string.

Alphabetic characters are the 26 letters of the alphabet, in upper or lower case. The empty string ('') is not considered to be an alphabetic string. (It is, however, a valid numeric string.)

```
ALPHA(expression)
```

Figure A. General Form of ALPHA Function

CORRECT USE	EXPLANATION
IF ALPHA(I CAT J) THEN GOTO 5	Transfers control to statement label 5 if current value of both variables I and J are letters or alphabetic strings.
PRINT ALPHA(N) OR ALPHA(M)	Prints a value of 1 if the current value of either M or N is a letter or alpha string.

Figure B. Examples of Correct Usage of ALPHA Function

ASCII
Function

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

The general form of the ASCII function is:

```
ASCII(expression)
```

The string value of the expression is converted from EBCDIC to ASCII, the normal ULTIMATE string representation. For example:

```
A = ASCII(B)
```

The value in variable B is assumed to be in EBCDIC, and is converted to its equivalent ASCII value. The ASCII value is stored in variable A.

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

```
ASCII(expression)
```

Figure A. General Form of ASCII Function

CORRECT USE	EXPLANATION
READT X ELSE STOP Y = ASCII(X)	Reads a record from the magnetic tape unit and assigns value to variable X. Assigns ASCII value of record to variable Y.

Figure B. Example of Correct Usage of ASCII Function

Assignment
Statements

Assignment statements assign values to variables. There are two forms: the = statement and the MAT = statement.

The = statement assigns a value to a simple variable. A simple variable is contained in one storage location. Please refer to the = (Assignment) statement, listed in the first part of this chapter. (Statements/functions idenfified by symbols precede the statements/functions beginning with "A".)

The MAT = statement assigns values to dimensioned arrays. A dimensioned array contains one storage location for each element in the array. Please refer to the MAT = (Assignment) statement, listed alphabetically in this chapter.

BEGIN CASE
Statement

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

The general form of the BEGIN CASE statement is:

```
BEGIN CASE
```

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

BREAK (ON/OFF)
Statement

The BREAK ON and BREAK OFF statements control the BREAK key on the terminal through a BASIC program.

The general forms of the BREAK statement are:

```
BREAK OFF

BREAK ON
```

The BREAK OFF statement disables the BREAK key on the terminal. When disabled, pressing the BREAK key will not be able 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 so as to allow interrupting a program, going to the BASIC debugger, etc.

Note that these commands increment/decrement the BREAK inhibit counter. Since these are cumulative, an equal number of BREAK ON's and BREAK OFF's must be executed to restore a break-able status.

CALL
Statement

The CALL statement provides external subroutine capabilities for a BASIC program. An external subroutine is a subroutine that is compiled (and possibly cataloged) separately from the program or programs that call it. An external subroutine can be called directly or indirectly.

The general form of the CALL statement is:

```
CALL {@}name {(argument list)}
```

The CALL statement with no @ is a direct call, and transfers control to the external subroutine named name. The name (item name of a program) may not have any special characters in it. The optional argument list consists of one or more expressions, 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. (See the next topic.) An argument list may continue on multiple lines; each line except the last must conclude with a comma.

The CALL @ form is used to specify an indirect call. When @name is present, name is a variable containing the name of the external subroutine to be called. The argument list performs the same function as in a direct call. For example:

```
NAME = 'XSUB1'
CALL @NAME
NAME = 'XSUB2'
CALL @NAME
```

The first call invokes subroutine XSUB1. The second call invokes subroutine XSUB2.

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 are the values of the arguments (plus any COMMON variables). External subroutines may call other external subroutines, including themselves. A sample external subroutine that involves two arguments, together with correctly formed CALL statements, is shown below.

```
CALL Statements          Subroutine ADD

CALL ADD (A,B,C)         SUBROUTINE ADD (X,Y,Z)
CALL ADD (A+2,F,X)       Z=X+Y
CALL ADD (3,495,Z)       RETURN
                         END
```

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 will be updated to reflect the most recent values of the corresponding variables in the subroutine. Constants and other expressions used as subroutine arguments will not be 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.

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

```
CALL {@}name {(argument list)}
```

Figure A. General Form of CALL Statement

CORRECT USE	EXPLANATION
CALL REVERSE (A,B)	Subroutine REVERSE has two arguments.
CALL REPORT	Subroutine REPORT has no arguments.
CALL VENDOR (NAME, ADDRESS, NUMBER)	Subroutine VENDOR returns three values.
CALL DISPLAY (A,B,C)	Subroutine DISPLAY accepts (and returns) three argument values.

Figure B. Examples of Correct Usage of CALL Statements

Arrays may be passed to external subroutines.

The general form for specifying an array in an argument list of CALL statements is:

```
CALL name (MAT variable)
```

The variable is the name of an array given in a DIM statement. 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. Consider the following example:

Calling Program	Subroutine
DIM X(4,5)	SUBROUTINE COPY (MAT A)
CALL COPY (MAT X)	DIM A(10,2)
END	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. Additional examples of array passing, both correct and incorrect, are shown in Figure B.

```
CALL name (MAT variable)
```

Figure A. General Forms of CALL Statement with Array Passing

CORRECT USE	EXPLANATION
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)	
INCORRECT USE	**EXPLANATION**
DIM TAB(100) CALL SHORT(TAB)	The word 'MAT' must precede array TAB in the parameter list.
DIM FOUR (2,2) CALL GOF (MAT FOUR)	Corresponding arrays must have the same number of elements in the calling program and the subroutine.
SUBROUTINE GOF(MAT NIX) DIM NIX(5)	

Figure B. Examples of Array Parameters

CASE
Statement

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

The general form of the CASE statement is:

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

If the logical value of the first expression is true (i.e., non-zero), then the statement or sequence of statements that immediately follows, up to the next CASE or END CASE, is executed, and control passes to the statement following END CASE. If the first expression is false (i.e., zero), then control passes to the next test expression, and so on.

Consider the following example:

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

If A<5, then the first PRINT statement will be executed. If 5<=A<10, then the second PRINT statement will be executed. Otherwise, the third PRINT statement will be executed. (Note that a test expression of 1 means "always true.")

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

Figure A. General form of CASE statement

CORRECT USE	EXPLANATION

BEGIN CASE
  CASE Y=B
  Y=Y+1
END CASE
	Increment Y if Y is equal to B. Note that this single-case example 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 if A is greater than zero.

BEGIN CASE
  CASE ST MATCHES "1A"
  MAT LET=1
  CASE ST MATCHES "1N"
  SGL=1; A.1(I)=ST
  CASE ST MATCHES "2N"
  DBL=1; A.2(J)=ST
  CASE ST MATCHES "3N"
  GOSUB 103
END CASE
	If ST is one letter, "1" is assigned to all LET elements and the entire CASE is ended. If ST is one number, "1" is assigned to SGL, ST is stored at element A.1(I), and the entire case is ended. If ST is two numbers, "1" is assigned to DBL, ST is stored at element A.2(J), and the entire case is ended. If ST is three numbers, subroutine 103 is executed.

Figure B. Examples of Correct Usage of CASE Statement

CHAIN
Statement

The CHAIN statement terminates program execution and executes a specified TCL command. The TCL command may be used to initiate another BASIC program using values from the first program.

The general form of the CHAIN statement is:

```
CHAIN "any TCL command"
```

The CHAIN statement causes the specified TCL command to be executed. The CHAIN statement may contain any valid verb or PROC name in the user's Master Dictionary. Consider the following example:

```
CHAIN "RUN FILE1 PROGRAM1 (I)"
```

This statement causes the previously compiled program named PROGRAM1 in the file named FILE1 to be executed. The I option specifies that the variables are not to be initialized. This causes them 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.

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. Consider, for example, the following two BASIC programs:

Program ABC in file BP

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

Program XYZ in file BP

```
PRINT X;PRINT Y;PRINT Z
END
```

Program ABC causes program XYZ to be executed. The I option used in the CHAIN statement specifies that the variable data area is not to be initialized, thus allowing program ABC to pass the values "500", "1", and "2" to program XYZ. Program XYZ, in turn, prints the values "500", "1", and "2" since they were allocated and passed in that order. The variable names do not need to correspond; only the order is significant to program XYZ.

Users should note that the workspace areas used for variable

storage are also used by other system software so their contents cannot be guaranteed when CHAINing from one BASIC program to another if there is any intermediate processing. In particular, CHAINing to a PROC which performs a Recall SELECT-type statement before invoking a BASIC program with the I option may cause the BASIC program's variables to be initialized to garbage.

Users should also note that control is never returned to the BASIC program originally executing the CHAIN statement. In order to accomplish this, an EXECUTE statement must be used instead of a CHAIN statement. (Please refer to the EXECUTE statement, listed alphabetically in this chapter.)

IMPORTANT: It is illegal to CHAIN from a subroutine, but legal to CHAIN a program that calls a subroutine.

```
CHAIN "any TCL command"
```

Figure A. General Form of CHAIN Statement

CORRECT USE	EXPLANATION
CHAIN "RUN FN1 LAX (I)"	Causes the execution of program LAX in file FN1. I option specifies that data area is not to be initialized (i.e., the program executing the CHAIN statement will pass values to program LAX).
CHAIN "LISTU"	Causes the execution of the LISTU PROC.
CHAIN "LIST FILE"	Causes the execution of the LIST Recall Verb.
CHAIN "RUN PROGRAMS ABC"	Causes the execution of program ABC in file PROGRAMS. Since I option is not used, values will not be passed to program ABC.

Figure B. Examples of Correct Usage of CHAIN Statement

CHAR
Function

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

The general form of the CHAR function is:

```
CHAR(expression)
```

The CHAR function converts the numeric value specified by the expression to its corresponding ASCII character string value. For example, the following statement assigns the string value for an Attribute Mark to the variable AM:

```
AM = CHAR(254)
```

CHAR always returns one character: if the value of expression is greater than 255, then CHAR(expression) = CHAR(expression MOD 256).

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

NOTE: For a complete list of ASCII codes, refer to Appendix C of this manual.

```
CHAR(expression)
```

Figure A. General Form of CHAR Function

CORRECT USE	EXPLANATION
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 Secondary Value Mark to variable SVM.

Figure B. Examples of Correct Usage of CHAR Function

CLEAR
Statement

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

The general form of the CLEAR statement is:

```
CLEAR
```

The CLEAR statement initializes all possible variables to zero (i.e., assigns the value 0 to all variables). It may appear anywhere in a program.

```
CLEAR
```

Figure A. General Form of CLEAR Statement

CORRECT USE	EXPLANATION
CLEAR	Assigns the value 0 to all possible variables.

Figure B. Correct Example of CLEAR Statement

CLEARFILE
Statement

The CLEARFILE statement is used to clear all data from a specified file.

The general form of the CLEARFILE statement is:

```
CLEARFILE {file-variable} {ON ERROR statements}
```

Upon execution of the CLEARFILE statement, the file which was previously assigned to the specified file-variable (via an OPEN statement) will be emptied (i.e., the data in the file will be deleted, but the file itself will not be deleted). If the file-variable is omitted from the CLEARFILE statement, then the internal default file-variable is used (thus specifying the file most recently opened without a file variable).

Consider the following example:

```
OPEN 'AFILE' TO X ELSE PRINT "CANNOT OPEN"; STOP
CLEARFILE X
```

These statements cause the data section of the file named AFILE to be cleared.

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 CLEARFILE statement. (Refer to Appendix B describing run-time error messages.)

The statement(s) after ON ERROR, if present, are executed only if the file (1) is a remote file (accessed via UltiNet) and (2) cannot be cleared due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 ON ERROR clause 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.

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 cleared due to network errors, the

program may terminate with an error message if no ON ERROR clause is present.

```
CLEARFILE {file-variable} {ON ERROR statements}
```

Figure A. General Form of CLEARFILE Statement

CORRECT USE	EXPLANATION
OPEN 'FN1' ELSE PRINT 'NO FN1';STOP READ I FROM 'I1' ELSE STOP CLEARFILE	Opens the data section of file FN1, reads item I1 and assigns value to variable I, and finally clears the data section of file FN1.
OPEN 'FILEA' TO A ELSE STOP OPEN 'FILEB' TO B ELSE STOP CLEARFILE A CLEARFILE B	Clears the data sections of files FILEA AND FILEB.
OPEN 'ABC' ELSE PRINT 'NO FILE'; STOP READV Q FROM 'IB3', 5 ELSE STOP IF Q='TEST' THEN CLEARFILE	Clears the data section of file ABC if the 5th attribute of the item with name IB3 has a string value of 'TEST'.

Figure B. Examples of Correct Usage of CLEARFILE Statement

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.

The general form of the CLOSE statement is:

```
CLOSE {file-variable} {ON ERROR statements}
```

The file-variable, if present, specifies the file variable to use in closing the file. If file-variable is omitted, the internal default file variable is assumed.

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 may abort to the 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. The file will, however, remain connected to any other file variables to which it is currently assigned in the program. In order to use the specified (closed) file variable again in I/O statements such as READ or WRITE, the variable must be re-connected to a file by means of another OPEN statement. In order to perform I/O on the file itself, the file must be OPENed to a different file variable, or re-OPENed to the same file variable.

The statement(s) after ON ERROR, if present, are executed only if the file (1) is a remote file (accessed via UltiNet) and (2) cannot be closed due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 ON ERROR clause 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.

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, the program may terminate with an error message if no ON ERROR clause is present.

Normally, 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.

```
CLOSE {file-variable} {ON ERROR statements}
```

Figure A. General Form of CLOSE Statement

CORRECT USE	EXPLANATION
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.

Figure B. Examples of Correct Usage of CLOSE Statement

COL1 and COL2
Functions

The COL1 and COL2 functions return the numeric values of the column positions immediately preceding and immediately following the sub-string selected by the most recent FIELD function.

The general form of the COL1 and COL2 functions are:

```
COL1()
COL2()
```

The COL functions are used in conjunction with the FIELD function. COL1() returns the numeric value of the column position immediately preceding the sub-string selected via the most recent FIELD function. For example:

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

These statements assign the numeric value 4 to the variable BEFORE (i.e., the value "YYY" which is returned by the FIELD function is preceded in the original string by column position 4).

COL2() returns the numeric value of the column position immediately following the sub-string selected via the most recent FIELD function. COL2() returns zero if the sub-string is not found. For example:

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

These statements assign the numeric value 8 to the variable AFTER (i.e., the value "YYY" which is returned by the FIELD function is followed in the original string by column position 8).

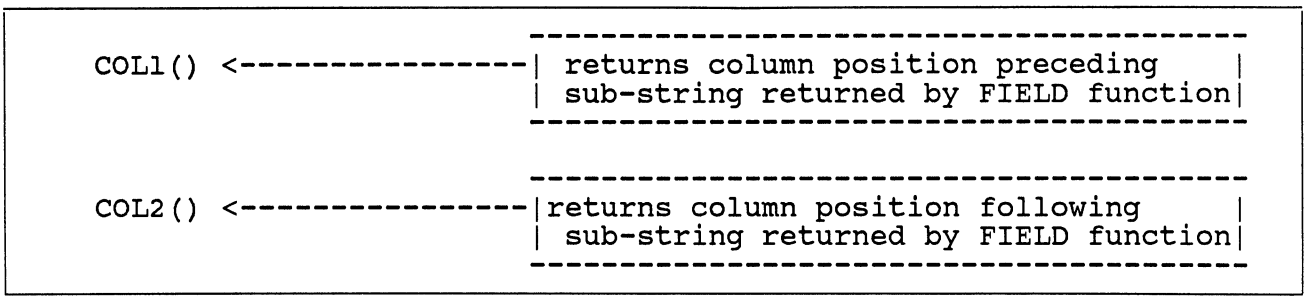


Figure A. General Form of COL1 and COL2 Functions

CORRECT USE	EXPLANATION
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.

Figure B. Examples of Correct Usage of COL1 and COL2 Functions

COMMON
Statement

The COMMON statement may be used to control the order in which space is allocated for the storage of variables, and for the passing of values between programs.

The general form of the COMMON statement is:

```
COM{MON} variable {,variable}...
```

The COMMON statement allows one or more variables specified by variable 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.

COMMON variables differ from subroutine arguments used with the CALL statement in that the actual storage locations of COMMON variables are shared by the main program and subroutines, 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 must be declared before any other variables, and in the same order, in all routines which access them. COMMON statements, then, should appear before any other statements which refer to variables.

Arrays 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: COMMON A(10), for example.

COMMON variables (including arrays) are allocated in the order in which they appear in COMMON statements. They 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. For example:

```
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. If SUBR had instead specified "COMMON Q(2), R(5)", then Q(1) would have corresponded to X, and Q(2) to Y.

```
COM{MON} variable {,variable} ...
```

Figure A. General Form of COMMON Statement

CORRECT USE	EXPLANATION
Item MAINPROG	
COMMON A,B,C(10) A = "NUMBER" B = "SQUARE ROOT" FOR I = 1 TO 10 C(I) = SQRT(I) NEXT I	Variables A, B, and array C are allocated space before any other variables.
CALL SUBPROG	Subroutine call to program SUBPROG.
PRINT "DONE" END	
Item SUBPROG	
COMMON X(2),Y(10) PRINT X(1), X(2) FOR J = 1 TO 10 PRINT J, Y(J) NEXT J	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.
RETURN	Returns to main-line program.
END	

Figure B. Example of Correct Usage of COMMON statement.

COS
Function

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

The general form of the COS function is:

```
COS(expression)
```

The expression must evaluate to a numeric expression that specifies the number of degrees in the angle.

Figure A shows a summary of all trigonometric functions. The value M represents the largest allowable number in BASIC, which is 14,073,748,835.5327, with PRECISION 4.

NOTE: Each trigonometric function is discussed separately. (Please refer to the function names, listed alphabetically in this chapter.)

FUNCTION	RANGE	DESCRIPTION
COS(X)	-M <= X <= M	Returns the cosine of an angle of X degrees.
SIN(X)	-M <= X <= M	Returns the sine of an angle of X degrees.
TAN(X)	-M <= X <= M	Returns the tangent of an angle of X degrees.
LN(X)	0 <= X <= M	Returns the natural (base e) logarithm of the expression X.
EXP(X)	-M <= RESULT <= M	Raises the number 'e' (2.7183) to the value of X.
PWR(X,Y)	-M <= RESULT <= M	Raises the first expression to the power denoted by the second expression.

Figure A. Summary of Trigonometric Functions

COUNT
Function

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

The general form of the COUNT function is:

```
COUNT(string,substring)
```

String and substring may be any valid expression, and may contain any number of characters.

The COUNT function returns a value of zero if the substring is not found, and returns the number of characters in the string if the substring is null. (That is, a null matches on any character.) For example:

```
   COMMAND                                  VALUE OF X

X = COUNT('THIS IS A TEST','IS')              2
X = COUNT('THIS IS A TEST','X')               0
X = COUNT('THIS IS A TEST','')               14
```

(There are 14 characters in the string.)

```
X = COUNT('AAAA','AA')                        3
```

There are 3 substrings within the string AAAA.

```
AAAA              STRING
XX                SUBSTRING 1
 XX               SUBSTRING 2
  XX              SUBSTRING 3
```

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.)

```
COUNT(string,substring)
```

Figure A. General Form of COUNT Function

CORRECT USE	EXPLANATION
A = "1234ABC5723" X = COUNT(A,'23')	Value returned in X is 2 since there are two occurrences of '23' in the string A.
X = COUNT('ABCDEFG','')	Value returned in X is 7 since a null substring will match any character.

Figure B. Examples of Correct Usage of COUNT Function

DATA
Statement

The DATA statement is used to store data for stacked input when using the CHAIN statement.

The general form of the DATA statement is:

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

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

Each expression in a DATA statement generates one line of stacked input. Normally, an input request such as from a BASIC INPUT statement prints a prompt character on the terminal and waits for the user to type in a line of data, ending with a carriage return. 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 pre-store input for commands or other BASIC programs invoked via the CHAIN statement. One BASIC program can set up parameters using DATA statements and then CHAIN to another program, which retrieves the parameters with INPUT statements. (For more information, see the INPUT and CHAIN statements, listed alphabetically in this chapter.)

Stacked input is removed (such as via the BASIC INPUT statement) in the same order that it is added via DATA statements. Stacked input may also be generated by the EXECUTE statement and by a PROC. (For more details, see the EXECUTE statement, listed alphabetically in this chapter.)

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

Figure A. General Form of the DATA Statement

CORRECT USE	EXPLANATION
DATA A DATA B DATA C CHAIN 'RUN BP TEST'	Stacks the values of A, B and C for subsequent input requests. Program 'TEST' may have three input requests which will be satisfied by the stacked input.

Figure B. Examples of Correct Usage of the DATA Statement.

DATE
Function

The DATE function returns the current internal date.

The general form of the DATE function is:

```
DATE()
```

The DATE function returns the string value containing the internal date. The internal date is the number of days since December 31, 1967. For example:

```
A = DATE()
```

This statement assigns the string value of the internal date to variable A.

```
DATE()
```

Figure A. General Form of DATE Function

CORRECT USE	EXPLANATION
`Q = DATE()`	Assigns string value of current internal date to variable Q.
`WRITET DATE() ELSE STOP`	Writes the string value of the current internal date onto a magnetic tape record.

Figure B. Examples of Correct Usage of DATE Function

DCOUNT
Function

The DCOUNT function counts the number of sub-strings which are separated by a specified delimiter in a string. It returns the number of sub-strings counted.

The general form of the DCOUNT function is:

```
DCOUNT(string,delimiter)
```

String and delimiter may be any valid expression. The string specifies the string to examine. Delimiter is the delimiter (string character) to use. The function returns the number of sub-strings within string that are separated by the delimiter. If string is null, a value of zero is returned.

Note that DCOUNT is similar to the COUNT function. (Please refer to the COUNT function, listed alphabetically in this chapter.) The DCOUNT function, however, differs from the COUNT function in that it returns a count of sub-strings separated by the specified delimiter, rather than the number of occurrences of the delimiter within the string. For example, consider the following string, where "^" represents an attribute mark, or AM:

```
A = ABC^DEF^GHI^JKL
```

Statement	Value of X
X = COUNT(A,AM)	3
X = DCOUNT(A,AM)	4

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.

```
DCOUNT(string,delimiter)
```

Figure A. General Form of DCOUNT Function

CORRECT USE	EXPLANATION
AM = CHAR(254) A = "123^456^ABC" X = DCOUNT(A,AM)	Value returned in X is 3 as there are three substrings in the string separated by Attribute Marks.
VM = CHAR(253) A = "123]456^ABC]DEF]HIJ" X = DCOUNT(A<1>,VM)	Value returned in X is 2 as there are two sub-strings in the string separated by Value Marks.
A = "" X = DCOUNT(A,AM)	Value returned in X is 0 since the string is null.

Figure B. Examples of Correct Usage of DCOUNT Function

DEL
Statement

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

The general form of the DEL statement is:

```
DEL variable <attribute# {, value# {,subval#}}>
```

The variable name identifies the dynamic array. The attribute#, value#, and subval# number(s) specify the position of the attribute, value, or subvalue to be deleted. The number(s) must be enclosed in angle brackets. For example, <3,5,1> denotes attribute 3, value 5, subvalue 1.

NOTE: This statement performs the same operation as the DELETE intrinsic function but also stores the function result back into the source variable. For example, DEL X<3> is equivalent to X=DELETE(X,3)

```
DEL variable <attribute# {, value# {, subvalue#}}>
```

Figure A. General Form of the DEL Statement

CORRECT USE	EXPLANATION
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.

Figure B. Examples of Correct Usage of DEL Statement

DELETE
Function

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

The general form of the DELETE function is:

```
DELETE(var,attr# {,value# {,subval#}})
```

The var is any expression that specifies the dynamic array to be used in the function. The other parameters may be any expressions that specify whether an attribute, a value, or a subvalue is deleted. Attr# specifies an attribute number, value# specifies a value, and subval# specifies a subvalue. If value# and subval# both have a value of 0 (or are both absent, then the attr# attribute is entirely deleted. If subval# only has a value of 0 (or is absent), then the value# value is deleted. If the attr#, value#, and subval# are all non-zero, then the subval# subvalue is deleted. In all cases, var remains unchanged.

If a value is deleted (i.e., subval# is zero or not expressed), the value mark associated with the value is also deleted. If an attribute is deleted (i.e., value# and subval# are are both zero or not expressed), the attribute mark associated with the attribute is also deleted.

Consider the following example:

```
OPEN 'INVENTORY' ELSE STOP
READ VALUE FROM 'ITEM2' ELSE STOP
VALUE = DELETE(VALUE,1,2,3)
WRITE VALUE ON 'ITEM2'
```

These statements delete subvalue 3 of value 2 of attribute 1 of item ITEM2 in file INVENTORY. The delimiter associated with subvalue 3 is also deleted.

Consider next the following example:

```
OPEN 'TEST' ELSE STOP
READ X FROM 'NAME' ELSE STOP
WRITE DELETE(X,2) ON 'NAME'
```

These statements delete attribute 2 (and its associated delimiter) of item NAME in file TEST.

NOTE: The DEL statement may be used to store the results of a DELETE operation on a variable back into the variable. For more information, see the DEL statement, listed alphabetically in this chapter.

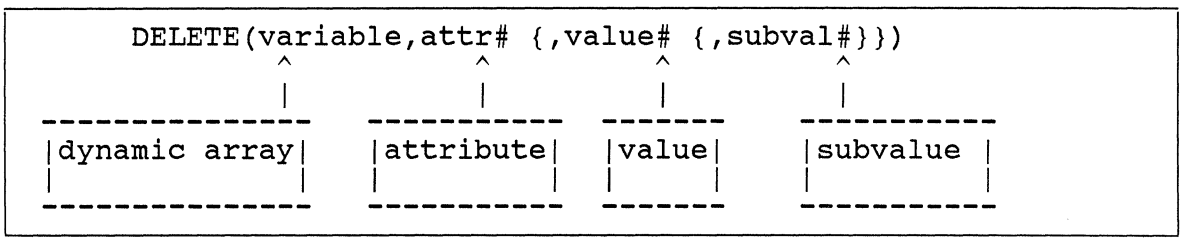


Figure A. General Form of DELETE Function

CORRECT USE	EXPLANATION
Y = DELETE(X,3,2)	Assigns to Y the dynamic array obtained by deleting value 2 (and its associated delimiter) of attribute 3 of dynamic array X.
A=1;B=2;C=3 DA = DELETE(DA,A,B,C-A)	Deletes subvalue 2 (and its associated delimiter) of value 2 of attribute 1 of dynamic array DA.
X = DELETE (X,7)	Deletes attribute 7 (and its associated delimiter) of dynamic array X.
PRINT DELETE(X,7,1)	Prints the dynamic array which results when value 1 of attribute 7 of dynamic array X is deleted.

Figure B. Examples of Correct Usage of DELETE Function

DELETE
Statement

The DELETE statement deletes a file item.

The general form of the DELETE statement is:

```
DELETE {file-variable,} item-id {ON ERROR statements}
```

The DELETE statement deletes the item which is specified by the expression item-id. If a file-variable is given, the item is assumed to be located in the file previously assigned to that specified file-variable via an OPEN statement. If the file-variable is omitted, then the internal default file variable is used; the default is the file most recently opened without a file-variable. For example:

```
DELETE AB,"TESTITEM"
```

This statement will delete the item named TESTITEM in the file previously opened and assigned to variable AB.

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

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 DELETE statement. (Please refer to Appendix B for run-time error messages.)

The statement(s) after ON ERROR, if present, are executed only if the file (1) is a remote file (accessed via UltiNet) and (2) cannot be accessed due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 accessing local files.

The ON ERROR clause 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.

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 accessed due to network errors, the program may terminate with an error message if no ON ERROR clause is present.

```
DELETE {file-variable,} item-id {ON ERROR statements}
```

Figure A. General Form of DELETE Statement

CORRECT USE	EXPLANATION
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.

Figure B. Examples of Correct Usage of DELETE Statement

DIM
Statement

A DIM statement declares the dimensions of an array with constant whole numbers, separated by commas.

The general form of the DIM statement is:

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

The variable specifies the array name. The dimensions specify the size of the array. If the array is a vector (one-dimensional), dimensions is the number of elements in the array; A(10), for example. If the array is a matrix (two-dimensional), dimensions gives the number of rows and the number of columns, separated by a comma, as in A(10,2).

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.

Before an array may be used in a BASIC program, the maximum dimension(s) of the array must be specified for storage purposes. The DIM (or COMMON) statement(s) must precede any references to the array(s), and are therefore usually placed at the beginning of the program. (Arrays need only be dimensioned once throughout the entire program.)

In the following example, the statement declares array A1 as a 10 by 5 matrix and declares array X as a 50 element vector:

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

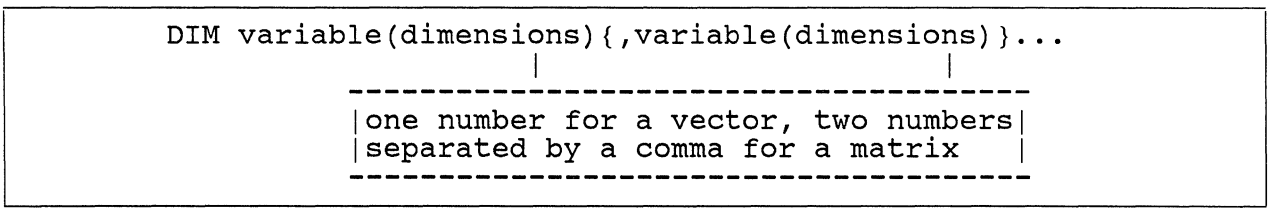


Figure A. General Form of DIM Statement

CORRECT USE	EXPLANATION
DIM MATRIX(10,12)	Specifies 10 by 12 matrix named MATRIX.
DIM Q(10),R(10), S(10)	Specifies three vectors named Q, R, and S (each to contain 10 elements).
DIM M1(50,10),X(2)	Specifies 50 by 10 matrix named M1, and two-element vector named X.

Figure B. Examples of Correct Usage of DIM Statement

DISPLAY
Statement

The DISPLAY statement outputs data to the terminal.

The general form of the DISPLAY statement is:

```
DISPLAY {print-list}
```

The print-list may consist of a single expression, or a series of expressions separated by commas, optionally ending with a colon. If the print-list is absent, only a carriage return and line feed will be displayed.

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:

1. Output is always to the terminal, regardless of PRINTER ON statements or the P option on the RUN verb.
2. Output cannot be redirected via OUT. in an EXECUTE statement.
3. Output is not affected by HEADING, FOOTING, or PAGE statements.

```
DISPLAY {print-list}
```

Figure A. General Form of DISPLAY Statement

```
CORRECT USE                    EXPLANATION

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

Figure B. Example of Correct Usage of DISPLAY Statement

EBCDIC
Function

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

The general form of the EBCDIC function is:

```
EBCDIC(expression)
```

The string value of the expression is converted from ASCII, the normal ULTIMATE string representation, to EBCDIC.

For example:

```
B = EBCDIC(A)
```

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

```
EBCDIC(expression)
```

Figure A. General Form of EBCDIC Function

CORRECT USE	EXPLANATION
B = EBCDIC(A)	Assigns the EBCDIC value of variable A to variable B.

Figure B. Example of Correct Usage of EBCDIC Function

ECHO (ON/OFF)
Statement

The ECHO ON and ECHO OFF statements control the system echo on the terminal.

The general forms of the ECHO statement are:

```
ECHO OFF

ECHO ON
```

The ECHO OFF statement disables the echo on the terminal. When the ECHO feature is disabled, characters typed on the keyboard are not displayed on the screen.

The ECHO ON statement enables the echo on the terminal. The ECHO feature will perform in its normal state, which is to display on the terminal screen the characters typed in on the keyboard.

END
Statement

The END statement may be used to designate the physical end of a program.

The general form of the END statement is:

```
END
```

The END statement may be used to indicate the end of a program. It is not required. Any statements appearing after an end-of-program END statement will be ignored.

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

A sample BASIC program illustrating the correct use of the END statement is presented in Figure B.

```
END
```

Figure A. General Form of END Statement

```
      *
      *
      *
      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,
 10   PRINT "POSITIVE PROFIT"        program will terminate
      END <--------------------------- Physical program end
```

Figure B. Sample Program with Correct Usage of END Statement

END CASE
Statement

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

The general form of the END CASE statement is:

```
END CASE
```

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

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

The two forms of the ENTER statement are:

```
ENTER program-name
```

where program-name is the item-id of the program to be ENTERed, and:

```
ENTER @variable
```

where variable has been assigned the program name to be ENTERed.

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 illegal to ENTER a program from a subroutine.

```
ENTER program-name
ENTER @variable
```

Figure A. General Forms of ENTER Statement

CORRECT USE	EXPLANATION
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".

Figure A. Examples of Correct Usage of ENTER Statement

EOF
Function

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

The general forms of the EOF function are:

```
EOF(ARG.)

EOF(MSG.)
```

The function must specify either the ARG. or MSG. redirection variable. When used with ARG., the function examines the program's argument list. When used with MSG., the function examines the internal system message buffer. The EOF function examines the specified redirection variable and returns a value of 1 if the end-of-file has been reached; otherwise, it returns a value of 0.

Checking the argument list

The ARG. redirection variable contains the arguments associated with a program. Arguments are specified following the program name in a statement which invokes a program. The arguments stored in ARG. are retrieved by the GET statement, which maintains a pointer into the argument list to determine which argument to return. If the last GET statement attempted to read past the end of the argument 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 a list of arguments.

Checking the system message buffer

The MSG. redirection variable contains error messages associated with programs executed from the most recent EXECUTE statement. The messages stored in MSG. are retrieved by the GET statement. The EOF function is used in conjunction with the GET statement. If the last GET statement attempted to read past the end of the internal system message buffer, an 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 list of errors.

Consider the following example:

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

This loop prints the messages stored in the MSG. redirection variable until the EOF function encounters the end-of-file condition. Then the loop exits.

```
EOF(MSG.)

EOF(ARG.)
```

Figure A. General Forms of EOF Function

CORRECT USE	EXPLANATION

LOOP
  GET(ARG.) X
UNTIL EOF(ARG.) DO
  PRINT X
REPEAT
	Prints arguments stored in the ARG. redirection variable until the EOF function (end-of-file) is true; then the loop exits.

Figure B. Example of Correct Usage of EOF Function

EQUATE
Statement

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

The general form of the EQUATE statement is:

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

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

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:

1. 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. In the example below, "AM" is the commonly used symbol for "attribute mark", one of the standard data delimiters.

2. 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)

EQUATE SALARY TO ITEM(3)
```

In this case, the variables BIRTHDATE, SOC.SEC.NO. and SALARY are rendered equivalent to the first three elements of the

array ITEM. These meaningful variables are then used in the remainder of the program.

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

Figure A. General Form of EQUATE Statement

CORRECT USE	EXPLANATION
EQUATE X TO Y	Symbol X and variable Y may be used interchangably within the program.
EQUATE PI TO 3.1416	Symbol PI is compiled as the value 3.1416.
EQUATE STARS TO "*****"	Symbol STARS is compiled as the value of five asterisks.
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.

Figure B. Examples of Correct Usage of EQUATE Statement

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.

The general form of the EXECUTE statement is:

EXECUTE expression {,//redirection phrase ...}

The expression is a string in the format of a TCL command, just as it would be typed in at the terminal; it names a verb, or cataloged BASIC program to be executed, followed by any parameters and options. After the verb/PROC/program being executed is completed, program control returns to the next statement following the EXECUTE statement.

An optional form of the EXECUTE statement uses "redirection phrases". A redirection phrase allows programs to redirect data to/from the verb/PROC/program being executed. Any number of redirection phrases may be included in one EXECUTE statement. The redirection phrases may continue on several lines; each line except the last must end with a comma.

Each redirection phrase has the following format:

redirection-variable direction expr

redirection-variable:	either IN., OUT., or SELECT.
direction:	either > or <.
expr :	a BASIC expression or variable to store output or use as input.

Selecting one of the three redirection variables

The names of these variables end with a period, as follows:

IN. (used only with direction symbol "<") specifies that expr data is to be re-directed to the input of the verb/PROC/program to be executed. "Input" usually means the user's CRT keyboard. This variable is equivalent to BASIC/PROC stacked input. If the verb, PROC, or program to be executed accepts more than one line of data, the redirected data must be delimited by attribute marks.

OUT. (used only with direction symbol ">") specifies that output from the verb/PROC/program being executed is to be redirected to expr. "Output" usually means the user's CRT screen or the spooler print file. If the verb, PROC, or program being executed produces more than one line of data, the redirected data is delimited by attribute marks. The last (or only) line of data is always terminated by an attribute mark.

SELECT. (used with either "<" or ">") specifies that a select list is to be redirected. Data on the list, typically item-ids, must be delimited by attribute marks; there is an attribute mark after the last datum. (See also Notes 2 and 3 below.)

Direction Symbols

< redirects to the input of the verb/PROC/program.

> redirects from the output of verb/PROC/program.

NOTES:

1. IN., OUT., and SELECT. are pre-defined variables with special meaning in the EXECUTE statement. They should not be used as ordinary variables in other statements. 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 pre-defined by the ULTIMATE system. Since variable names in this format may or may not be treated as names of pre-defined 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.

2. The select-list produced by an EXECUTE statement (e.g., EXECUTE "SELECT..") cannot be carried over automatically to the next EXECUTE statement. It can be redirected or used in a READNEXT statement in the same program. Thus:

```
EXECUTE "SELECT MD 'ED'"
EXECUTE "LIST MD"
```

will list all items in MD. But:

```
EXECUTE "SELECT MD 'ED'", //SELECT. > X
EXECUTE "LIST MD", //SELECT. < X
```

will list the selected items. And:

```
    EXECUTE "SELECT MD 'ED'"
10  READNEXT ID ELSE STOP
    PRINT ID
    GOTO 10
```

will print just one item.

3. The select list produced by an EXECUTE statement is a list of data (typically item-ids), each of which terminates in an attribute mark, including the last (or only) datum. If a program counts the number of item-ids with the DCOUNT(list,CHAR(254)) function, the number returned is one higher than the actual number of item-ids in the list.

4. When the select list is stored in a variable (...//SELECT. > X), the list X may be used directly in a READNEXT statement. It is not a dynamic array, and no SELECT statement should be used prior to the READNEXT.

5. The list X (see 4, above) 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.

6. But, if a dynamic array element is changed in the list X, the list itself is converted into a dynamic array. To then use READNEXT, the program must SELECT the dynamic array to a list. Assuming select list X:

```
X<1> = "ABC"
READNEXT ID FROM X   will fail

X<1> = "ABC"
SELECT X TO X
READNEXT ID FROM X   will work
```

```
EXECUTE expression {,//redirection phrase ...}
```

Figure A. General Form of EXECUTE Statement

CORRECT USE	EXPLANATION
`EXECUTE "WHO"`	The command WHO is executed; the output (e.g., "0 SYSPROG") is displayed on user's CRT 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 (e.g., "0 SYSPROG":AM) 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",` `  //IN. < "BACKUPPROG1"`	The command COPY is executed using, instead of a user's keyboard input, the redirected string "BACKUPPROG1".
`EXECUTE "RUN BP TWOINPUT",` `  //IN. < "ONE":CHAR(254):"TWO"`	Assume that the BASIC program TWOINPUT has two INPUT statements. The first INPUT statement will receive the data "ONE"; the second, the data "TWO".

`EXECUTE "ED BP X",`
`  //IN. < "L22":CHAR(254):"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"', //SELECT. > X`
`EXECUTE 'LIST EMP.ADDR.FILE',`
`  //SELECT. < X`
	In the first EXECUTE statement, the select list is redirected to variable X, with the item-ids separated by attribute marks; this makes X a variable array comprised of item-ids from SELECT. The select list is then redirected into the LIST command in the second EXECUTE statement.

Figure B. Examples of Correct Usage of EXECUTE Statement

EXIT
Statement

The EXIT statement transfers control out of a program loop initiated by a LOOP statement.

The general form of the EXIT statement is:

```
EXIT
```

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 nearest REPEAT. The EXIT statement must be used within a LOOP...REPEAT program loop; otherwise, the BASIC compiler will flag an error.

```
EXIT
```

Figure A. General Form of EXIT Statement

CORRECT USE	EXPLANATION

LOOP ※
  READNEXT ID ELSE EXIT
  GOSUB PROCESSIT
REPEAT
PRINT "DONE"
	Subroutine PROCESSIT is called after each value from a pre-selected list is read by READNEXT. When the list is exhausted, the program loop is exited, causing the message "DONE" to be printed.

Figure B. Example of Correct Usage of EXIT Statement

EXP
Function

The EXP trigonometric function returns the value of the number 'e' raised to a specified power.

The general form of the EXP function is:

EXP(expression)

The EXP (exponential) function raises the number 'e' (2.7183) to the value of the expression. The EXP function is the inverse of the LN (natural logarithm) function. If the value of the expression is such that 'e' to that power is greater than the largest allowable number, the function returns a value of zero.

In the following summary M is used to denote the largest allowable number in BASIC, which is 14,073,748,835.5327 with PRECISION 4.

FUNCTION	RANGE	DESCRIPTION
COS(X)	-M <= X <= M	Returns the cosine of an angle of X degrees.
SIN(X)	-M <= X <= M	Returns the sine of an angle of X degrees.
TAN(X)	-M <= X <= M	Returns the tangent of an angle of X degrees.
LN(X)	0 <= X <= M	Returns the natural (base e) logarithm of the expression X.
EXP(X)	-M <= RESULT <= M	Raises the number 'e' (2.7183) to the value of X.
PWR(X,Y)	-M <= RESULT <= M	Raises the first expression to the power denoted by the second expression.

Figure A. Summary of Trigonometric Functions

EXTRACT
Function

The EXTRACT function returns an attribute, a value, or a subvalue from a dynamic array.

The general form of the EXTRACT function is:

```
EXTRACT(expr,attr# {,value#} {,subvalue#})
```

The value of expr specifies the dynamic array to extract data from. The values of attr#, value#, and subvalue# determine whether the data is an attribute, a value, or a subvalue. Attr# specifies an attribute, value# specifies a value, and subvalue# specifies a subvalue. If value# and subvalue# both have a value of 0 (or are both absent), then an entire attribute is extracted. If subvalue# only has a value of 0 (or is absent), then a value is extracted. If attr#, value#, and subvalue# are all non-zero, then a subvalue is extracted.

Consider the following example:

```
OPEN 'TEST' ELSE STOP
READ X FROM 'NAME' ELSE STOP
PRINT EXTRACT(X,3,2)
```

These statements cause value 2 of attribute 3 of item NAME in file TEST to be printed.

Consider next the following example:

```
OPEN 'ACCOUNT' ELSE STOP
READ ITEM1 FROM 'ITEM1' ELSE STOP
IF EXTRACT(ITEM,3,2,1)=25 THEN PRINT "MATCH"
```

These statements cause the message "MATCH" to be printed if subvalue 1 of value 2 of attribute 3 of item ITEM1 in file ACCOUNT is equal to 25.

NOTE: The EXTRACT intrinsic 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>.

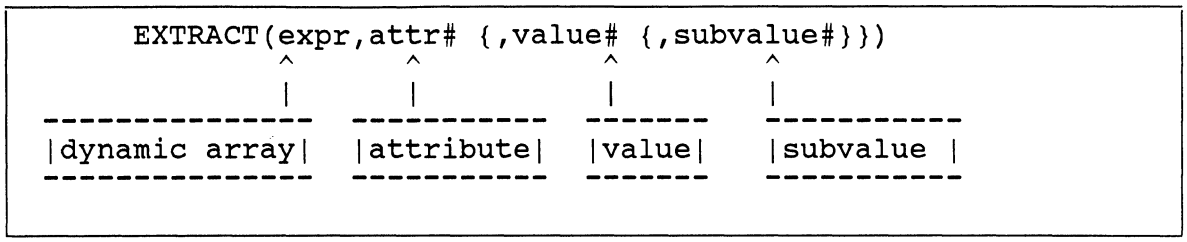


Figure A. General Form of EXTRACT Function

CORRECT USE	EXPLANATION
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,3,2,1)>5 THEN PRINT MSG GOSUB 100 END	If subvalue 1 of value 2 of attribute 3 of dynamic array B is greater than 5, then the value of MSG is printed and a subroutine call is made to statement 100.
PRINT EXTRACT(D,25,2)	Prints value 2 of attribute 25 of dynamic array D.

Figure B. Examples of Correct Usage of EXTRACT Function

FADD
Function

The FADD (floating point addition) function adds two floating point numbers and returns the result as a floating point number.

The general form of the FADD function is:

```
FADD (FX, FY)
```

FX and FY may be any valid floating point numbers.

A standard or string number must be converted to floating point before performing the FADD function. The FFLT function is provided to float a number or string. The FFIX function is provided to fix a floating point number, returning a string number. (Please refer to the FFLT and FFIX functions, listed alphabetically in this chapter.)

If either FX or FY contains an invalid (non-floating point) value, an error message is generated; the result of the addition will be "0E0".

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 or a string would be valid.

```
FADD (FX,FY)
```

Figure A. General Form of FADD Function

CORRECT USE	EXPLANATION
TOTAL=FADD(SUBTOT1,SUBTOT2)	Assigns sum of variables SUBTOT1 and SUBTOT2 to variable TOTAL.
PRINT (FADD(X,"4E-3")	Prints sum of variable X and floating point constant (.004).
A=FADD("1030476E-6",B)	Assigns to variable A the sum of floating point constant (1.030476) and 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.

Figure B. Examples of Correct Usage of FADD Function

FCMP
Function

The FCMP (floating point compare) function compares two floating point numbers and returns a result of -1 (less than), 0 (equal), or 1 (greater than).

The general form of the FCMP function is:

```
FCMP (FX, FY)
```

FX and FY may be any valid floating point numbers.

A standard or string number must be converted to floating point before performing the FCMP function. A FFLT function is provided to float a number or string. A FFIX function is provided to fix a floating point number, returning a string number. (Please refer to the FFLT and FFIX functions, listed alphabetically in this chapter.)

If either FX or FY contains an invalid (non-floating point) value, an error message is generated; the result of the comparison will be zero (0).

The result of the FCMP function is a number: -1, 0, or 1. If FX is less than FY, the result is -1. If they 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.

```
FCMP (FX,FY)
```

Figure A. General Form of FCMP Function

CORRECT USE	EXPLANATION
`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.
`IF FCMP(FX,FY) < 0 THEN PRINT X:" IS LESS THAN ":Y`	The PRINT operation is executed only if the result of the IF statement is true (-1 was the result of the FCMP function).
`IF FCMP(FX,FY) > 0 THEN PRINT X:"IS GREATER THAN ":Y`	The PRINT operation is executed only if the result of the IF statement is true (1 was the result of the FCMP function).
`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.

Figure B. Examples of Correct Usage of FCMP Function

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.

The general form of the FDIV function is:

```
FDIV (FX, FY)
```

FX and FY may be any valid floating point numbers.

A standard or string number must be converted to floating point before performing the FDIV function. A FFLT function is provided to float a number or string. A FFIX function is provided to fix a floating point number, returning a string number. (Please refer to the FFLT and FFIX functions, listed alphabetically in this chapter.)

If either FX or FY contains an invalid (non-floating point) value, an error message is generated; the result of the addition will be "0E0". If FY is zero, an error message will state that division by zero is illegal; the result will be "0E0".

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 or a string would be valid.

```
FDIV (FX,FY)
```

Figure A. General Form of FDIV Function

CORRECT USE	EXPLANATION
VELOCITY=FDIV(DISTANCE,TIME)	Assigns result of variables DISTANCE divided by TIME to variable VELOCITY.
PRINT (FDIV(X,"4E-3")	Prints quotient of variable X divided by floating point constant (.004).
A=FDIV("1030476E-6",B)	Assigns to variable A the result of dividing floating point constant (1.030476) 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 sum to variable X.

Figure B. Examples of Correct Usage of FDIV Function

FFIX
Function

The FFIX (fix a floating point number) function returns the value of a floating point number as a string number.

The general form of the FFIX function is:

```
FFIX (FX {,N})
```

FX may be any valid floating point number. The optional N operand may be any valid integer number. When present, N sets the maximum number of digits to the right of the decimal point in the result. If N is omitted or is negative, the result will contain all possible digits to the right of the decimal point. Whenever the result has fewer digits to the right of the decimal than the maximum allowed, the unused digits are truncated. The result is not rounded.

NOTE: This function is normally 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, 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.

```
FFIX(FX {N})
```

Figure A. General Form of FFIX Function

CORRECT USE	EXPLANATION
PRINT FFIX(FADD(FX,FY))	The result of the floating point addition is fixed as 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.
LINE=FFIX(FX,0):" ":FFIX(FY,0)	The variable LINE is assigned the value of FX converted into a string integer, concatenated with 3 spaces, concatenated with the value of FY converted into a string integer.

Figure B. Examples of Correct Usage of FFIX Function

FFLT
Function

The FFLT (float a number or string number) function converts a number or string number into a floating point number.

The general form of the FFLT function is:

```
FFLT(X)
```

X may be any valid number or string number. If X is not valid, an error message is generated and the result will be "0E0". If the number contains more than 13 significant digits, it will be truncated to 13 significant digits.

This function must precede floating point arithmetic performed on a standard number or string number.

NOTE: The FFLT function is normally used with the 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.

```
FFLT(X)
```

Figure A. General Form of FFLT Function

CORRECT USE	EXPLANATION
X=FFLT(Y)	The floating point value of Y is assigned to X.
A=FMUL(FFLT(X),FFLT(Y))	The variables X and Y are floated and then used in a floating point multiplication function; the result is assigned to variable A.
FLOAT.PI=FFLT("3.1415926")	The constant pi is floated and assigned to FLOAT.PI.

Figure B. Examples of Correct Usage of FFLT Function

FIELD
Function

The FIELD function returns a sub-string from a string by specifying a delimiter character.

The general form of the FIELD function is:

```
FIELD(string,delimiter,occurrence)
```

All three FIELD parameters may be any valid expression. The FIELD function searches string for a sub-string delimited by the delimiter character. Occurrence specifies which occurrence of the sub-string is to be returned. If occurrence has a value of 1, then the FIELD function will return the sub-string from the beginning of the string up to the first occurrence of the delimiter. For example, the statement below assigns the string "XXX" to the variable A:

```
A = FIELD("XXX.YYY.ZZZ.555",".",1)
```

If occurrence has a value of 2, then the sub-string delimited by the first and second occurrences of the specified delimiter character delimiter will be returned. A value of 3 for occurrence will return the sub-string delimited by the second and third occurrences of delimiter, and so on. For example, the statement below assigns the string "ZZZ" to variable C:

```
C = FIELD("XXX.YYY.ZZZ.555",".",3)
```

Note that the end of the string also delimits sub-strings, so that:

```
FIELD("XXX.YYY.ZZZ.555",".",4)
```

returns a value of "555". If the specified substring is not found, the function returns a null string ('').

NOTE: The COL1() and COL2() functions are used in conjunction with the FIELD function. (Please refer to the COL functions, listed alphabetically in this chapter.)

```
FIELD(string,delimiter,occurrence)
```

Figure A. General Form of FIELD Function

CORRECT USE	EXPLANATION
T = "12345A6789A98765A" G = FIELD(T,"A",1)	Assigns the string value "12345" to variable G.
T = "12345A6789A98765A" G = FIELD(T,"A",3)	Assigns the string value "98765" 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.
X = "77ABCXX" Y = "$" Z = "ABC" IF FIELD(X,Y,2)=Z THEN STOP	The IF statement will cause the program to terminate (i.e., the value returned by the FIELD function is "ABC", which equals the value of Z, thus making the test condition true).

Figure B. Examples of Correct Usage of FIELD Function

FMUL
Function

The FMUL (floating point multiplication) function multiplies two floating point numbers and returns the result as a floating point number.

The general form of the FMUL function is:

```
FMUL (FX, FY)
```

FX and FY may be any valid floating point numbers.

A standard or string number must be converted to floating point before performing the FMUL function. A FFLT function is provided to float a number or string. A FFIX function is provided to fix a floating point number, returning a string number. (Please refer to the FFLT and FFIX functions, listed alphabetically in this chapter.)

If either FX or FY contains an invalid (non-floating point) value, an error message is generated; the result of the multiplication will be "0E0".

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.

```
FMUL (FX,FY)
```

Figure A. General Form of FMUL Function

CORRECT USE	EXPLANATION
PAY=FMUL(HOURS,RATE)	The variable PAY is assigned the product of HOURS times RATE.
PRINT FMUL(X,"10015E-4")	The variable X is multiplied by constant 1.0015 and the result is printed.
A=FMUL("1030476E-6",B)	The constant 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.

Figure B. Examples of Correct Usage of FMUL Function

FOOTING
Statement

The FOOTING statement causes the specified text string to be printed at the bottom of each page of output.

The FOOTING statement has the following general form:

```
FOOTING expression
```

The first FOOTING or HEADING statement executed in a program will initialize the page parameters. Subsequently, the footing literal data may be changed at any time in the BASIC program by another FOOTING statement; this change will take effect when the end of the current page is reached. The special footing option characters listed in Figure B may be used as part of a FOOTING string expression. When used, these special characters will be converted and printed as part of the footing. Option characters are enclosed in single quotes.

Consider, for example:

```
FOOTING "STATISTICS AS OF 'T' PAGE 'PL'"
```

This statement will print at the bottom of each page a footing consisting of the words "STATISTICS AS OF", followed by the current time and date, followed by the word "PAGE", followed by the current page number, followed by a carriage return and line feed. Page numbers are assigned in ascending order starting with page 1.

NOTE: The FOOTING statement affects only print file zero, the default output device.

```
FOOTING expression
```

Figure A. General Form of Footing Statements

Character Used in FOOTING String Expression	Character is Converted to:
PN	Current page #, left-justified
P	Current page #, right-justified in a field of 4 blanks
L	Carriage return/line feed
T	Current time and date
C	Centers the line
Cn	Centers with specified line length
D	Current date

Figure B. Special Control Characters for FOOTING Statement

CORRECT USE	EXPLANATION
FOOTING "TIME & DATE: 'TL'"	The text "TIME & DATE:" will be printed, followed by the current time and date plus a carriage return/line feed.
FOOTING "'C60'PAGE 'PL'"	The text "PAGE" will be centered, within a page width of 60, followed by the current page number and a carriage return-line feed.
FOOTING "'LTPL'"	The following footing will be printed: the current time, date, and page number.

Figure C. Examples of Correct Usage of FOOTING Statements

FOR
Statement

The FOR statement is used to specify the beginning point of a program loop. A loop is a portion of a program written in such a way that it will execute repeatedly until some test condition is met. The FOR statement is always used with a NEXT statement that specifies the ending point of the loop.

The general forms of the FOR statement are:

```
FOR variable = expr1 TO expr2 {STEP expr3} {WHILE expr4}
FOR variable = expr1 TO expr2 {STEP expr3} {UNTIL expr5}
```

A FOR and NEXT loop causes execution of a set of statements for successive values of the specified variable until the limit is reached. The values of the expressions are used as follows: expr1 is the initial value for variable; expr2 is the limit value; the optional expr3 is the 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. When the limit value (expr2) is exceeded, program control proceeds to the statement after the NEXT statement.

Expr1 is evaluated only once (when the FOR statement is executed). Expr2 and expr3 are evaluated on each iteration of the loop.

One of the optional condition clauses (WHILE and UNTIL) may be used in a FOR statement. If the WHILE clause is used, expr4 will be evaluated for each iteration of the loop. If it evaluates to false (i.e., zero), then program control will pass to the statement immediately following the accompanying NEXT statement. If it evaluates to true (i.e., non-zero) the loop will re-iterate.

If the UNTIL clause is used, expr5 will be evaluated for each iteration of the loop. If it evaluates to true (i.e., non-zero), then program control will pass to the statement immediately following the accompanying NEXT statement. If it evaluates to false (i.e., zero) the loop will re-iterate.

The following FOR and NEXT loop, for example, will execute until I=10 or until the statements within the loop cause variable A to exceed the value 100:

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

The program loop concludes with a NEXT statement. The function of the NEXT statement is to return program control to the beginning of the loop after a new value of the

variable has been computed. The NEXT statement is discussed in a separate topic. (For details, please refer to the NEXT statement, alphabetically listed in this chapter.)

The following example shows a complete FOR/NEXT loop:

```
150  FOR J=2 TO 11 STEP 3
160  PRINT J+5
170  NEXT J
```

Statement 150 sets the initial value of J to 2 and specifies that J thereafter will be incremented by 3 each time the loop is performed, until J exceeds the limiting value, 11. Statement 160 prints the current value of the expression J+5. Statement 170 assigns J its next value (i.e., J=2+3=5) and causes program control to return to statement 150. Statement 160 is again executed, and statement 170 again increments J and causes the program to loop back. This process continues with J being incremented by 3 after each pass through the loop. When J attains the limiting value of 11, statement 160 will again be executed and control will pass to 170. J will again be incremented (i.e., J=11+3=14), and since 14 is greater than the limiting value of 11, the program will "fall through" statement 170 and control will pass to the next sequential statement following statement 170.

FOR and NEXT loops may be "nested"; a nested loop is a loop which is wholly contained within another loop. For example:

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

The above statements illustrate a two-level nested loop. The inner loop will be executed ten times for each of ten passes through the outer loop, i.e., the statement PRINT B(I,J) will be 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 any number of levels. However, a nested loop must be completely contained within the range of the outer loop (i.e., the ranges of the loops may not cross).

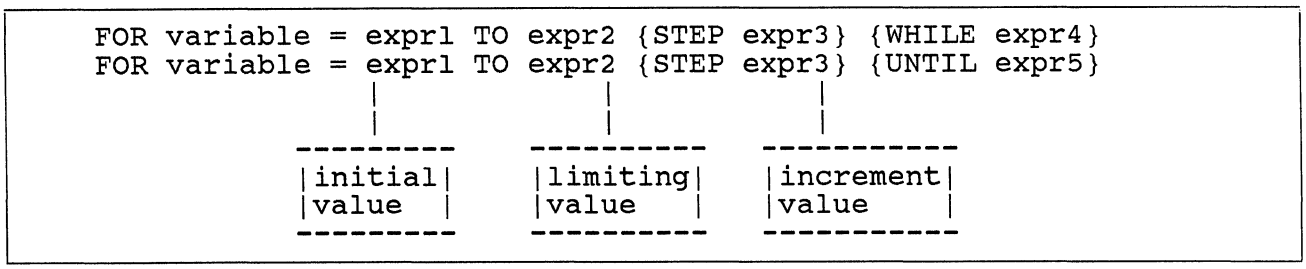


Figure A. General Forms of FOR Statement

CORRECT USE	EXPLANATION
FOR A=1 TO 2+X-Y . . NEXT A	Limiting value is current value of expression 2+X-Y; increment value is +1.
FOR K=10 TO 1 STEP -1 . . NEXT K	Increment value is -1 (i.e., variable K will decrement by -1 for each of 10 passes through the loop).
FOR VAR= 0 TO 1 STEP .1 . . NEXT VAR	Increment value is .1 (i.e., variable K will decrement by .1 for each of 11 passes through the loop).

Figure B. Examples of Correct Usage of FOR/NEXT Statements

CORRECT USE	EXPLANATION
ST="X" FOR B=1 TO 10 UNTIL ST="XXXXX" ST=ST CAT "X" NEXT B	Loop will execute 4 times (i.e., an "X" is added to the string value of variable ST until the string equals "XXXXX").
A=20 FOR J=1 TO 10 WHILE A<25 A=A+1 PRINT J,A NEXT J	Loop will execute 5 times (i.e., variable A reaches 25 before variable J reaches 10).
A=0 FOR J=1 TO 10 WHILE A<25 A=A+1 PRINT J,A NEXT J	Loop will execute 10 times (i.e., variable J reaches 10 before variable A reaches 25).

Figure C. Examples of UNTIL and WHILE Clauses in FOR/NEXT Statements

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.

The general form of the FSUB function is:

```
FSUB (FX, FY)
```

FX and FY may be any valid floating point numbers.

A standard or string number must be converted to floating point before performing the FSUB function. The FFLT function is provided to float a number or string. The FFIX function is provided to fix a floating point number, returning a string number. (Please refer to the FFLT and FFIX functions, listed alphabetically in this chapter.)

If either FX or FY contains an invalid (non-floating point) value, an error message is generated; the result of the subtraction will be "0E0".

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 or a string would be valid.

```
FSUB (FX,FY)
```

Figure A. General Form of FSUB Function

CORRECT USE	EXPLANATION
TOTAL=FSUB(SUBTOT1,SUBTOT2)	Assigns difference of variables SUBTOT1 and SUBTOT2 to variable TOTAL.
PRINT (FSUB(X,"4E-3")	Prints difference of variable X and floating point constant (.004).
A=FSUB("1030476E-6",B)	Assigns to variable A the difference of floating point constant 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.

Figure B. Examples of Correct Usage of FSUB Function

GET
Statement

The GET statement allows a program to retrieve data stored in either the ARG. or MSG. redirection variable. When used with ARG., it retrieves an argument from the program's argument list. When used with MSG., it retrieves a message resulting from the last EXECUTE statement.

The general forms of the GET statement are:

```
GET ( ARG. {, arg#} ) var {THEN stmt} {ELSE stmt}
GET ( MSG. {, arg#} ) var {THEN stmt} {ELSE stmt}
```

ARG. refers to the list of arguments (if any) following the program name in the TCL command which invoked the program. For example,

```
RUN BP MYPROG ARG1 ARG2
```

invokes program MYPROG, which can retrieve the strings "ARG1" and "ARG2" using the GET (ARG.) statement.

MSG., as used in the GET statement, refers to the list of message identifiers and parameters (if any) resulting from the last EXECUTE statement's program execution. A message identifier is the item-id of an item in the system ERRMSG file. Each message element in MSG. contains a message identifier and any parameter values for the message generated. One or more GET (MSG.) statements, then, can be used to retrieve the system messages generated by a program invoked via an EXECUTE statement. These messages are normally formatted according to the ERRMSG items and printed on the terminal or printer, but only the essential data (item-ids and parameters) are copied to MSG.. MSG. is reset to null just prior to the execution of an EXECUTE statement.

Retrieving Arguments

When using the GET statement syntax to retrieve arguments:

```
GET ( ARG. {, arg#} ) var {THEN stmt} {ELSE stmt}
```

arg# is an integer that specifies the argument to be retrieved; if arg# is not present, the next argument on the argument list is returned. (If this is the first GET statement executed, the first argument on the list is returned.) If an argument is present in the position specified, it is returned to the BASIC var and the THEN branch, if specified, is taken. If no argument is present in that position, var is not changed and the ELSE branch, if specified, is taken.

Retrieving Messages

When using the GET statement syntax to retrieve messages:

```
GET ( MSG. {, arg#} ) var {THEN stmt} {ELSE stmt}
```

arg# is an integer that specifies the message to be retrieved; if arg# is not present, the next message in the list is returned. (If this is the first GET statement executed, the first message is returned.) If a message is present in the position specified, it is returned to the BASIC variable var in the following format:

```
msg-id VM parm1 VM parm2 ...
```

where VM is a value mark, msg-id is the message identifier (ERRMSG item-id) and the parm#'s are the parameters associated with the system error. The THEN branch is taken if a message was found; if no message is in that position, the ELSE branch is taken.

NOTE: The EOF function is available to test for end-of-arguments or end-of-messages. (Please refer to the EOF function, listed alphabetically in this chapter.)

System messages are typically generated by commands or BASIC programs just before terminating execution. BASIC programs may generate messages using the PUT, STOP, and ABORT statements. For further information, please refer to these statements, listed alphabetically in this chapter.

ARG. and MSG. are pre-defined variables with special meaning in the GET statement and should not be used as ordinary variables in other statements. It is recommended that programmers not use names ending in a period in order to distinguish them from the variables pre-defined by ULTIMATE. Since variable names in this format may or may not be treated as names of pre-defined variables in all cases, depending on operating system release, the ULTIMATE Corp. strongly suggests programmers rewrite their software, if necessary, to avoid possible conflicts.

```
GET ( ARG. {, arg#} ) var {THEN stmt} {ELSE stmt}

GET ( MSG. {, arg#} ) var {THEN stmt} {ELSE stmt}
```

Figure A. General Forms of the GET Statement

CORRECT USE	EXPLANATION
GET(ARG.) PARAM1 ELSE PARAM1 = 0 GET(ARG.) PARAM2 ELSE PARAM2 = 0	The first GET statement retrieves the first argument on the list; the second retrieves the next argument.
GET(ARG.,2) PARAM2 ELSE PRINT "NEED MORE PARAMETERS" STOP GET (ARG.) PARAM3	The first GET statement retrieves the second argument on the list or prints a message and the program terminates. The second GET statement retrieves the next (third) argument.
GET(MSG.,1) ERR1 ELSE GOTO START PRINT ERR1; STOP	The first GET statement retrieves the first message; if found, it is printed and the program terminates. If no message has been stored in MSG., the program branches to START.

Figure B. Examples of Correct Usage of the GET Statement

GOSUB
Statement

The GOSUB statement transfers control to an internal subroutine. An internal subroutine is a subroutine that is contained within the program that calls it. The GOSUB statement is always used with a RETURN or RETURN TO statement that transfers control back to the main routine.

The general form of the GOSUB statement is:

```
GOSUB statement-label
```

The GOSUB statement transfers control to the subroutine that starts at the specified statement-label. Execution proceeds sequentially from that statement until a RETURN or RETURN TO statement is encountered. Either of these statements can be used to transfer control back to the main (branched-from) routine.

The ON GOSUB statement is a "computed" GOSUB statement, a combination of the ON GOTO statement and the GOSUB statement. An expression is used to compute which subroutine to execute next. The ON GOSUB statement is discussed in a separate topic. (Please refer to the ON GOSUB statement, listed alphabetically in this chapter.)

```
GOSUB statement-label
```

Figure A. General Form of GOSUB Statement

```
GOSUB FIRST             Transfers control to subroutine
...                     FIRST. After the subroutine's RETURN
FIRST:                  statement is encountered, control
...                     returns to the statement after the
...                     GOSUB statement.
RETURN
```

Figure B. Example of Correct Usage of GOSUB Statement

GOTO (GO TO) Statement

The GO{TO} statement unconditionally transfers program control to any statement within the BASIC program.

The general form of the GO{TO} statement is:

GOTO statement-label or GO statement-label

Execution of the GO{TO} statement causes program control to be transferred to the statement which begins with the specified statement-label. If a statement does not exist with the specified statement-label an error message will be printed at compile time. (Refer to Appendix A describing compiler error messages.)

Figure B illustrates a correct use of the GO{TO} statement. Flow of program control is illustrated by the arrows. Note that control may be transferred to statements following the GO{TO} statement, as well as to statements preceding the GO{TO} statement. Figure C illustrates incorrect use of the GO{TO} statement.

```
GO{TO} statement-label
```

Figure A. General Form of GOTO Statement

```
-----> 100 A-0
|           .
|           .
|           REM BRANCH TO STATEMENT 500
|       200 GOTO 500 ---
|           .           |
|           .           |
|   --------------------
|   |       .
|   |--> 500 A-B+C
|           D-100
|           .
|           .
|           REM REPEAT PROGRAM
|           GOTO 100 ---
|----------------------|
```

Figure B. Example of Correct Usage of GOTO Statement

STATEMENTS	EXPLANATION
100 A-0 . . --> 200 GOTO 200 -- |----------|	GOTO statement branches to itself. Program will permanently "hang", re-executing this statement.

Figure C. Example of Incorrect Usage of GOTO Statement

HEADING
Statement

The HEADING statement causes the specified text string to be printed as the next page heading.

The general form of the HEADING statement is:

```
HEADING expression
```

The first HEADING or FOOTING statement executed in a program will initialize the page parameters. Subsequently, the heading literal data may be changed at any time in the BASIC program by another HEADING statement; this change will take effect at the beginning of the next page. The special HEADING option characters listed in Figure B may be used as part of a HEADING string expression. When used, these special characters will be converted and printed as part of the heading. Option characters are enclosed in single quotes.

Consider, for example:

```
HEADING "'C'INVENTORY LIST 'T' PAGE 'PL'"
```

This statement will print at the top of each page a centered heading (using the default page width). The heading will consist of the words "INVENTORY LIST", followed by the current time and date, followed by the word "PAGE", followed by the current page number, followed by a carriage return and line feed. Page numbers are assigned in ascending order starting with page 1.

NOTE: The HEADING statement affects only print file zero, the default output device.

```
HEADING expression
```

Figure A. General Form of HEADING Statements

Character Used in HEADING String Expression	Character is Converted to:
PN	Current page#, left-justified
P	Current page#, right-justified in a field of 4 blanks
L	Carriage return/line feed
T	Current time and date
C	Centers the line
Cn	Centers, with specified line length
D	Current date

Figure B. Special Control Characters for HEADING Statement

CORRECT USE	EXPLANATION
HEADING "TIME & DATE: 'TL'"	The text "TIME & DATE:" will be printed followed by the current time and date plus a carriage return/line feed.
HEADING "'C60'PAGE 'PL'"	The text "PAGE" will be centered, within a page width of 60, followed by the current page number and a carriage return/line feed.
HEADING "'LTPL'"	The following heading will be printed: the current time, date and page.

Figure C. Examples of Correct Usage of HEADING Statements

ICONV
Function

The ICONV function converts a string according to a specified type of input conversion.

The general form of the INCONV function is:

```
ICONV(string,code)
```

The string specifies a string value. The code specifies the type of input conversion. The resultant value is always a string value.

The value of code must be a string. The following codes may be used for input conversions:

- D — Convert date to internal format
- G — Extract group of characters
- L — Test string length
- MC — Mask characters by numeric, alpha, or upper/lower case
- 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 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 Recall Reference Manual for details.) NOTE: This type of conversion is inefficient if several items or attributes will be accessed.
- U — Convert by subroutine call to assembly routine, either system- or user-defined. 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, please refer to the Assembler Manual.)

The conversion codes are the same as those used for Recall Conversions and Correlatives. For a detailed treatment of these and other conversion capabilities, refer to the ULTIMATE Recall Reference Manual.

WARNING: Some conversion codes used in Recall, such as "F", cannot be used with the ICONV function. Also note that "MR" and "ML" conversions may be done with format strings. In general, values which cannot be successfully converted will cause a null string to be returned as a result. For example, a string which is not a valid date will cause ICONV to return a null string when used with the "D" conversion code. (For details on format strings, refer to the chapter "Representing Data" in this manual.)

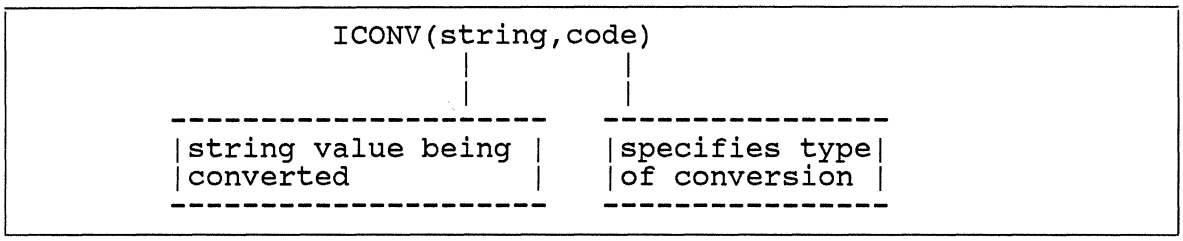


Figure A. General Form of ICONV Function

CORRECT USE	EXPLANATION
IDATE = ICONV("07-01-74","D")	Assigns the string value "2374" (i.e., 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,"MT")="" THEN GOTO 10	Causes a branch to label 10 if the value of T is not a valid time of day (if the "MT" conversion returns a null string).

Figure B. Examples of Correct Usage of ICONV Function

IF
Statement

The Single-Line IF statement provides the conditional execution of a sequence of BASIC statements, or the conditional execution of one of two sequences of statements.

The general form of the Single-Line IF statement is:

```
IF expression {THEN statements} {ELSE statements}
```

The expression may be any legal BASIC expression. If the result of the test condition specified by the expression is true (i.e., non-zero), then the statement or sequence of statements in the THEN clause (if present) are executed. If the result of the expression is false (i.e., zero), then the statement or sequence of statements in the ELSE clause (if present) are executed.

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present. The sequence of statements in the THEN or ELSE clauses may consist of one or more statements on the same line. For example:

```
IF X>1 THEN GOTO 50
```

In this example control will be transferred to statement 50 if the current value of X is greater than 1. Since the ELSE clause is not used here, control will pass to the next statement in the program if X is not greater than 1. An ELSE clause can be used instead of a THEN clause:

```
IF X ELSE GO 100
```

In this example, if X is false (zero), control is passed to the ELSE clause. If X is true (non-zero), control will simply pass to the next statement in the program. This statement then replaces the longer forms:

```
IF X THEN NULL ELSE GO 100
     or
IF NOT(X) THEN GO 100
```

If more than one statement is contained in either the THEN or ELSE clause, they must be separated by semicolons. Consider the example:

```
IF ITEM THEN PRINT X; X=X+1 ELSE PRINT X*5; GOTO 10
```

If the current value of ITEM is non-zero (i.e., true), then this statement will print the current value of X, add 1 to the current value of X, and then transfer control to the next sequential instruction in the program. If the value of ITEM is zero (i.e., false), then the value of X*5 will be printed and control will transfer to statement 10.

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

```
IF expression {THEN statements} {ELSE statements}
```

Figure A. General Form of IF Statement

CORRECT USE	EXPLANATION
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 or equal to 10.
IF Q THEN PRINT A ELSE PRINT B; STOP	The value of A is printed if Q is non-zero. If Q=0, then the value of B is printed and the program is terminated.
F A#"STRING" ELSE PRINT "NO MATCH"	Prints "NO MATCH" if value of A is not the string "STRING".
IF X>9 ELSE IF X>5 THEN GOTO 10	Transfers control to statement 10 if X is greater than 5 but less than 10.
IF A=B ELSE IF C THEN GOTO 20	Program goes to next statement if A=B; control is passed to statement 20 if A does not equal B and if C is non-zero.
IF A=B THEN STOP ELSE IF C THEN GOTO 20	Program is terminated if A=B; control is passed to statement 20 if A does not equal B and if C is non-zero.

Figure B. Examples of Correct Usage of Single-Line IF Statement

IF
Statement
(Multi-Line)

The Multi-Line IF statement is functionally identical to the Single-Line IF statement. It provides the conditional execution of a sequence of BASIC statements, or the conditional execution of one of two sequences of statements. The statement sequences, however, may be placed on multiple program lines.

The Multi-Line IF statement is actually an extension of the Single-Line format. With this format, the statement sequences in the THEN and ELSE clauses may be placed on multiple program lines, with each sequence being terminated by an END. There are four general formats of the Multi-Line IF statement as shown in Figure A.

In these forms, either the THEN clause or ELSE clause may be omitted, but at least one must be present. Any statements may appear in the THEN and ELSE clauses.

```
FORM 1:    IF expression THEN
           statements
           ...
           END ELSE statements

FORM 2:    IF expression ELSE
           statements
           ...
           END

FORM 3:    IF expression THEN
           statements
           ...
           END ELSE
           statements
           ...
           END

FORM 4:    IF expression THEN statements ELSE
           statements
           ...
           END
```

Figure A. General Form of Multi-Line IF Statement

CORRECT USE	EXPLANATION

IF ABC=ITEM+5 THEN
  PRINT ABC
  STOP
END ELSE PRINT ITEM; GOTO 10
	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 THEN
  PRINT MESSAGE
  PRINT VAL
  VAL=100
END
	If the value of VAL is non-zero then the value of MESSAGE is printed, the value of VAL is printed, and VAL is assigned a value of 100; otherwise control passes to the next statement following END.

10 IF S="XX" THEN PRINT "OK" ELSE
  PRINT "NO MATCH"
  PRINT S
  STOP
  END
20 REM REST OF PROGRAM
	If the value of S is the string "XX" then the message "OK" is printed and control passes to statement 20; otherwise "NO MATCH" is printed, the value of S is printed and the program terminates.

IF X>1 THEN
  PRINT X
  X=X+1
END ELSE
  PRINT "NOT GREATER"
  GOTO 75
END
	If X>1 the value of X is printed and then incremented, and control passes to the next statement following the second END: otherwise "NOT GREATER" is printed and control passes to statement 75.

Figure B. Examples of Correct Usage of Multi-Line IF Statement

INDEX
Function

The INDEX function searches a string for the occurrence of a sub-string and returns the starting column position of that sub-string.

The general form of the INDEX function is:

```
INDEX(string,sub-string,occurrence)
```

All of the INDEX function parameters may be any valid expressions. The string specifies the string to be examined. The sub-string specifies the sub-string to search for. The occurrence specifies which occurrence of that sub-string is sought. The resultant numeric value of the function is the starting column position of the sub-string within the string. If the sub-string is not found, a value of 0 is returned.

Consider the following example:

```
START = INDEX("ABCDEFGHI","DEF",1)
```

The first occurrence of the sub-string "DEF" starts at column position 4 of the string "ABCDEFGHI". This statement assigns the value of 4 to the variable START.

Next, consider the example:

```
A = INDEX("AAXXAAXXAA","XX",2)
```

The second occurrence of sub-string "XX" starts at column position 7 of string "AAXXAAXXAA". This statement assigns the value of 7 to variable A.

The following example assigns a value of 0 to the variable VAR because the sub-string "Z" is not present within the string "ABC123":

```
  Q = "Z"
  R = "ABC123"
VAR = INDEX(R,Q,1)
```

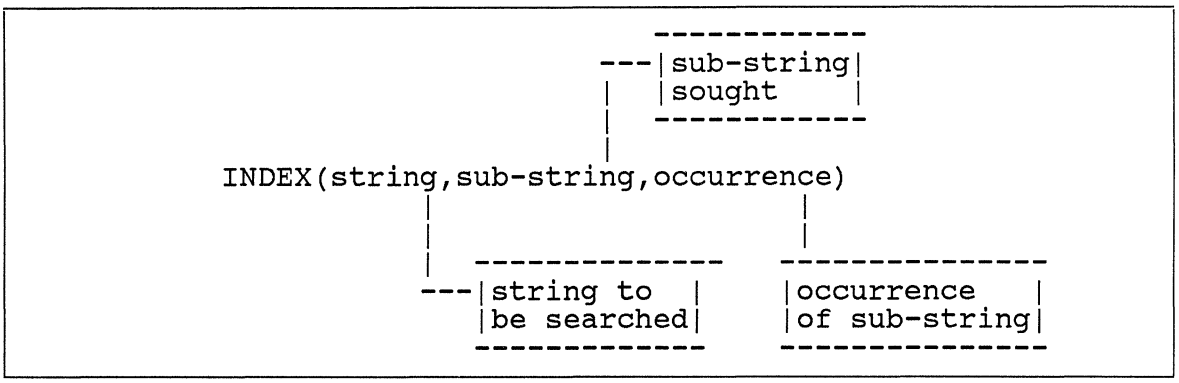


Figure A. General Form of INDEX Function

CORRECT USE	EXPLANATION
A = INDEX("ABCAB","A",2)	Assigns value of 4 to variable A (i.e., 2nd occurrence of "A" is at column position 4 of "ABCAB").
X = "1234ABC" Y = "ABC" IF INDEX(X,Y,1)=5 THEN GOTO 3	The IF statement will transfer control to statement 3 (i.e., "ABC" starts at column position 5 of "1234ABC", so the result of the IF statement is "true".
Q = INDEX("PROGRAM","S",5)	Assigns value of 0 to variable Q (i.e., "S" does not occur in "PROGRAM").
S = "X1XX1XX1XX" FOR I=1 TO INDEX(S,"1",3) NEXT I	The loop will execute 8 times (i.e., 3rd occurrence of "1" appears at column position 8 of the string named S).

Figure B. Examples of Correct Usage of INDEX Function

INPUT
Statement

The INPUT statement is used to request input data from the user's terminal. The cursor position and data format may also be specified. If stacked input is present, the next line of stacked input will be used instead of requesting data from the terminal (see DATA statement).

The general form of the INPUT statement is:

```
INPUT {@(x,y){:}} var {,len}{:}{format}{_}
      {THEN stmt} {ELSE stmt}
```

where: @(x,y) is a specified cursor position
: (after @(x,y)) means display old value as the default before updating
var specifies the variable for storage of input data
len is the maximum length (default is 140 char. maximum)
: (after var or len) means do not echo carriage return and line feed at end of input
format is the format string for input validation and output formatting
_ sounds the bell until <CR> pressed if more than the maximum len (or 140 characters) is entered
THEN signals statements to execute when at least one character is input by the user
ELSE signals statements to execute when only <CR> is pressed by the user

All options must be in the order shown above. An INPUT statement causes a prompt character to be printed at the user's terminal. The user's response is assigned to the specified variable (var).

The @(x,y) option allows the input to be placed at a specified cursor position. The values (x,y) are the coordinates for the cursor location where input is to be supplied (x is the column; y is the row). X and y may be any BASIC expressions. The prompt character is printed one character prior to the x coordinate. If @(x,y) is followed by a colon (:), the existing value of var, if any, is displayed at the (x,y) position, formatted according to format, if present.

If the @(x,y) option is omitted, the prompt character is displayed at the current cursor position. (For more information about prompt characters, please refer to the PROMPT statement, listed alphabetically in this chapter.)

Maximum input is 140 characters unless len specifies a different maximum. If the user enters the maximum number of characters, an automatic <CR> is executed unless the underline (_) option is present. If the optional _ is used, the operator must physically press <CR>, and the bell signal is echoed to the terminal if the operator attempts to enter more than the maximum number of characters before pressing <CR>. If the optional : is used, a <CR> entry will be inhibited on the screen and the cursor will

remain positioned after the input data. The automatic <CR> feature is useful when programming fixed length input fields as it eliminates requiring the operator to enter a carriage return.

Format may contain any ULTIMATE format string characters. (For details on format strings, please refer to Section 2.9, Format Strings.) It may, alternatively, contain any valid Recall Date Conversion Code.

When the @(x,y): options (position cursor and display old value as the default) are used, format is used to format the original value of var and to reformat and re-display the input data after it is entered at the terminal.

If the user presses only the carriage return <CR>, then no input validation or formatting takes place. If the user enters one or more characters and, if needed, a <CR>, the input will be validated against the format, if present.

If format contains a decimal digit specification and/or a scaling factor, then numeric checking will be performed. If format contains a length specification (eg. R#10), then length checking will be performed. If format is 'D' (or any other valid date format) then a date verification will be performed. If the input data does not conform, a warning message is printed at the bottom of the screen, and the user is re-prompted for input. (Also, the type-ahead buffer is cleared, if type-ahead is in effect.) 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
```

Warning messages disappear automatically when a correctly formatted value is input, at which time it is assigned to var.

Note that data is converted on output and input. Thus, if a date is to be input, the default should be stored in internal format; it will be displayed in external format. Input values will be converted from external format and stored in var in internal format; they will be re-displayed in the external format specified by format. For further information on conversions, please see the ICONV and OCONV functions, listed alphabetically in this chapter; also see the ULTIMATE Recall Manual.

Note also that the percent sign ("%") is a numeric character verification symbol. Thus, for example, the statement:

```
INPUT @(20,10):SOC.SEC '%%%-%%-%%%%'
```

may be used to input social security numbers. If the data entered is 423-15-6897, then the variable SOC.SEC will contain the value 423156897, though it will be re-displayed with the embedded hyphens as 423-15-6897.

The THEN and ELSE clauses are 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.

The THEN clause specifies statement(s) to be executed only if the operator enters at least one character of input other than a carriage return. The ELSE clause specifies statement(s) to be executed only if the operator enters no characters, just a carriage return.

If either THEN or ELSE is used, a null input (only <CR>) causes var to retain its old value. If no THEN or ELSE is present, null input stores a null string ('') in var. For example:

```
INPUT A
```

This statement will cause a prompt character to be printed at the user's terminal. The data which the user then inputs, or a null string if only <CR> is pressed, will become the current value of variable A.

NOTE: Two statements, INPUTCLEAR and PRINTERR, and a system function, SYSTEM(11), are useful for type-ahead control in systems with the type-ahead feature. (Please refer to INPUTCLEAR, PRINTERR, and the SYSTEM function listed alphabetically in this chapter.)

```
INPUT {@(x,y){:}} var {,len}{:}{format}{_}
      {THEN stmt} {ELSE stmt}
```

Figure A. General Form of INPUT Statement

CORRECT USE	EXPLANATION
INPUT VAR	Will request a value for variable VAR at the user's terminal.
INPUT X,3	Will request input for variable X. When 3 characters have been entered, an automatic carriage return will be executed.
INPUT X,3_	Same as above but will wait and beep for operator-entered carriage return.
INPUT X:	Requests input for variable X. No carriage return and line feed will be printed after a value is entered.
INPUT ZIP,5:	Requests a value for ZIP. No carriage return and line feed will be printed after a value is entered. Input stops and the program continues if five characters (not including a <CR>) are entered.
INPUT X ELSE STOP	Stops only if a <CR> is entered.
INPUT X THEN RETURN	Returns unless only a <CR> entered.
INPUT @(25,2):INV.DATE 'D'	Inputs a date.
INPUT @(35,7):AMOUNT 'R2,'	Inputs a dollar value.
INPUT @(20,14):NAME 'L#40'	Inputs a text field with a length specification.
INPUT @(0,10):DESC	Inputs data with no mask.

Figure B. Examples of Correct Usage of INPUT Statement

INPUTCLEAR
Statement

The INPUTCLEAR statement allows users with the type-ahead feature to clear the type-ahead buffer for the terminal line on which the BASIC program is running.

The general form of the INPUTCLEAR statement is:

```
INPUTCLEAR
```

When typeahead is enabled (the default case on most systems), users may type in data in anticipation of input requests, before the system has even printed a prompt character and waited for input. 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 erases any previously entered 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 condition(s).

NOTE: The typeahead buffer is also cleared by the PRINTERR statement. (Please refer to the PRINTERR statement, listed alphabetically in this chapter.)

INS
Statement

The INS statement allows a program to insert data into a dynamic array.

The general form of the INS statement is:

```
INS expression BEFORE var < attr# {, value# {, subval#}} >
```

The expression may be any valid BASIC expression. The value of the expression is inserted into the dynamic array specified as var before (preceding) the specified attribute, value, or subvalue expressions. Attr# specifies an attribute, value# specifies a value, subval# specifies a subvalue. If value# and subval# both have a value of 0 (or are both absent, then the expression is an attribute to be inserted before attr#. If subval# only has a value of 0 (or is absent), then expression is a value to be inserted before value#. If attr#, value#, and subval# are all non-zero, then expression is a subvalue to be inserted before subval#.

To insert the expression after the last attribute, value, or subvalue in the dynamic array, -1 should be specified as the value of attr#, value#, or subval#, respectively.

Note: The INSERT function performs the same operation as an intrinsic function rather than a separate statement. (Please refer to the INSERT function, listed alphabetically in this chapter.)

```
INS expression BEFORE var < attr# {, value# {, subval#}} >
```

Figure A. General Form of INS Statement

CORRECT USE	EXPLANATION
INS "JOHN" BEFORE NAMELIST<3,1>	Inserts the string "JOHN" before the first value of the third attribute of dynamic array NAMELIST.
INS "LAST" BEFORE V<1,1,-1>	Inserts the string "LAST" after the last subvalue of the first value in the first attribute of V.

Figure B. Examples of Correct Usage of INS Statement

INSERT
Function

The INSERT function returns a dynamic array with a specified attribute, value, or subvalue inserted.

The general forms of the INSERT function are:

```
INSERT(array-var,attr#,value#,subval#,expr)
                        or
INSERT(array-var,attr# {,value# {,subval#}} ;expr)
```

The value of the expression array-var specifies the dynamic array to insert data into. The values of attr#, value#, and subval# determine whether the data is an attribute, a value, or a subvalue. If value# and subval# both have a value of 0 (or are both absent), then an entire attribute is inserted. If subval# only has a value of 0 (or is absent), then a value is inserted. If attr#, value#, and subval# are all non-zero, then a subvalue is inserted. The value of expr specifies the data to be inserted. The data is normally inserted before the specified attribute, value, or subvalue, except when an index of -1 is used (see below).

The following example shows two ways to code a function:

Using First Form

```
OPEN 'TEST-FILE' ELSE STOP
READ X FROM 'NAME' ELSE STOP
X = INSERT(X,10,0,0,'XXXXX')
WRITE X ON 'NAME'
```

Using Second Form

```
OPEN 'TEST-FILE' ELSE STOP
READ X FROM 'NAME' ELSE STOP
X = INSERT(X,10;'XXXXX')
WRITE X ON 'NAME'
```

These statements insert before attribute 10 of item NAME the value "XXXXX", thus creating a new attribute.

Note that in the second form, trailing zero subvalue or value numbers are not required and that a semicolon separates the attribute, value, and subvalue numbers from the new data expression (expr).

If the last index specified (attr#, value#, or subval#) has a value of -1, the new data is inserted after the last attribute, last value, or last subvalue (respectively) of the dynamic array. For example:

Using First Form

```
OPEN 'FN1' ELSE STOP
READ B FROM 'IT5' ELSE STOP
A = INSERT(B,-1,0,0,'EXAMPLE')
WRITE A ON 'IT5'
```

Using Second Form

```
OPEN 'FN1' ELSE STOP
READ B FROM 'IT5' ELSE STOP
A = INSERT(B,-1;'EXAMPLE')
WRITE A ON 'IT5'
```

These statements insert the string value "EXAMPLE" after the last attribute of item IT5 in file FN1.

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.

```
INSERT(array-var,attr#,value#,subval#,expr)

INSERT(array-var,attr# {,value# {,subval#}} ;expr)
```

dynamic array	attribute	value	sub-value	new value

Figure A. General Forms of INSERT Function

CORRECT USE	EXPLANATION
Y = INSERT(X,3,2,0,"XYZ") or Y = INSERT(X,3,2;"XYZ")	Inserts before value 2 of attribute 3 of dynamic array X the string value "XYZ" (thus creating a new value), and assigns the resultant dynamic array to variable Y.
NEW = "VALUE" TEMP = INSERT(TEMP,9,0,0,NEW) or TEMP = INSERT(TEMP,9;NEW)	Inserts before attribute 9 of dynamic array TEMP the string value "VALUE" (thus creating a new attribute).
A = "123456789" B = INSERT(B,3,-1,0,A) or B = INSERT(B,3,-1;A)	Inserts the value "123456789" after the last value of attribute 3 of dynamic array B.
Z = INSERT(W,5,1,1,"B") or Z = INSERT(W,5,1,1;"B")	Inserts the string value "B" before subvalue 1 of value 1 of attribute 5 in dynamic array W (thus creating a new subvalue), and assigns the resultant dynamic array to variable Z.

Figure B. Examples of Correct Usage of INSERT Function

INT
Function

The INT function returns an integer value.

The general form of the INT function is:

```
INT(expression)
```

The INT function returns the integer portion of the specified numeric expression. (The fractional portion of expression is truncated.) For example:

```
PRINT INT(5.37)
```

This statement causes the value 5 to be printed.

```
INT(expression)
```

Figure A. General Form of INT Function

CORRECT USE	EXPLANATION
A = 3.55 B = 3.6 C = INT(A+B)	Assigns the value 7 to variable C.
J = INT(5/3)	Assigns the value 1 to variable J.

Figure B. Examples of Correct Usage of INT Function

LEN
Function

The LEN function returns the length of a string.

The general form of the LEN function is:

```
LEN(expression)
```

The LEN function returns the length of the string specified by the expression in number of characters. For example:

```
A = "1234ABC"
B = LEN(A)
```

These statements assign the value of 7 to variable B.

```
LEN(expression)
```

Figure A. General Form of LEN Function

CORRECT USE	EXPLANATION
`Q = LEN("123")`	Assigns the value 3 to variable Q (i.e., the length of string "123").

`X = "123"`
`Y = "ABC"`
`Z = LEN(X CAT Y)`
	Assigns the value 6 to variable Z.

Figure B. Examples of Correct Usage of LEN Function

LET
Statement

The LET statement is an optional part of the Assignment (=) statement.

The general form of an Assignment statement using LET is:

```
LET variable = expression
```

For more information, please see the Assignment (=) statement, listed at the beginning of this chapter.

LN
Function

The LN (natural logarithm) function returns the natural logarithm of a number.

The general form of the LN function is:

LN(expression)

The LN (natural logarithm) function generates the natural (base e) logarithm of the expression. Expression must evaluate to a numeric expression. If the value of the expression is less than or equal to zero, the LN function returns a value of zero.

NOTE: The LN function is the inverse of the EXP function. (Please refer to the EXP function, listed alphabetically in this chapter.)

In the following summary M is used to denote the largest allowable number in BASIC, which is 14,073,748,835.5327 with PRECISION 4.

FUNCTION	RANGE	DESCRIPTION
COS(X)	-M <= X <= M	Returns the cosine of an angle of X degrees.
SIN(X)	-M <= X <= M	Returns the sine of an angle of X degrees.
TAN(X)	-M <= X <= M	Returns the tangent of an angle of X degrees.
LN(X)	0 <= X <= M	Returns the natural (base e) logarithm of the expression X.
EXP(X)	-M <= RESULT <= M	Raises the number 'e' (2.7183) to the value of X.
PWR(X,Y)	-M <= RESULT <= M	Raises the first expression to the power denoted by the second expression.

Figure A. Summary of Trigonometric Functions

LOCATE
Statement

The LOCATE statement may be used to find the index of an attribute, a value, or a subvalue within a dynamic array. A starting position for the search may be specified. The elements of the dynamic array may be specified as being in ascending or descending ASCII sequence, and sorted with either right or left justification. If the specified attribute, value, or subvalue is not present in the dynamic array in the proper position, an index value is returned. The index value may be used in an INS statement to place the sought element into its proper location.

There are two general forms for the LOCATE statement. Both operate identically, with a single exception. 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.

The general forms of the LOCATE statement are as follows:

```
LOCATE(expr, item {,attr# {,val#}}; var {; seq} )
      {THEN statements} {ELSE statements}

LOCATE expr1 IN item {<attr# {,val#} >} {, expr2}
       {BY seq} SETTING var {THEN stmt} {ELSE stmt}
```

Note that either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

Both expr and expr1 are the element to be located in dynamic array item; var is the variable into which the index of expr is to be stored. Attr# and val# are optional parameters which restrict the scope of the search within item. Attr# limits the search to a specific attribute. If val# is also present, the search is limited to a specific value in the specified attribute.

The second form of LOCATE allows for further restriction of scope; expr2 is the starting attribute, value, or subvalue number for the search. If expr2 is not present, the default of 1 is assumed.

If none of the above optional parameters are present, expr is tested for equality with any attribute in item, and var returns an attribute number.

If attr# is present, expr is compared with values within the specified attribute, and var returns a value number. If val# is also present, the search is conducted for subvalues of the specified attribute and value, and var returns a subvalue number.

The optional seq must be a string whose first (or only) character is either "A" or "D". Any other values for seq are ignored. If the first character is "A", the elements in item are assumed to be sorted in ascending sequence; if "D", in descending sequence.

The second character of seq, if present, determines the justification to be used when sorting the elements. "R" indicates right justification; this is useful with numeric elements. For any other value, including null, left justification is used.

If the element is located, its index is stored as specified and the THEN statements, if any, are executed. If the element is not located, var contains the index of the correct position (the "next" position) in the array where the element could be inserted. (Please refer to the INS statement or the INSERT function, listed alphabetically in this chapter.)

For example, a program may need to locate and/or insert data within a dictionary item. One statement can perform both the LOCATE and INSERT operations:

```
LOCATE('D',ITEM,4;VAR) ELSE ITEM = INSERT(ITEM,4,VAR,0,'D')
```

If the string 'D' is not found in attribute 4 of ITEM, it is inserted as the "next" value number (with no subvalues).

NOTE: This single statement 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.

```
LOCATE(expr, item {,attr# {,val#}}; var {; seq} ) {THEN stmt} {ELSE stmt}

LOCATE expr1 IN item {<attr# {,val#} >} {, expr2}
       {BY seq} SETTING var {THEN stmt} {ELSE stmt}
```

where:

expr/expr1	Sought element. May be a literal string, variable, array element or function.
item	A dynamic array.
attr#	Optional: if present restricts scope of search to specified attribute. Returns value number index.
val#	Optional: if present restricts scope of search to specified value. Returns subvalue number index.
expr2	Optional in second form: if present, specifies starting point (attribute, value, subvalue) of search.
var	Index of sought element returned in Var.
seq	Optional: 'A', 'D', 'AL', 'DL', 'AR', or 'DR'. If seq is not specified and the string is not found the default will be to the last position.

Figure A. General Forms of LOCATE Statement.

CORRECT USE:

```
LOCATE('55',ITEM,3,1;VAR;'AR') ELSE ITEM = INSERT(ITEM,3,1,VAR,'55')
```

EXPLANATION

The third attribute, first value of dynamic array 'ITEM' is searched for the numeric literal '55'. 'VAR' will return with the subvalue index if the numeric is found, and will return with the correct subvalue index if the numeric is not found.

If it is not found, control passes to the ELSE clause which inserts the numeric into the correct position by virtue of the index contained in 'VAR'. The optional parameter 'AR' specifies ascending sequence and right justification.

CORRECT USE:

```
LOCATE(STR,REC;VAR) THEN NAME=REC<VAR> ELSE NAME='INVALID'
PRINT NAME
```

EXPLANATION

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.

CORRECT USE:

```
LOCATE "JOHN" in NLIST<3> SETTING X ELSE INS "JOHN" BEFORE NLIST<3,X>
```

EXPLANATION

If "JOHN" is not found, it is inserted at the correct position.

OTHER CORRECT EXAMPLES OF THE SECOND FORM OF LOCATE:

```
LOCATE DESC IN NLIST SETTING X ELSE GOTO 100

LOCATE DATE IN NLIST <4> BY "AR" SETTING X THEN DEL NLIST<4,X>

LOCATE NEXT IN NLIST <7>,LAST SETTING X ELSE NULL
```

EXPLANATION

These examples illustrate locating attributes and values, using the sequence option, and specifying a starting location for a search.

Figure B. Examples of Correct Usage of the LOCATE Statement

LOCK
Statement

The LOCK statement provides a file and execution lock capability for BASIC programs. The LOCK statement sets execution locks. This statement works in conjunction with the UNLOCK statement, which unsets the locks.

The general form of the LOCK statement is:

```
LOCK expression {ELSE statements}
```

The value of the expression specifies which execution lock is to be set (0-47). If the execution lock is currently unlocked, the statement sets the lock. If the lock is already set by the current program, the LOCK statement has no effect. If the specified execution lock has already been set by another concurrently running program (and the ELSE clause is not used), then program execution will temporarily halt until the lock is released by the other program. If the ELSE clause is used, then the statement(s) following the ELSE will be executed. The statements in the ELSE clause 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.

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 will either execute an alternate set of statements or will pause until the lock is released (via an UNLOCK statement) by the program which set the lock.

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 a complete execution lock sequence. Process A sets execution lock 42 before executing a non-reentrant section of code (that is, code which should not be executed by more than one process simultaneously). Process B executing the same program reaches the "LOCK 42" instruction, but cannot lock that section of code until Process A has unlocked 42. Thereby, the code is rendered non-reentrant.

When a program terminates execution (for any reason, including the BASIC Debugger END command), any execution locks still locked for that line are unlocked.

NOTE: The UNLOCK statement is discussed in a separate topic. (Please refer to the UNLOCK statement, listed alphabetically in this chapter.)

```
LOCK expression {ELSE statements}
```

Figure A. General Form of LOCK Statement

CORRECT USE	EXPLANATION
LOCK 15 ELSE STOP	Sets execution lock 15. if lock 15 is already set, program will terminate.
LOCK 2	Sets execution lock 2. If lock 2 is already set program will temporarily halt until lock 2 is released.
LOCK 10 ELSE PRINT X; GOTO 5	Sets execution lock 10; if lock 10 is already set, the value of X will be printed and the program will branch to statement 5.

Figure B. Examples of Correct Usage of LOCK Statement

LOOP
Statement

The LOOP statement constructs a program loop.

The general forms of the LOOP statement are:

```
LOOP {statements} {WHILE expression DO {statements}} REPEAT
LOOP {statements} {UNTIL expression DO {statements}} REPEAT
```

Both the WHILE clause and UNTIL clause are optional. If neither clause is used, an endless loop can be constructed which executes all statements (if any) between LOOP and REPEAT repeatedly until 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.)

Execution of a LOOP statement with a WHILE or UNTIL clause proceeds as follows. First the statements (if any) following "LOOP" will be executed. Then the WHILE or UNTIL expression is evaluated. One of the following is then performed depending upon the form used:

1. If the "WHILE" form is used, then the statements following "DO" (if any) will be executed. If the expression evaluates to true (i.e., non-zero), program control will loop back to the beginning of the loop. If the expression evaluates to false (i.e., zero), program control will proceed with the next sequential statement following "REPEAT" (i.e., control passes out of the loop).

2. If the "UNTIL" form is used, then the statements following "DO" (if any) will be executed. If the expression evaluates to false (i.e., zero), program control will loop back to the beginning of the loop. If the expression evaluates to true (i.e., non-zero), program control will proceed with the next sequential statement following "REPEAT" (i.e., control passes out of the loop).

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

Consider the following example:

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

Assuming that the value of variable A is 0 when the LOOP statement is first executed, this statement will print the sequential values of A from 1 through 4 (i.e., the loop will execute 4 times). As a further example, consider the statement:

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

Assuming, for instance, that the value of variable X is 100 when the above LOOP statement is first executed, this statement will print the values of X from 90 down through 50 in increments of -10 (i.e., the loop will execute 5 times).

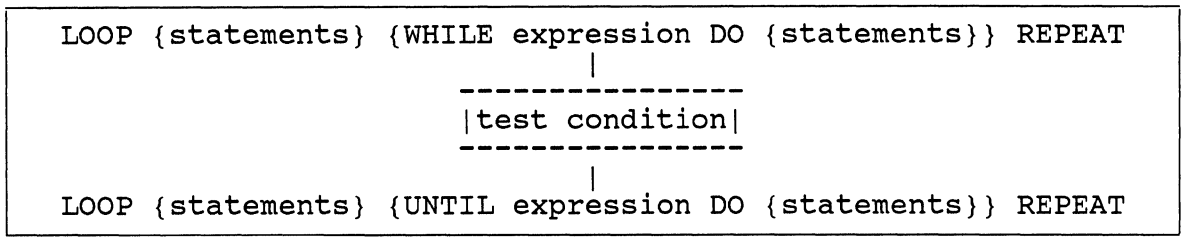


Figure A. General Forms of LOOP Statement

CORRECT USE	EXPLANATION
J=0 LOOP PRINT J J=J+1 WHILE J<4 DO REPEAT	Loop will execute 4 times (i.e., sequential values of variable J from 0 through 3 will be printed.
Q=6 LOOP Q=Q-1 WHILE Q DO PRINT Q REPEAT	Loop will execute 5 times (i.e., values of variable Q will be printed in the following order: 5, 4, 3, 2, and 1).
Q=6 LOOP PRINT Q WHILE Q DO Q=Q-1 REPEAT	Loop will execute 7 times (i.e., values of variable Q will be printed in the following order: 6, 5, 4, 3, 2, 1, and 0).
B=1 LOOP UNTIL B=6 DO B=B+1 PRINT B REPEAT	Loop will execute 5 times (i.e., sequential values of variable B from 2 through 6 will be printed).

Figure B. Examples of Correct Usage of LOOP Statement

MAT =
Statement
(Assignment and Copy)

The MAT = (Assignment and Copy) statements are used to assign a value to each element in the array. The MAT Assignment statement assigns a specified value to all elements. The MAT Copy statement copies one array to another.

The general form of the MAT Assignment statement is:

```
MAT variable = expression
```

The value of the expression (which may be any legal expression) is assigned to each element of the array. The variable specifies the array, which must have been previously dimensioned via a DIM statement. The following statement, for example, assigns the current value of X+Y-3 to each element of array A:

```
MAT A = X+Y-3
```

The general form of the MAT Copy statement is:

```
MAT variable = MAT variable
```

The first element of the array variable on the right becomes the first element of the array variable on the left, the second element on the right becomes the second element on the left, and so forth. Each variable name must have been dimensioned, and the number of elements in the two arrays must match; if not, an error message occurs.

Arrays are copied in row major order, i.e., with the second subscript (column) varying first. 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 as both having ten elements (5x2=10), initializes array Y elements to the numbers 1 through 10, and then copies array Y to array X, giving the array elements the indicated values.

NOTES: 1. The MAT = statement assigns values to dimensioned arrays. The = (Assignment) statement assigns a value to a simple variable. See the =

(Assignment) statement in this chapter; the = statement precedes statements that begin with "A".)

2. The MAT = statement is not the same as the MAT used in an argument list for passing arrays in the CALL and SUBROUTINE statements. (See CALL and SUBROUTINE statements, listed alphabetically in this chapter.)

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

Figure A. General Forms of MAT Assignment and Copy Statements

CORRECT USE	EXPLANATION
`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 vectors of equal length, and assigns 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 (10x10=50x2), and copies TAB2 values to TAB1 in row major order.

Figure B. Examples of Correct Usage: MAT Assignment and Copy Statements

MATREAD
Statement

The MATREAD statement reads a file item and assigns the value of each attribute to consecutive vector elements.

The general form of the MATREAD statement is:

MATREAD var FROM {file-var,} item-id {ON ERROR statements}
{THEN stmt} {ELSE stmt}

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

The MATREAD statement reads the file item specified by the item-id expression and assigns the string value of each attribute to consecutive elements of the dimensioned array vector specified by variable var. If the file-var is used, the item will be read from the file previously assigned to that variable via an OPEN statement. If the file-var is omitted, then the internal default file variable is used (that is, the file most recently opened without a file variable). The THEN statements, if any, are then executed.

Unlike the MATREADU statement, the MATREAD statement can successfully read an item even if it is a locked item or is in a locked file group. (For details, see the MATREADU statement, listed alphabetically in this chapter.)

If the item specified in the item-id does not exist, the contents of the array var remain unchanged. Then the ELSE statements, if any, will be executed. The ELSE clause statements may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multiple-line IF statement.

For example:

```
MATREAD IN FROM 'ITEM' ELSE STOP
```

This statement will read into array "IN" the item named "ITEM" from the file most recently opened without a file variable. If ITEM does not exist, the program stops. Note: 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, will be assigned to the last element of the array.

NOTE: An overview of the ULTIMATE file structure is given in Section 5.1, along with a summary of the standard ULTIMATE file delimiters (attribute, value, and subvalue).

NOTE: For file updating, also see the MATWRITE statement and its variations, listed alphabetically in this chapter.

The statement(s) after ON ERROR, if present, are executed only if the file (1) is a remote file (accessed via UltiNet) and (2) cannot be read due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 ON ERROR clause 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.

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 read due to network errors, the program may terminate with an error message if no ON ERROR clause is present.

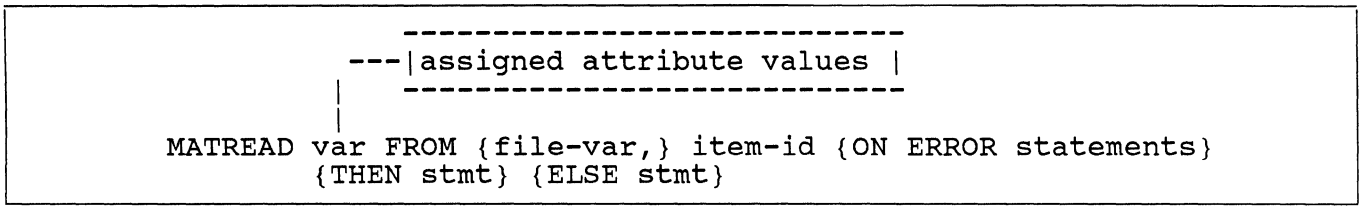


Figure A. General Form of MATREAD Statement

CORRECT USE	EXPLANATION
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 vector 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, or retrieves error number and performs local subroutine on UltiNet error number.

Figure B. Example of Correct Usage of MATREAD Statement

MATREADU
Statement

The MATREADU statement provides the facility to lock a disk file item prior to updating it. This can be used to prevent updating an item by two or more programs simultaneously while still allowing multiple programs to access the file.

The general form of the MATREADU statement is:

```
MATREADU var FROM {file-var,} item-id {ON ERROR stmt)
        {LOCKED stmt} {THEN stmt} {ELSE stmt}
```

The MATREADU statement functions like the MATREAD statement, but additionally locks the item lock associated with the item to be accessed. If the item is currently locked by another BASIC program, the statement will not perform the read operation. 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.

Item locks are assigned based on (1) the group of the disk file which contains (or would contain) an item and on (2) 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 the LOCKED clause is present, the statements specify an action to take if the MATREADU statement is unable to lock the item because another program has already locked it.

If an item is currently unlocked, setting a corresponding item lock will prevent 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, will be 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, is updated by a WRITE, WRITEV, or MATWRITE 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, it will be 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, which does not pay attention to item locks.

```
MATREADU var FROM {file-var,} item-id {ON ERROR stmt}
         {LOCKED stmt} {THEN stmt} {ELSE stmt}
```

Figure A. General Form of MATREADU Statement

CORRECT USE	EXPLANATION
MATREADU T FROM XM, "N4" ELSE NULL	The item will be 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.

Figure B. Example of Correct Usage of MATREADU Statement

MATWRITE
Statement

The MATWRITE statement writes a file item with the contents of a dimensioned array vector. If the item was initially locked, it is unlocked.

The general form of the MATWRITE statement is:

MATWRITE var ON {file-var,} item-id {ON ERROR stmts}

The MATWRITE statement writes the contents of the dimensioned array vector specified by the array var to the disk file item specified by the expression item-id. The statement replaces the attributes of item-id with the string value of the consecutive elements of the vector named by var. If the file-var is used, the item will be written in the file previously assigned to that variable via an OPEN statement. If the file-var is omitted, then the internal default file variable is used (that is, the file most recently opened without a file variable).

If the item-id specifies an item which does not exist, then a new item will be created.

When updating an existing file item, none of the existing attributes are retained. The number of attributes written will generally be the same as the total number of attributes (elements) in the array, except that null attributes at the end of the item will be deleted. Normally, each element of the array will consist of one attribute, though this is not required; the MATREAD statement, for instance, will store several attributes in the last element of an array if an item has more attributes than the array has elements. MATWRITE will assign a value of 0 to all attributes whose corresponding array elements have not been assigned a value.

The MATWRITE statement clears the item lock associated with the item being written, if it was initially locked. 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 statement(s) after ON ERROR, if present, are executed only if the file (1) is a remote file (accessed via UltiNet) and (2) cannot be written to due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 to local files.

The ON ERROR clause 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.

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 written to due to network errors, the program may terminate with an error message if no ON ERROR clause is present.

```
MATWRITE var ON {file-var,} item-id {ON ERROR statements}
```

Figure A. General Form of MATWRITE Statement

```
CORRECT USE                          EXPLANATION

DIM ITEM (10)                        Writes an item named JUNK
OPEN 'TEST' ELSE STOP                in the file named TEST.
FOR I=1 TO 10                        The item written will contain
ITEM(I)=I                            10 attributes whose string
NEXT I                               values are 1 through 10.
MATWRITE ITEM ON "JUNK"

MATWRITE ITEM ON "JUNK" ON ERROR
  ERRNUM=SYSTEM(0)                   Writes as above, or retrieves
  GOSUB PROCESSERR                   error number and performs
  GOTO TOP                           local subroutine on UltiNet
END                                  error number.
```

Figure B. Example of Correct Usage of MATWRITE Statement

MATWRITEU
Statement

The MATWRITEU statement writes a file item with the contents of a dimensioned array vector. The item remains locked after execution of the MATWRITEU statement. The letter "U" is appended to the statement name to imply "update", not "unlock".

The general form of the MATWRITEU statement is:

MATWRITEU var ON {file-var,} item-id {ON ERROR stmt}

The MATWRITEU statement functions the same as the MATWRITE statement except for the locking feature. It does not unlock the item after completing the write operation. This variation on the MATWRITE statement is used primarily for master file updates when several transactions are being processed and an update of the master item is made following each transaction update.

If the item is not locked before the MATWRITEU statement is executed, it will be locked afterwards. For more information on item locks, please refer to the MATREADU, READU, or READVU statements, listed alphabetically in this chapter.

NOTE: The RELEASE statement can be used to unlock an item. (Please refer to the RELEASE statement, listed alphabetically in this chapter.)

```
MATWRITEU variable ON {file-var,} item-id {ON ERROR stmts)
```

Figure A. General form of MATWRITEU Statement

CORRECT USAGE	EXPLANATION
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 vector 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.

Figure B. Example of Correct Usage of MATWRITEU Statements

MOD
Function

The MOD function generates the remainder of one number divided by another.

The general form of the MOD function is:

```
MOD (dividend,divisor)
```

Both dividend and divisor are expressions. The MOD function returns the remainder from the (integer) division of dividend by divisor.

Note: The MOD function is the same as the REM function, listed alphabetically in this chapter.

```
MOD (dividend,divisor)
```

Figure A. General Form of MOD Function

CORRECT USE	EXPLANATION
Q = MOD(11,3)	Assigns the value 2 to variable Q.

Figure B. Example of Correct Usage of MOD Function

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.

The general form of the NEXT statement is:

```
NEXT variable
```

The variable in the NEXT statement must be the same as the variable in the FOR statement.

A loop is a portion of a program written in such a way that it will execute repeatedly until some test condition is met. A FOR-NEXT loop causes execution of a set of statements for successive values of a variable until a limiting value is encountered. The FOR statement is discussed in a separate section. (Please refer to the FOR statement, listed alphabetically in this chapter.)

The function of the NEXT statement is to return program control to the beginning of the loop after a new value of the variable has been computed. As an example, consider the execution of the following statements:

```
150 FOR J=2 TO 11 STEP 3
160 PRINT J+5
170 NEXT J
```

Statement 150 sets the initial value of J to 2 and specifies that J thereafter will be incremented by 3 each time the loop is performed, until J exceeds the limiting value 11. Statement 160 prints out the current value of the expression J+5. Statement 170 assigns J its next value (i.e., J=2+3=5) and causes program control to return to statement 150. Statement 160 is again executed, and statement 170 again increments J and causes the program to loop back. This process continues with J being incremented by 3 after each pass through the loop. When J attains the limiting value of 11, statement 160 will again be executed and control will pass to 170. J will again be incremented (i.e., J=11+3=14), and since 14 is greater than the limiting value of 11, the program will "fall through" statement 170 and control will pass to the next sequential statement following statement 170.

```
NEXT variable
```

Figure A. General Form of NEXT Statement

CORRECT USE	EXPLANATION
FOR A=1 TO 2+X-Y . . NEXT A	Limiting value is current value of expression 2+X-Y; increment value is +1.
FOR K=10 TO 1 STEP -1 . . NEXT K	Increment value is -1 (i.e., variable K will decrement by 1 for each of 10 passes through the loop).

Figure B. Examples of Correct Usage of FOR and NEXT Statements

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.

The general form of the NOT function is:

```
NOT(expression)
```

The NOT function returns the logical inverse of the specified expression; it returns a value of true (i.e., generates a value of 1) if the expression evaluates to 0, and returns a value of false (i.e., generates a value of 0) if the expression evaluates to a non-zero quantity. The specified expression must evaluate to a numeric quantity or a numeric string. The following statement, for example, assigns the value 1 to the variable X:

```
X = NOT(0)
```

As a futher example, the following statements cause the value 0 to be printed:

```
A = 1
B = 5
PRINT NOT(A AND B)
```

```
NOT(expression)
```

Figure A. General Form of NOT Function

CORRECT USE	EXPLANATION
X=A AND NOT(B)	Assigns the value 1 to variable X if current value of variable A is 1 and current value of variable B is 0. Assigns a value of 0 to X otherwise.
IF NOT(X1)THEN STOP	Program terminates if current value of variable X1 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.

Figure B. Examples of Correct Usage of NOT Function

NULL
Statement

The NULL statement specifies a non-operation. It may be used anywhere in the program where a BASIC statement is required.

The general form of the NULL statement is:

```
NULL
```

The NULL statement is used in situations where a BASIC statement is required, but no operation or action is desired. Consider the following example:

```
INPUT X ELSE NULL
```

This statement 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.

```
NULL
```

General Form of NULL Statement

CORRECT USE	EXPLANATION
10 NULL	This statement does not result in any operation or action; however, since it 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.
IF A=0 THEN NULL ELSE PRINT "A NON-ZERO" GOSUB 45 STOP END	If the current value of variable A is non-zero, the sequence of statements following the ELSE will be executed. If A=0, no action is taken and control passes to the next sequential statement following the END.
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. (Refer to description of READ statement for more information.)

Figure B. Examples of Correct Usage of NULL Statement

NUM
Function

The NUM function returns a value of true (1) if the given expression evaluates to a number or a numeric string.

The general form of the NUM function is:

```
NUM(expression)
```

The NUM function tests the given expression for a numeric value. If the expression evaluates to a number or numeric string the NUM function will return a value of true (i.e., a value of 1). Inversely, an expression evaluating to an alphabetic or other non-numeric string will cause the NUM function to return a value of false.

The NUM function considers a numeric string to be one which is either:

(1) a sequence of decimal digits, optionally preceded by a plus or minus sign, and optionally containing a decimal point, or

(2) a null string ('')

Consider the following examples:

```
IF NUM(X) THEN PRINT "NUMERIC DATA"
```

This statement will print the text "NUMERIC DATA" if the current value of variable X is a number or a numeric string.

```
NUM(expression)
```

Figure A. General Form of NUM Function

CORRECT USE	EXPLANATION
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.

Figure B. Examples of Correct Usage of NUM Function

OCONV
Function

The OCONV function converts a string according to a specified type of output conversion.

The general form of the OCONV function is:

```
OCONV(string,code)
```

The string specifies a string value. The code specifies the type of output conversion. The resultant value is always a string value.

The value of code must be a string. The following codes may be used for output conversions:

- D Convert date to external format
- G Extract group of characters
- L Test string length
- MC Mask characters by numeric, alpha, or upper/lower case
- ML Mask left-justified decimal data
- MP Convert packed decimal number to integer
- MR Mask right-justified decimal data
- MT Convert time to external 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 Recall Reference Manual for details.) NOTE: This type of conversion is inefficient if several items or attributes will be accessed.
- U Convert by subroutine call to assembly routine, either system- or user-defined. The absolute address of the routine must be given. The value of 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, please refer to the Assembler Manual.)

The conversion codes are the same as those used for Recall Conversions and Correlatives. For a detailed treatment of these and other conversion capabilities, refer to the ULTIMATE Recall Reference Manual.

WARNING: Some conversion codes used in Recall, such as 'F', cannot be used in the OCONV function. Also note that 'MR' and 'ML' conversions may be done with format strings. (For details on format strings, refer to the chapter "Representing Data" in this manual.)

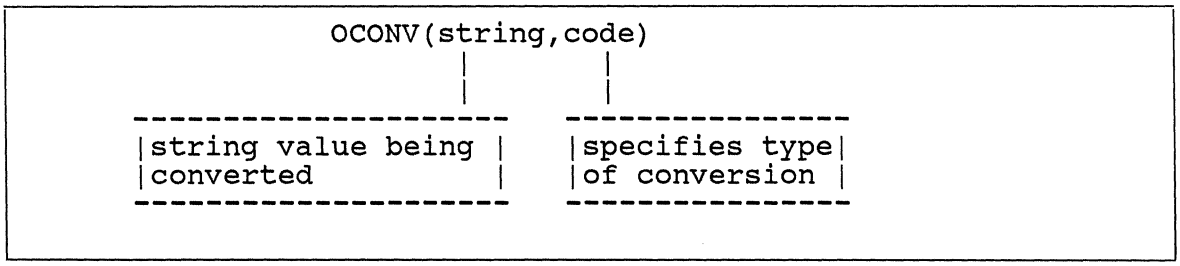


Figure A. General Form of OCONV Function

CORRECT USE	EXPLANATION
COLOR=OCONV("RED^BLUE^WHITE", "G1^1")	Extracts "BLUE" from the string and assigns it to the variable COLOR.
A="2374" B="D" XDATE = OCONV(A,B)	Assigns the string value of "01 JUL 1974" (i.e., the external date) to the variable XDATE.
A = OCONV(0, 'U50BB') PRINT A END	Assigns the string value of the line number and user account name to A. User entry '50BB' is the mode-ID of the system subroutine that performs the above operation.
TEAMS=OCONV("TEEMS","TGAMES;X;1;1")	Reads the first attribute of item "TEEMS" in the file "GAMES". Consult the section 'Defining File Translation' in RECALL Manual.

Figure B. Examples of Correct Usage of OCONV Function

ON GOSUB and ON GOTO
Statements

The ON GOSUB and ON GOTO statements transfer program control to a computed and selected subroutine or statement-label. The ON GOSUB statement selects a subroutine, and the ON GOTO statement selects a statement-label, by the current value of an index expression.

The general form of the ON GOTO statement is:

ON expression GOTO statement-label, statement-label,...

The expression is evaluated, and the result is truncated to an integer value. The value of this index is used to compute the statement to which control should be transferred. Statement-labels in the list are numbered 1, 2, 3, and so on. If the index is 1, statement-label 1 (the first label) is selected; if 2, statement-label 2 (the second label) is selected, and so on. Upon execution of the ON GOTO statement, program control is transferred to the statement which begins with the statement-label selected by the expression.

The ON GOTO statement may continue on multiple lines; each line except the last must conclude with a comma.

Consider the following example:

```
    ON I GOTO 50, 100, LAST
. 50  .
. 100 .
. LAST: .
```

(The labels in the label list may precede or follow the ON GOTO statement.) If the current value of variable I=1, control transfers to the first statement-label, i.e., the statement with label 50. If I=3, control transfers to the third statement-label, i.e., the statement with label LAST.

If the value of the expression evaluates to less than one or greater than the number of statement-labels, no action is taken; that is, the statement immediately following the ON GOTO will be executed next.

The ON GOSUB statement is a "computed" GOSUB statement: an expression is used to compute which subroutine to execute next. The general form of the ON GOSUB statement is:

ON expression GOSUB statement-label, statement-label, ...

The expression is evaluated and truncated to an integer value. The result is used as an index into the list of statement-labels. The ON GOSUB statement may continue on multiple lines; each line except the last must conclude with a comma. A subroutine branch is executed to the selected

statement-label. When a RETURN statement is encountered, control returns to the statement following the ON GOSUB.

If the expression evaluates to less than 1 or to a value greater than the number of statement-labels, no action is taken; that is, the statement immediately following the ON GOSUB will be executed next.

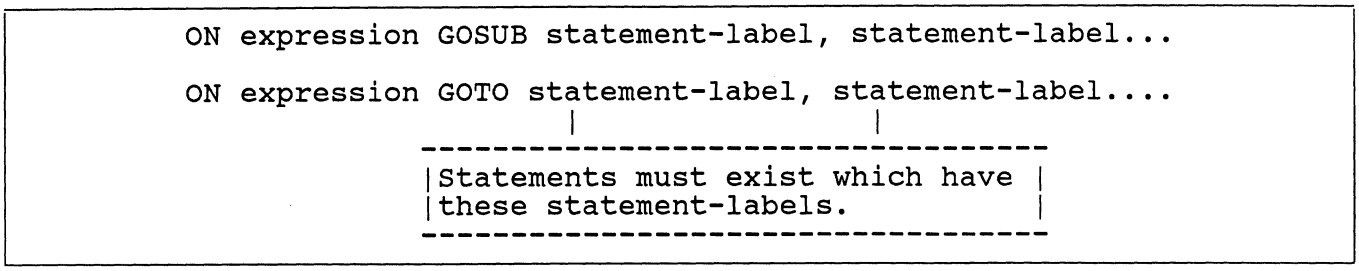


Figure A. General Form of ON GOSUB and ON GOTO Statements

CORRECT USE	EXPLANATION
ON M+N GOTO 40, 61, 5, 7	Transfer control to statement 40, 61, 5, or 7 depending on the value of M+N being 1, 2, 3, or 4 respectively.
ON C GOTO ELEMENTARY, ELEMENTARY, ADVANCED.	Transfer control to statement-label ELEMENTARY is C = 1 or 2; to statement-label ADVANCED is 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.
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.
START: ON CHECK GOSUB ONE, TWO, THREE	Branches to subroutine located at ONE, TWO, or THREE, depending on value of variable CHECK.

Figure B. Examples of Correct Usage of ON GOSUB and ON GOTO Statements

INCORRECT USE	EXPLANATION
ON M+N=1 GOTO 40, 61, 5, 7	Index should be an arithmetic, not logical, quantity. This statement, if executed, would cause an unconditional jump to statement 40.

Figure C. Example of Incorrect Usage of ON GOSUB and ON GOTO Stmts.

OPEN
Statement

The OPEN statement is used to select an ULTIMATE file for subsequent input, output, or update.

Before an ULTIMATE file can be accessed by a read, write, or updating statement (e.g., READ, WRITE, DELETE, MATREAD, MATWRITE, READV, or WRITEV), it must be opened via an OPEN statement.

The general form of the OPEN statement is:

```
OPEN {expr,} file {TO file-var} {ON ERROR statements}
     {THEN stmt} {ELSE stmt}
```

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

An ULTIMATE file name must be specified. This is a string expression in one of the following formats:

1. "filename"
2. "DICT filename"
3. "DATA filename"
4. "dictname,filename"
5. "DATA dictname,filename"

For all except format 2, above, the data section of the file, not the dictionary, is opened. Format 3 is equivalent to format 1, and format 5 is equivalent to format 4. Formats 4 and 5 are useful with data sections of files having names different from that of their dictionary section.

The optional expr may also be used to specify that the dictionary section of the file is to be opened. Expr is prefixed to file to form the complete file name, so expr is typically "DICT". If expr is the null string (''), then file is the complete file name.

If the "TO file-var" option is used, then the dictionary or data section of the file will be assigned to the specified variable for subsequent reference. If the "TO file-var" option is omitted, then an internal default file variable is used; subsequent I/O statements not specifying a file variable will then automatically default to this file.

The statement(s) after ON ERROR, if present, are executed only if the file (1) is a remote file (accessed via UltiNet) and (2) cannot be opened due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 ON ERROR clause 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.

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 opened due to network errors, the program may terminate with an error message if no ON ERROR clause is present.

After the file is opened, the THEN statements, if any, are executed. If the ULTIMATE file indicated in the OPEN statement does not exist, then the ELSE statements, if any are executed. The statements in the ELSE clause may be placed on the same line separated by semicolons, or may be placed on multiple lines terminated by an END, as in the multi-line IF statement.

There is no limit to the number of files that may be open at any given time. Consider the following example:

```
OPEN "DICT","QA4", TO F1 ELSE PRINT "NO FILE"; STOP
```

This statement will open the dictionary portion of the file named QA4 and will assign it to variable F1. If QA4 does not exist, the message "NO FILE" will be printed and the program will terminate. The data portion of a file named TEST is opened as illustrated below:

```
OPEN 'TEST' ELSE
    PRINT "TEST DOES NOT EXIST"
    GOTO 100
END
```

In this example, the file is assigned to an internal default file variable. The message "TEST DOES NOT EXIST" will be printed and control will pass to statement 100 if the file named TEST does not exist.

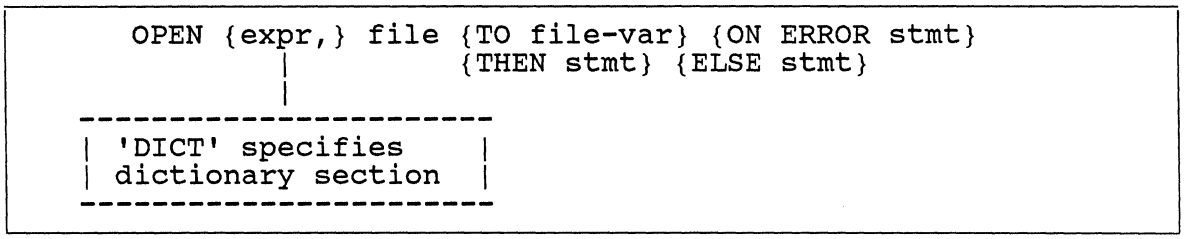


Figure A. General Form of OPEN Statement

CORRECT USE	EXPLANATION
A='DICT' OPEN A, 'XYZ' TO B ELSE PRINT "NO XYZ" STOP END	Opens the dictionary portion of file XYZ and assigns it to variable B. If XYZ does not exist, the text "NO XYZ" is printed and the program terminates.
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.
X='' Y='TEST1' Z='NO FILE' OPEN X, Y ELSE PRINT Z; GOTO 5	Opens data section of file TEST1 and assigns it to internal default variable. If TEST1 does not exist, "NO FILE" is printed and control passes to statement 5.
OPEN 'DICT DT' TO DT ELSE STOP	Opens DICT of file DT and assigns it to variable DT.
OPEN 'EMPLOY' TO EMP ELSE STOP	Opens DATA section of the file EMPLOY and assigns it to variable EMP
OPEN 'EMPLOY' TO EMP ON ERROR ERRNUM=SYSTEM(0) GOSUB PROCESSERR GOTO TOP END ELSE STOP	Opens as above, or retrieves error number and performs local suroutine on UltiNet error number.

Figure B. Examples of Correct Usage of OPEN Statement

PAGE
Statement

The PAGE statement causes the current output device to start a new page. The page number may optionally be reset.

The general form of the PAGE statement is:

PAGE {expression}

The expression optionally specifies the 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 on the page just ending will be printed with a page number one less than expression.

The maximum number of print lines per page is controlled by the current TERM command. (For details, please refer to the TERM command in the System Commands manual.) The PAGE statement allows users to end a page before the maximum number of lines has been reached and automatic paging occurs.

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.

If only a footing is desired, a null HEADING statement should be assigned. The PAGE statement will cause a new page to be started even if no heading or footing has been assigned.

```
PAGE {expression}
```

Figure A. General Form of PAGE Statement

CORRECT USE	EXPLANATION
HEADING "ANNUAL STATISTICS" FOOTING "XYZ CORPORATION" PAGE	The PAGE statement will cause both the specified heading and footing to be printed out when the paging is executed.
PAGE 1	The current footing, if any, will print (with a page number of 0). The current heading, if any, will print with a page number of 1.
PAGE X+Y	The current footing and heading will be output, and the page number set to the evaluated result of X+Y.

Figure B. Examples of Correct Usage of PAGE Statement

PRECISION
Statement

The PRECISION declaration statement allows the user to select the degree of precision to which all numeric values are calculated within a given program.

The general form of the PRECISION statement is:

```
PRECISION n
```

where n is a number from 0 to 9.

The default precision value is 4; that is, all numeric values are normally stored in an internal form with 4 fractional places, and all computations are performed to this degree of precision. A program may specify the desired number of fractional digits by a PRECISION declaration within the range of 0 to 9.

Only one PRECISION declaration 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 will be printed.

Changing the precision changes the acceptable form of a number; a number is defined as having a maximum of "n" fractional digits, where "n" is the precision value. Thus, the value:

```
1234.567
```

is a legal number if the precision is 3 or 4, but is not a legal number if the precision is 0, 1 or 2.

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. (See Section 2.7 for an overview of floating point and string arithmetic.)

Figure C illustrates some problems to avoid when using PRECISION declaration statements.

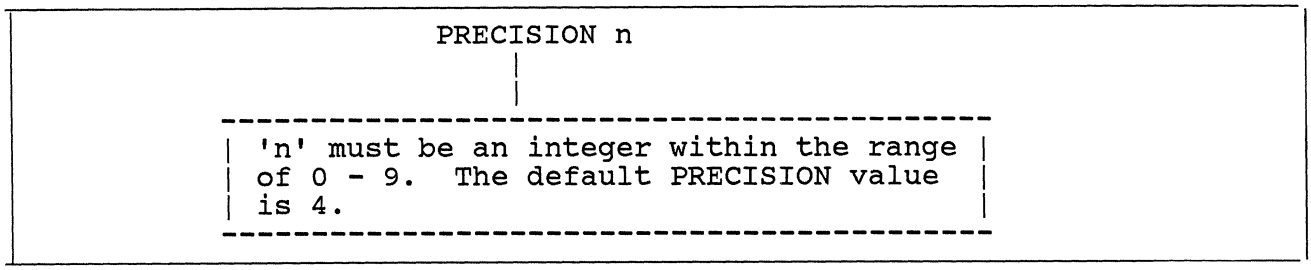


Figure A. General Form of PRECISION Declaration

CORRECT USE	EXPLANATION
PRECISION 0 A = 3 B = A/2	All numeric values in the program will be treated as integers. The value returned for B will be 1, not 1.5.
PRECISION 1	All numeric values in the program will be calculated to one fractional digit.
PRECISION 2	All numeric values in the program will be calculated to two fractional digits.
PRECISION 3	All numeric values in the program will be calculated to three fractional digits.

Figure B. Examples of Correct Usage of PRECISION Declaration

INCORRECT USE	EXPLANATION
PRECISION 2 A = B + C PRECISION 3	PRECISION may be set only once within a given program. Otherwise a warning message is issued and the second PRECISION declaration is ignored.
PRECISION 2 CALL SUBA SUBROUTINE SUBA PRECISION 3	PRECISION must be the same for the mainline program and any subroutine it calls. Otherwise a warning message is issued and the second PRECISION declaration is ignored.

Figure C. Examples of Incorrect Usage of the PRECISION Declaration

PRINT
Statement

The PRINT statement outputs data to the current output device, typically the terminal or the line printer. The PRINT ON option allows output to multiple print files.

The general form of the PRINT statement is:

```
PRINT {ON expression} {print-list}
```

The PRINT statement without the ON option is used to output variable or literal values to the terminal or other output destination. By default, output is to the terminal. The PRINTER statement may be used to route output from PRINT statements to the line printer or other spooled output destination. (Refer to the PRINTER statement, listed alphabetically in this chapter.) The "P" option on the RUN command will also route output to a line printer or other spooled output destination. Furthermore, if a program is invoked via an EXECUTE statement, its output may be redirected or captured, and therefore appear neither on the terminal nor on the line printer. (For more information on output redirection, refer to the EXECUTE statement, listed alphabetically in this chapter.)

If the optional print-list is absent, only a carriage return and line feed will be output. The print-list may consist of a single expression, or a series of expressions separated by commas. (Commas are used to denote output formatting; refer to the next section on the PRINT statement). The expressions may be any legal BASIC expressions. The following statement, for example, will print the current value of the expression X+Y:

```
PRINT X+Y
```

The PRINT ON statement (i.e., with the ON option) is used, when PRINTER ON is in effect, to output the print-list items to a numbered print file. This is usually done when building several reports at the same time, each having a different number. The expression following ON indicates the print file number, which may be from 0 to 254 (selected arbitrarily by the program). Consider the following example:

```
PRINT ON 1 A,B,C,D
PRINT ON 2 E,F,G,H
PRINT ON 3 X,Y,Z
```

These statements will generate 3 separate output listings, one containing A, B, C, and D values, one containing E, F, G, and H values, and the third containing X, Y and Z values.

```
PRINT ON 1 A,B,C,D
PRINT ON 2 E,F,G,H
PRINT ON 2 X,Y,Z
```

These statements will generate 2 separate output listings, one containing A, B, C, and D values, and the second containing E, F, G, X, Y, and Z values.

When the ON expression is omitted, print file zero is used.

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.

When PRINTER OFF is in effect, both PRINT ON and PRINT operate identically, i.e., all output is to the terminal. The contents of all print files used by the program, including print file zero, will be output to the printer in sequence when a PRINTER CLOSE statement is given or on termination of the program.

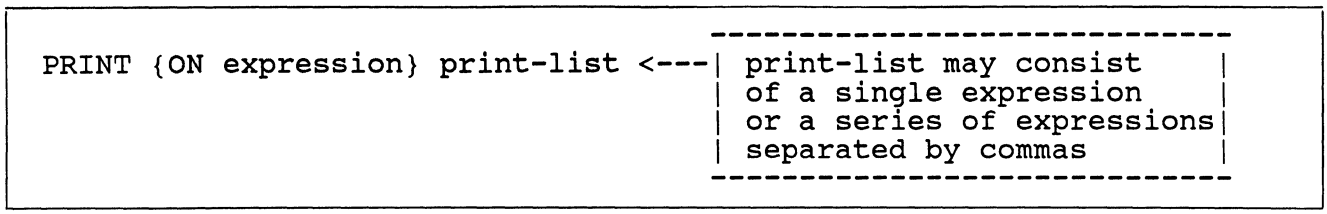


Figure A. General form of PRINT Statement

CORRECT USE	EXPLANATION
PRINT X	Causes the value of X to be output to the terminal (with no PRINTER ON in effect).
PRINTER ON PRINT X	Causes the value of X to be output to print file 0.
PRINTER ON PRINT ON 24 X	Causes the value of X to be output to print file 24.
N=50 PRINT ON N X,Y,Z	Outputs print-list to print file 50.
PRINTER ON PRINT ON 15 "100" PRINT ON 40 "100"	Causes the value 100 to be copied to both print file 15 and print file 40.
PRINTER ON PRINT A PRINT B	Print file 0 will contain the values of A and B.
PRINTER ON PRINT ON 10 F1,F2,F3 PRINT ON 20 M,N,P PRINT ON 10 F4,F5,F6	Print file 10 will contain the values of F1 through F6; print file 20 will contain the values M, N and P.

Figure B. Examples of Correct Usage of PRINT Statement

PRINT
Statement
(Output Formatting)

The print-list of the PRINT statement may specify tabulation or concatenation when printing multiple items, and carriage control at the end of a print line.

The general form of multiple print-list expressions is:

```
expression,expression,expression ...
```

The expression(s) may be any valid BASIC expressions.

Output values may be aligned at tab positions across the output page by using commas to separate the print-list expressions. Tab positions are pre-set at every 18 character positions. Consider the following example:

```
PRINT (50*3)+2, A, "END"
```

Assuming that the current value of A is 37, this statement will print the values across the output page as follows:

```
152                37                END
```

Normally, after the entire print-list has been printed, a carriage return and line feed will be executed. The <CR> and 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 will be printed on the same line at the very next character position. For example, these statements:

```
PRINT A:B,C,D:
PRINT E,F,G
```

will produce exactly the same output as this statement:

```
PRINT A:B,C,D:E,F,G
```

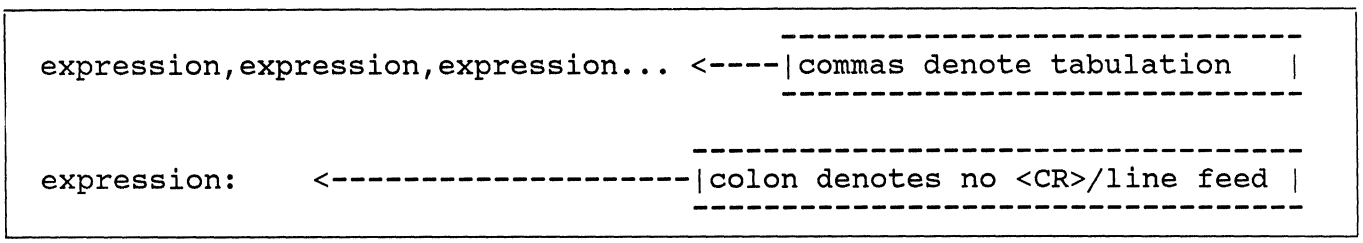


Figure A. Print-List Formats for PRINT Statement

CORRECT USE	EXPLANATION
PRINT A:B: PRINT C:D: PRINT E:F	Prints the current values of A, B, C, D, E, and F contiguously across the output page, each value concatenated to the next.
PRINT A=1	Prints 1 if "A=1" is true; prints 0 otherwise.
PRINT A*100,Z	Prints the value of A*100 starting at column position 1; prints the value of Z on the same line starting at column position 18 (i.e., 1st tab position).
PRINT	Prints an empty (blank) line.
PRINT "INPUT":	Prints the text "INPUT" and does not execute a carriage return or line feed.
PRINT " ", B	Prints the value of B starting at column position 18 (i.e., 1st tab position).

Figure B. Examples of Correct Usage of PRINT Statement Formatting

PRINTER
Statement

The PRINTER statement selects either the user's terminal or the line printer for subsequent program output.

The general formats of the PRINTER statement are:

```
PRINTER ON
PRINTER OFF
PRINTER CLOSE
```

The PRINTER ON statement directs program output data specified by subsequent PRINT, HEADING, FOOTING, or PAGE statements to be output to the line printer (or other destination as specified by the SP-ASSIGN command). The PRINTER OFF statement directs subsequent program output to the user's terminal.

Once executed, a PRINTER ON or PRINTER OFF statement will remain in effect until a new PRINTER ON or PRINTER OFF statement is executed. If a PRINTER ON statement has not been executed, all output will be 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 line printer (unless immediate printing is forced via an SP-ASSIGN command with the I option, as described in the ULTIMATE System Commands manual). 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 line printer prior to program termination. The PRINTER CLOSE statement will cause all data currently stored in the intermediate buffer area to be immediately printed.

When a PRINTER OFF statement has been issued, subsequent output data is always printed at the user's terminal immediately upon execution of the PRINT, HEADING, FOOTING, or PAGE statements (i.e., the PRINTER CLOSE statement applies only to output data directed to the line printer).

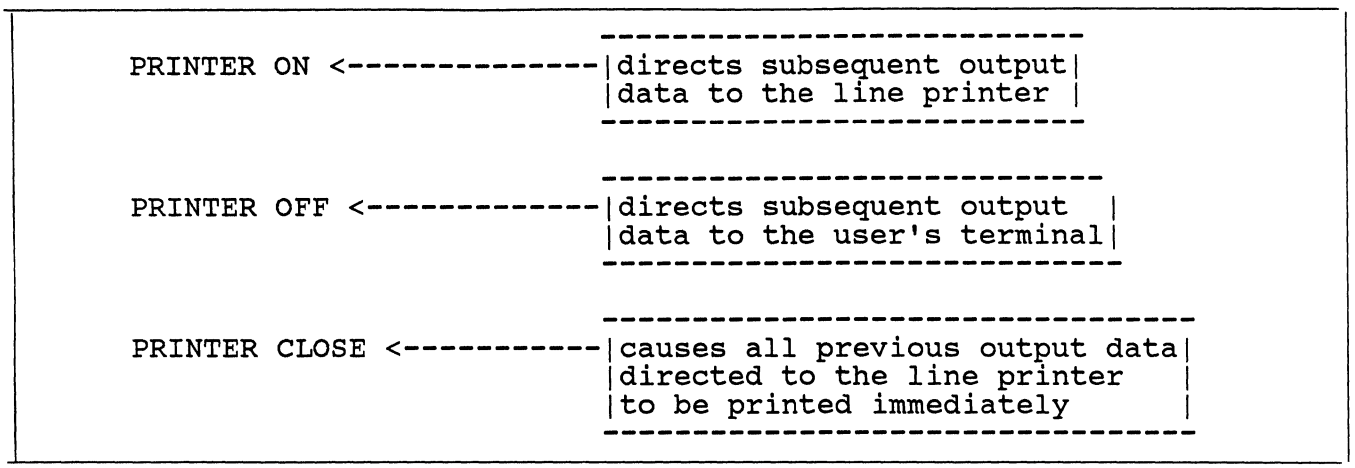


Figure A. General Form of PRINTER Statements

CORRECT USE	EXPLANATION
PRINTER ON PRINT A PRINTER OFF PRINT B	Causes the value of variable B to be immediately printed at the user's terminal, and the value of variable A to be printed on the line printer when the program is finished executing.
PRINTER ON PRINT A PRINTER CLOSE PRINTER OFF PRINT B	Causes the value of variable A to be immediately printed on the line printer, and thereafter causes the value of variable B to be printed at the user's terminal.
PRINTER ON PRINT A PRINTER OFF PRINT B PRINTER CLOSE	Causes the value of variable B to be immediately printed at the user's terminal, and thereafter causes the value of variable A to be printed on the line printer.

Figure B. Examples of Correct Usage of PRINTER Statements

PRINTERR
Statement

The PRINTERR statement prints a specified error message on the terminal screen.

The general form of the PRINTERR statement is:

```
PRINTERR error-text
```

The PRINTERR statement is designed as a support function for the INPUT statement. It allows a program to signal an operator with an error message relating to the operator's input.

The message is specified by the error-text, which may be any valid expression, including a literal string enclosed in quotation marks.

PRINTERR prints the error-text at the bottom line on the terminal screen.

The PRINTERR statement sets a flag so that the next time an INPUT statement is executed the bottom line will be blanked out. It also clears the type-ahead buffer on systems with the type-ahead feature.

```
PRINTERR 'error-text'
```

Figure A. General Form of PRINTERR Statement

CORRECT USE	EXPLANATION
PRINTERR "BAD INPUT"	Prints BAD INPUT on the bottom of the screen and sets the flag to clear that line (the message line) on next INPUT statement.

Figure B. Example of Correct Usage of PRINTERR 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.

The general form of the PROCREAD statement is:

PROCREAD variable {THEN statements} {ELSE statements}

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 variable. 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 variable retains its original value.

For more information on PROC, please refer to the ULTIMATE PROC Reference Manual.

A THEN clause and an 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 variable ELSE statement(s)
```

Figure A. General Form of the PROCREAD Statement

CORRECT USE	EXPLANATION

PROCREAD ITEM ELSE
  PRINT 'Not executed 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 statement-label 100. If the program is executed from PROC, then variable ITEM is printed.

PROCREAD ITEM ELSE ITEM='' PRINT ITEM	'ITEM' is set to null if program was not executed from PROC. The contents of ITEM will always be printed.

PROCREAD ITEM ELSE
  PRINT 'ITEM not found'
  STOP
END
FOR X=1 TO 10
  PRINT ITEM<X>
NEXT X
	If the program was executed from PROC, ITEM 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.

Figure B. Examples of Correct Usage of PROCREAD Statement

PROCWRITE
Statement

The PROCWRITE statement allows programs executed from PROC to write to the primary input buffer.

The general form of the PROCWRITE statement is:

PROCWRITE expression

The PROCWRITE command writes the string value of the expression to the PROC primary input buffer. The expression is treated as a dynamic array, and each attribute becomes one parameter in the primary input buffer. This statement will be ignored if the program was not executed from a PROC.

```
PROCWRITE expression
```

Figure A. General Form of the PROCWRITE Statements

CORRECT USE	EXPLANATION
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.

Figure B. Examples of Correct Usage of PROCWRITE Statement

PROGRAM
Statement

The PROGRAM statement may be used to indicate the name of a program.

The general form of the PROGRAM statement is:

```
PROGRAM {name}
```

The optional name may be used to indicate the name of a program, but this is ignored by the compiler. A program is invoked by specifying the item-id of the program item, or the item-id of a catalog pointer.

The PROGRAM statement is not required in any program except compile-and-go programs in an account's Master Dictionary. If a PROGRAM statement is used, it must be the first statement in a program.

External subroutines cannot contain PROGRAM statements since the first statement in an external subroutine must be a SUBROUTINE statement.

```
PROGRAM {name}
```

Figure A. General Form of PROGRAM Statement

CORRECT USE	EXPLANATION
PROGRAM	Indicates start of main-line program.
PROGRAM MYPROG	Indicates start of main-line program MYPROG.

Figure B. Examples of Correct Usage of PROGRAM Statement

PROMPT
Statement

The PROMPT statement is used to select the "prompt character" which is printed at the terminal to prompt the user for input. Any character may be selected.

The general form of the PROMPT statement is:

```
PROMPT expression
```

The value of the expression becomes the prompt character. For example:

```
PROMPT ":"
```

This statement selects the character ":" as the prompt character for subsequent INPUT statements. If the string value of the expression consists of more than one character, the first (leftmost) character will be used. For example:

```
PROMPT "ABC"
```

This statement selects the character "A" as the prompt character.

If the string value of the expression is the null string ('') then no prompt character will be used.

When a PROMPT statement has been executed, it will remain in effect until another PROMPT statement is executed. If a PROMPT statement is not executed, a question mark (?) will be the prompt character.

NOTE: The PROMPT statement is used with the INPUT statement. (For additional information, please refer to the INPUT statement, listed alphabetically in this chapter.)

```
PROMPT expression
```

Figure A. General Form of PROMPT Statement

CORRECT USE	EXPLANATION
PROMPT "@"	Specifies that the character @ will be used as a prompt character for subsequent INPUT statements.
PROMPT A	Specifies that the current value of A will be used as a prompt character.

Figure B. Examples of Correct Usage of PROMPT Statement

PUT
Statement

The PUT statement places a system message into the output of a program. The message may also be passed back to a calling program.

The general form of the PUT statement is:

```
PUT (MSG.) expression1, {expression2, ...}
```

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 pre-defined variable 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. A message identifier is the item-id of an item in the system ERRMSG file.

In the PUT statement, expression1 specifies the system message identifier (ERRMSG item-id); expression2 and any following expressions are the parameters associated with the message, if any.

System messages are normally formatted and displayed on the user's terminal, or on the printer if a PRINTER ON statement has been executed. If a program is invoked by another program (the "calling" program) via the EXECUTE statement, or by a PROC, messages are also copied to a buffer area where they can be later inspected by the calling program or PROC.

A calling program can retrieve the generated messages using the GET statement. GET normally retrieves messages in the same order that they are generated by PUT, STOP, and ABORT statements. For more information on the GET statement, please refer to the GET statement, listed alphabetically in this chapter.

```
PUT (MSG.) expression1, {expression2, ...}
```

Figure A. General Form of PUT Statement

CORRECT USE	EXPLANATION
PUT (MSG.) 415	Displays system message 415 (without parameters) and copies message-id to MSG. output buffer.
PUT 201, FILENAME	Displays message 201 with one parameter: the string value of variable FILENAME; also copies 201 and FILENAME value to MSG. output buffer.

Figure B. Example of Correct Usage of PUT Statement

PWR
Function

The PWR trigonometric function raises a number to a specified power.

The general form of the POWER function is:

PWR(base,exponent)

The base and exponent are expressions with a numeric value; base specifies the value to be raised to a power and exponent specifies the value of the power. If exponent is zero, the function will return the value of one (1). If the base is zero and exponent is any number other than zero, the function will return 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 will return unpredictable numbers.

Note: Another way to express the PWR function is X^Y, where X is raised to the Y power.

IMPORTANT: 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 B contains two subroutines recommended for this task.

In the following summary M is used to denote the largest allowable number in BASIC, which is 14,073,748,835.5327 with PRECISION 4.

FUNCTION	RANGE	DESCRIPTION
COS(X)	-M <= X <= M	Returns the cosine of an angle of X degrees.
SIN(X)	-M <= X <= M	Returns the sine of an angle of X degrees.
TAN(X)	-M <= X <= M	Returns the tangent of an angle of X degrees.
LN(X)	0 <= X <= M	Returns the natural (base e) logarithm of the expression X.
EXP(X)	-M <= RESULT <= M	Raises the number 'e' (2.7183) to the value of X.
PWR(X,Y)	-M <= RESULT <= M	Raises the first expression to the power denoted by the second expression.

Figure A. Summary of Trigonometric Functions

```
      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=INT(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
      J=POWER
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 B. Subroutines for Extended Arithmetic Power Function

READ
Statement

The READ statement reads a file item and assigns its value to a variable.

The general form of the READ statement is:

```
READ var FROM {file-var,} item-id {ON ERROR statements}
     {THEN stmt} {ELSE stmt}
```

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

The READ statement reads the file item specified by the item-id and assigns its string value to var. The file-var (file variable) is optional. If file-var is present, the item will be read from the file previously assigned to that variable via an OPEN statement. If file-var is omitted, then the internal default file variable is used (thus specifying the file most recently opened without a file variable).

The statement(s) after ON ERROR, if present, are executed only if the file (1) is a remote file (accessed via UltiNet) and (2) cannot be read due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 ON ERROR clause 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.

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 read due to network errors, the program may terminate with an error message if no ON ERROR clause is present.

If the item-id specifies the name of an item which does not exist, then the ELSE statement(s), if any, will be executed; the value of var will remain unchanged. If the read is successful, the THEN statement(s), if any, will be executed.

The statements in the THEN or ELSE clauses may appear on one line separated by semicolons, or on multiple lines terminated by an END, as in the multi-line IF statement.

Consider the following example:

```
READ X1 FROM W,"TEMP" ELSE PRINT "NON-EXISTENT"; STOP
```

This statement will read the item named TEMP from the file opened and asigned to variable W, and will assign its string value to variable X1; program control will then pass to the next sequential statement in the program. If the file item TEMP does not exist, the message "NON-EXISTENT" will be printed and the program will terminate.

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 READ statement. (Refer to the appendix describing run-time error messages.)

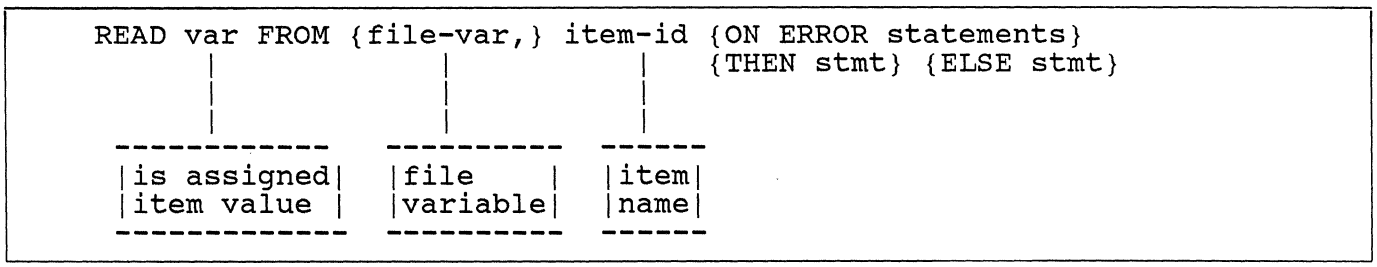


Figure A. General Form of READ Statement

CORRECT USE	EXPLANATION
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="TEST" B="1" READ X FROM C,(A CAT B) ELSE STOP	Reads item TEST1 from the file opened and assigned to file variable C, and assigns its value to variable X. Program terminates if TEST1 does not exist.
READ Z FROM "Q" ELSE PRINT X; STOP	Reads item Q from the file opened without a file variable and assigns its value to variable Z. Prints value of X and terminates program if Q does not exist.
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.

Figure B. Examples of Correct Usage of READ Statement

READNEXT
Statement

The READNEXT statement reads the next item-id or other data element from a selected list. If multiple select lists are present, a select variable specifies the list to use.

The general form of the READNEXT statement is:

```
READNEXT var {,var}{FROM select-var} {ON ERROR statements}
         {THEN stmt} {ELSE stmt}
```

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

The first variable var is assigned the string value of the next select list element (typically an item-id). If present, the second variable var is assigned the value number associated with this element, or zero if no value number is present. The optional select-var specifies a particular select list when several such lists are available to a program. If select-var is absent, the internal default select variable will be used. The statements in the THEN clause, if present, are then executed.

If the select list has been exhausted, or if no selection has been performed, the ELSE statements, if any, will be executed. The statements in the THEN or ELSE clauses may be placed on one line separated by semicolons, or may be placed on multiple lines terminated by an END, as in the multiple line IF statement.

The statement(s) after ON ERROR, if present, are executed only if a file is being read and it (1) is a remote file (accessed via UltiNet) and (2) cannot be accessed due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 accessing local files.

The ON ERROR clause 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.

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 accessed due to network errors,

the program may terminate with an error message if no ON ERROR clause is present.

A select list is a list of data generated by a BASIC SELECT statement, or by a verb such as SELECT, SSELECT, QSELECT, or GET-LIST. When a select-type command 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. For example:

```
>SSELECT NAMEFILE BY LASTNAME

100 items selected.
>RUN BP COUNTER
```

Here the Recall SSELECT command is used to generate a select list of 100 item-ids, corresponding to items in file NAMEFILE, sorted by attribute LASTNAME. The BASIC program COUNTER can then retrieve these item-ids in order, using READNEXT:

```
LOOP
  READNEXT ID ELSE STOP
  READ ITEM FROM ID ELSE GOTO ERR
  PRINT ITEM<3>
REPEAT
```

A select list may also be generated by a command specified in an EXECUTE statement. If not redirected, the list will be assigned to the internal default select variable of the program performing the EXECUTE. Alternatively, the select list may be redirected via the pre-defined variable SELECT., setting up a variable as a select variable for use by READNEXT. For example:

```
...
EXECUTE "SSELECT NAMEFILE BY LASTNAME",
     //SELECT.>LIST
LOOP
  READNEXT ID FROM LIST ELSE STOP
  READ ITEM FROM ID ELSE GOTO ERR
  PRINT ITEM<3>
REPEAT
```

Finally, 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. 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-type verbs (SELECT, SSELECT, QSELECT, GET-LIST), please refer to the ULTIMATE Recall manual.

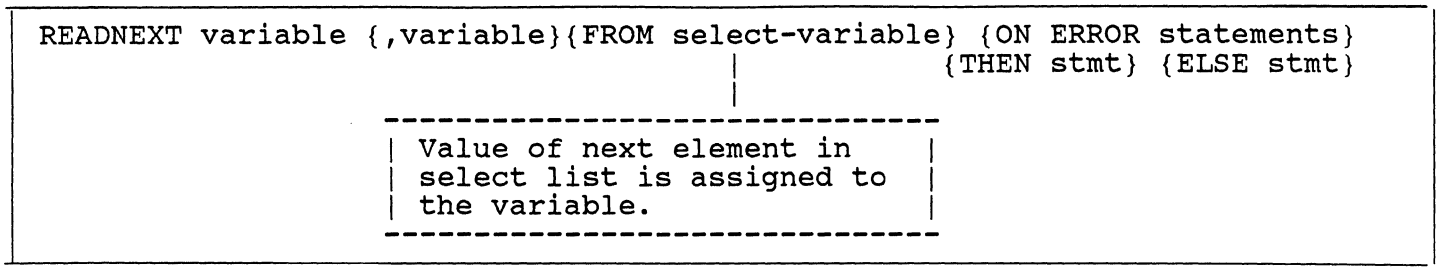


Figure A. General Form of READNEXT Statements

CORRECT USE	EXPLANATION
READNEXT A FROM X ELSE STOP	Specifies the list selected and assigned to the select-variable X. Assigns the value of that list's next element to variable A. If select list is exhausted, program will terminate.
READNEXT X2 ELSE PRINT "UNABLE" GOTO 50 END	Specifies the last list selected without a select-variable. Assigns the value of the next element to variable X2. If unable to read, "UNABLE" is printed and control transfers to statement 50.
FOR X=1 TO 10 READNEXT B(X) ELSE STOP NEXT X	Reads next ten item-ids or other data and assigns values to array elements B(1) through B(10).
SSELECT AFILE BY EXP,INV# READNEXT ID,VN	Uses select list to read next item-id and assigns value number.
READNEXT A FROM X ON ERROR ERRNUM=SYSTEM(0) GOSUB PROCESSERR GOTO TOP END ELSE STOP	Reads as first example above, or retrieves error number and performs local subroutine on UltiNet error number.

Figure B. Examples of Correct Usage of READNEXT Statements

READT
Statement

The READT statement reads a 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 executed at the TCL level. A tape unit must be previously attached; if it is subsequently set off line, the system detects the condition and allows the user to correct it and proceed.

The general form of the READT statement is:

```
READT variable {THEN statements} {ELSE statements}
```

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

The READT statement reads the next record from the "current" magnetic tape unit and assigns the string value to the specified variable. The THEN statements, if any, are then executed.

If this statement is the first tape instruction (READT, REWIND, WEOF, or WRITET) in the BASIC program, the tape unit must have previously been attached. If the tape unit has not been attached, or if an End-of-File (EOF) mark is read, then the ELSE statements, if any, will be executed, and the system function SYSTEM(0) will return 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.)

For example:

```
READT X ELSE PRINT "CANNOT READ"; STOP
```

or

```
READT X ELSE
PRINT "END OF TAPE OR TAPE NOT ATTACHED"
STOP
END
```

The next tape record is read and assigned to variable X. If an EOF is read (or no tape unit is attached), then "CANNOT READ" is printed and the program terminates.

NOTE: Refer to the SYSTEM Function for an alternative to printing the error messages.

If, however, the tape drive is adversely 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. Thus, the ELSE statements are not executed in either case, and the BASIC program has no way to detect such adverse action.

IMPORTANT: 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.). 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 the user to (C)ontinue, it is not guaranteed that valid data is then read or written.

Note: The READT statement is used in conjunction with the WRITET, WEOF, and REWIND statements. (For additional information, see each statement listed alphabetically in this chapter.) See the ULTIMATE System Commands manual for more information on the T-ATT command.

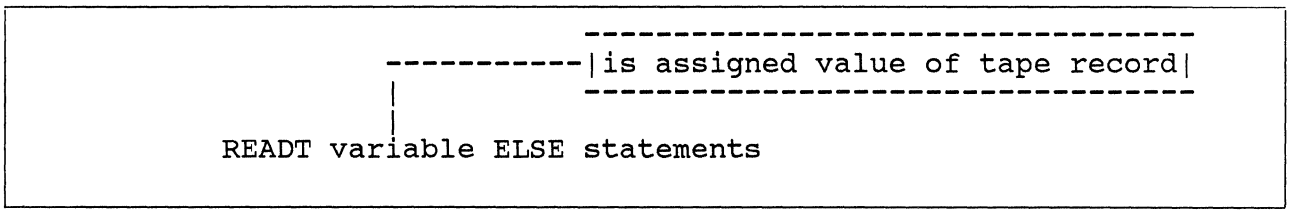


Figure A. General Form of READT Statement

CORRECT USE	EXPLANATION
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 not attached), then "NO" is printed and control passes to statement 5.

Figure B. Examples of Correct Usage of READT Statement

READU
Statement

The READU statement provides the facility to lock a disk file item prior to updating it. This can be used to prevent updating an item by two or more programs simultaneously while still allowing multiple programs to access the file.

The general form of the READU statement is:

```
READU var FROM {file-var,} item-id {ON ERROR statments}
      {LOCKED stmt} {THEN stmt} {ELSE stmt}
```

The READU statement functions like the READ statement, but additionally locks the item lock associated with the item to be accessed. If the item is currently locked by another BASIC program, the statement will 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.

Item locks are assigned based on (1) the group of the disk file which contains (or would contain) an item and (2) 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 the LOCKED clause is present, the statements specify an action to take if the READU statement is unable to lock the item because another program has already locked it.

If an item is currently unlocked, setting a corresponding item lock will prevent access to the item, and 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, will be 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, is updated by a WRITE, WRITEV, or MATWRITE 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 numgber 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, it will be 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, which does not pay attention to item locks.

```
READU var FROM {file-var,} item-id {ON ERROR statements}
        {LOCKED stmt} {THEN stmt} {ELSE stmt}
```

Figure A. General Form of READU Statement

CORRECT USE	EXPLANATION
READU ITEM FROM INV, S5 ELSE GOSUB 4 END	Lock item S5. Read S5 to variable ITEM or, if S5 is non-existent, execute the ELSE clause; in either case the item remains locked until it is updated, or until it is unlocked by a RELEASE statement.
READU ITEM FROM INV, "30" LOCKED GOTO 500 ELSE NULL	Lock item "30". If the item is already locked, go to label 500.
READU ITEM FROM INV, "30" ON ERROR GOTO PROCESSERR ELSE NULL	Read and lock as above, or branch to local subroutine to process UltiNet error number.

Figure B. Examples of Correct Usage of READU Statement

READV
Statement

The READV statement is used to read a single attribute value from an item in a file.

The general form of the READV statement is:

```
READV var FROM {file-var,} item-id, attr#
      {ON ERROR stmt} {THEN stmt} {ELSE stmt}
```

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

The READV statement reads the attribute specified in the attr# (attribute number expression) from the item specified by the item-id expression. The string value of the attribute is assigned to variable var.

The file-var is optional and specifies the file variable; if it is used, the attribute will be read from the file previously assigned to that variable via an OPEN statement. If file-var is omitted, then the internal default file variable is used (thus specifying the file most recently opened without a file variable).

The statement(s) after ON ERROR, if present, are executed only if the file (1) is a remote file (accessed via UltiNet) and (2) cannot be read due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 read.

The ON ERROR clause 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.

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 read due to network errors, the program may terminate with an error message if no ON ERROR clause is present.

If a non-existent item is specified, the ELSE statements, if any, will be executed. If the read is successful, the THEN statements, if any, will be executed. The statements in the THEN or ELSE clauses may be placed on one line separated by

semicolons, or may be placed on multiple lines terminated by END, as in the multi-line IF statement.

Consider the following example:

```
READV A FROM F,"XYZ", 3 ELSE STOP
```

This statement reads the third attribute of item XYZ (in the file opened and assigned to variable F) and assigns its value to variable A. If item XYZ does not exist, the program terminates.

The BASIC program will abort with an appropiate error message if the specified file has not been opened prior to the execution of the READV statement.

NOTE: 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.

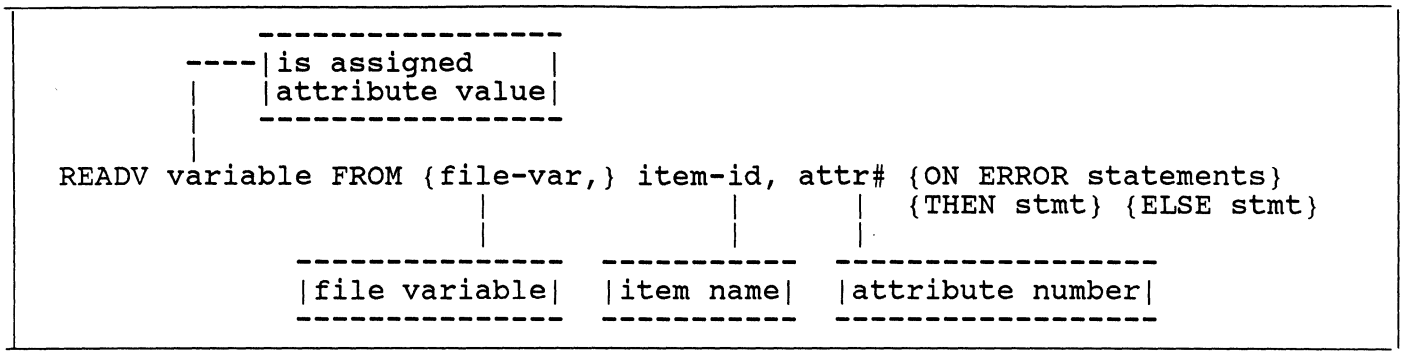


Figure A. General Form of READV Statement

CORRECT USE	EXPLANATION
READV X FROM A, "TEST", 5 ELSE PRINT ERR GOTO 70 END	Reads 5th 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.

Figure B. Examples of Correct Usage of READV Statements

READVU
Statement

The READUVU statement provides the facility to lock a disk file item prior to updating it. This can be used to prevent updating an item by two or more programs simultaneously while still allowing multiple program access to the file.

The general form of the READVU statement is:

```
READVU var FROM {file-var,} item-id, attr# {ON ERROR stmt}
       {LOCKED stmt} {THEN stmt} {ELSE stmt}
```

The READVU statement functions like the READV statement, but additionally locks the item lock associated with the item to be accessed. If the item is currently locked by another BASIC program, the statement will 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.

Item locks are assigned based on (1) the group of the disk file which contains (or would contain) an item and (2) 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 the LOCKED clause is present, the statements specify an action to take if the READVU statement is unable to lock the item because another program has already locked it.

If an item is currently unlocked, setting a corresponding item lock will prevent access to the item, and 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, will be 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, is updated by a WRITE, WRITEV, or MATWRITE 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 numgber 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, it will be 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, which does not pay attention to item locks.

```
READVU var FROM {file-var,} item-id, attr# {ON ERROR stmt} {LOCKED stmt}
                          {THEN stmt} {ELSE stmt}
```

Figure A. General Form of READVU Statement

CORRECT USE	EXPLANATION
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 example above, or branch to local subroutine to process UltiNet error number.

Figure B. Example of Correct Usage of READVU Statement

RELEASE
Statement

The RELEASE statement unlocks specified items or all items locked by the program.

The general form of the RELEASE statement is:

RELEASE {{file-var,} item-id} {ON ERROR statements}

The RELEASE statement unlocks the item lock locked, or potentially locked, by the expression item-id. If the file-var is present, the file containing the item referenced by item-id is the one previously assigned to that variable via on OPEN statement. If file-var is absent, then the internal default file variable is used (thus specifying the file most recently opened without a file variable).

If the RELEASE statement is used without a file-var or item-id all items which have been locked by the program will be unlocked. This form of the statement is:

RELEASE

The statement(s) after ON ERROR, if present, are executed only if the file (1) is a remote file (accessed via UltiNet) and (2) cannot be accessed due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 accessing local files.

The ON ERROR clause 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.

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 accessed due to network errors, the program may terminate with an error message if no ON ERROR clause is present.

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.

NOTE: The RELEASE statement is used in conjunction with the

READU, READVU, and MATREADU statements. (Please refer to these statements, listed alphabetically in this chapter.)

```
RELEASE {{file-var,} item-id} {ON ERROR statements}
```

Figure A. General Form of RELEASE Statement

CORRECT USAGE	EXPLANATION
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 ITEM 3'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.

Figure B. Examples of Correct Usage of RELEASE Statement

REM
Function

The REM function returns the remainder of one number divided by another.

The general form of REM function is:

```
REM (dividend,divisor)
```

Both dividend and divisor are expressions. The REM function returns the remainder of the (integer) division of dividend by divisor.

Note: The REM function operates the same as the MOD (modulo) function. (Please refer to the MOD function, listed alphabetically in this chapter.)

```
REM (dividend,divisor)
```

Figure A. General Form of REM Function

CORRECT USE	EXPLANATION
Q = REM(11,3)	Assigns the value 2 to variable Q.

Figure B. Examples of Correct Usage of REM Function

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.

A Remark statement can be specified in one of three ways: by typing the characters "REM", by the asterisk character (*), or by the exclamation point (!). Thus, there are three general forms of the Remark statement:

```
REM text ...
! text ...
* text ...
```

REM, !, or * must be placed at the beginning of the statement, but may appear anywhere on a line (e.g., 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.

For example:

```
REM THE TEXT FOLLOWING THESE STATEMENTS
!   DOES NOT AFFECT
*   PROGRAM EXECUTION
```

Note that there are extra blank spaces in some of the statements above. These blank spaces appearing in the program line (which are not part of a data item) will be ignored. Thus, blanks may be used freely within the program to enhance the appearance and readability of a program and its comments.

```
REM text              ! text              * text
```

Figure A. General Forms of Remark Statement

```
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
```

Figure B. Sample Program With Remark Statements

REPEAT
Statement

The REPEAT statement is the last statement in a LOOP statement sequence.

The general form of the REPEAT statement is:

```
REPEAT
```

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.

The general forms of the REPLACE function are:

```
REPLACE(array-var,attr#,value#,subval#,expr)

REPLACE(array-var,attr# {,value# {,subval#}}; expr)
```

The value of the expression array-var specifies the dynamic array in which to replace data. The values of attr#, value#, and subval# determine whether the data is an attribute, a value, or a subvalue. Attr# specifies an attribute, value# specifies a value, and subval# specifies a subvalue. If value# and subval# both have a value of 0 (or are both absent), then an entire attribute is replaced. If subval# only has a value of 0 (or is absent), then a value is replaced. If attr#, value#, and subval# are all non-zero, then a subvalue is replaced. The value of expr specifies the replacement value. Data may be inserted following the last attribute, value, or subvalue by specifying an index of -1 (see below).

The following examples show two ways to express the same replacement function. They replace attribute 4 of item NAME in file INVENTORY with the string value "ABC":

Using the First Form

```
OPEN 'INVENTORY' ELSE STOP
READ X FROM 'NAME' ELSE STOP
X = REPLACE(X,4,0,0,'ABC')
WRITE X ON 'NAME'
```

Using the Second Form

```
OPEN 'INVENTORY' ELSE STOP
READ X FROM 'NAME' ELSE STOP
X = REPLACE(X,4;'ABC')
WRITE X ON 'NAME'
```

Note that in the second form, trailing zero subvalue or value numbers are not required and that a semicolon separates the attribute, value, and subvalue numbers from the new data.

If the last index specified (attr#, value#, or subval#) has a value of -1, the new data is inserted after the last attribute, last value, or last subvalue of the dynamic array. In this case, the REPLACE function is identical to an INSERT function with the same parameters. For example:

Using the First Form

```
OPEN 'XYZ' ELSE STOP
READ B FROM 'ABC' ELSE STOP
Y=REPLACE(B,3,-1,0,'NEW VALUE')
WRITE Y ON 'ABC'
```

Using the Second Form

```
OPEN 'XYZ' ELSE STOP
READ B FROM 'ABC' ELSE STOP
Y=REPLACE(B,3,-1;'NEW VALUE')
WRITE Y ON 'ABC'
```

These statements insert the string value "NEW VALUE" after the last value of attribute 3 of item ABC in file XYZ.

Note: An assignment statement may 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,2,0,0,6). For more information, please see the = (Assignment) statement, listed alphabetically in this chapter.

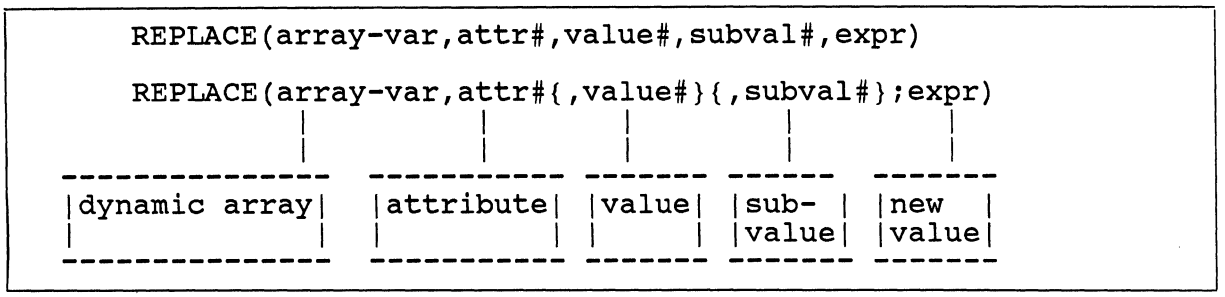


Figure A. General Form of REPLACE Function

CORRECT USE	EXPLANATION
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 the resultant dynamic array to Y.
VALUE="TEST STRING" DA=REPLACE(DA,4,3,2,VALUE)	Replaces subvalue 2 of value 3 of attribute 4 in dynamic array DA with the string value "TEST STRING".
X="ABC123" Y=REPLACE(Y,1,1,-1,X)	Inserts the value "ABC123" after the last subvalue of value 1 of attribute 1 in dynamic array Y.
A=REPLACE(B,2,3,0,"XXX")	Replaces value 3 of attribute 2 of dynamic array B with the value "XXX", and assigns the resultant dynamic array to A.

Figure B. Examples of Correct Usage of REPLACE Function

RETURN (TO)
Statement

The RETURN and RETURN TO statements return control from a subroutine.

The general forms of the RETURN statement are:

```
RETURN

RETURN TO statement-label
```

The RETURN statement will transfer control from a subroutine back to the statement immediately following the GOSUB statement (for local subroutines) or CALL statement (for an external subroutine) which invoked the subroutine. The RETURN TO statement returns control from a subroutine to the statement within the local BASIC program having the specified statement-label. Sample RETURN and RETURN TO statements are included in Figure B.

The statements in a subroutine may be any BASIC statements, including other GOSUB and CALL statements. To insure proper flow of control, each subroutine must return to the calling program by using a RETURN (or RETURN TO) statement, not a GOTO statement. The user should also insure that a local subroutine cannot be executed by any flow of control other than through the execution of a GOSUB statement.

If the RETURN TO statement refers to a statement-label which is not present in the current program, an error message will be printed at compile time (refer to APPENDIX A - BASIC COMPILER ERROR MESSAGES).

Consider the set of statements shown in Figure B. Upon execution of statement 10, control will transfer to statement 30 as illustrated in the left-hand side of the figure. The statements within the subroutine will be executed and statement 40 will then return control to statement 15. Execution will then proceed sequentially to statement 20, whereby control will again be transferred to the subroutine as shown in the right-hand side of the figure. The conditional RETURN TO path is taken instead of the normal RETURN if the logical variable ERROR is true (=1).

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.

The example in Figure C illustrates incorrect usage of the BASIC subroutine capability (i.e., the GOTO statement should not be used to transfer control to a subroutine).

```
RETURN

                                    ----------------------------------------
RETURN TO statement-label <---------|A statement must exist which has this|
                                    |statement-label.                      |
                                    ----------------------------------------
```

Figure A. General Form of RETURN and RETURN TO Statements

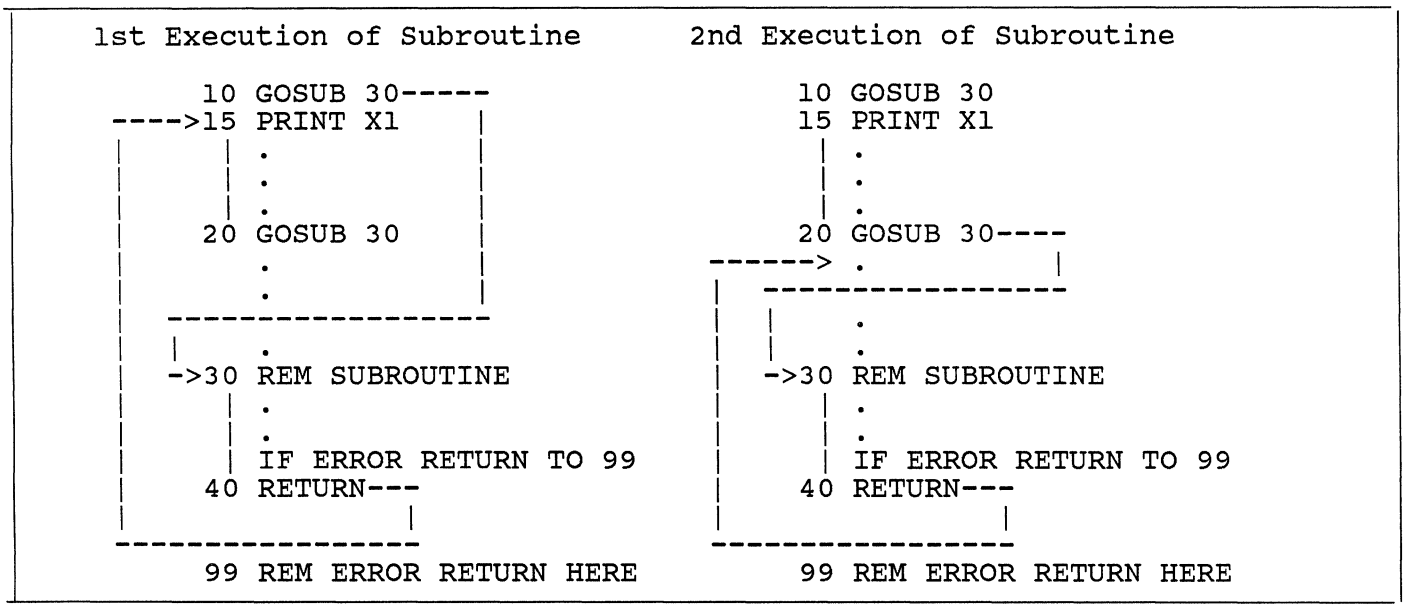


Figure B. Example of Correct Subroutine Usage

```
A=1
GOSUB 100
A=A+1
GOTO 110  <-------- INCORRECT: This statement transfers
100 PRINT A          control into the body of a subroutine,
110 B=A*D            causing the "RETURN" to produce an
PRINT B              error message: "RETURN executed with
RETURN               no GOSUB"
```

Figure C. Example of Incorrect Subroutine Usage

REWIND
Statement

The REWIND statement rewinds a magnetic tape unit. The tape unit is as specified by the most recent T-ATT command executed at the TCL level.

The general form of the REWIND statement is:

```
REWIND {THEN statements} {ELSE statements}
```

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

The REWIND statement rewinds the "current" magnetic tape unit to the Beginning-of-Tape (BOT). The THEN statements, if any, are then executed. If a previously attached tape unit is subsequently set off line, the system detects the condition and allows the user to correct it and proceed.

If this statement is the first tape instruction (READT, REWIND, WEOF, or WRITET) in the BASIC program, the tape unit must have been previously attached. If the tape unit has not been attached, then the ELSE statements, if any, will be executed, and the system function SYSTEM(0) will return 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, for an alternative to printing error messages.)

If, however, the tape drive is adversely 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 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.

IMPORTANT: 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.). 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 the user to (C)ontinue, it is not guaranteed that valid data is then read or written.

Note: The REWIND statement is used in conjunction with the READT, WRITET, and WEOF statements. (For additional information, see each statement listed alphabetically in this chapter.) See the System Commands manual for more information on the T-ATT command.

REWIND {THEN statements} {ELSE statements}

Figure A. General Form of REWIND Statement

CORRECT USE	EXPLANATION
REWIND ELSE STOP	Tape is rewound to BOT.

Figure B. Examples of Correct Usage of REWIND Statement

RND
Function

The RND function returns a random number.

The general form of the RND function is:

```
RND(expression)
```

The RND function generates an integer between 0 and the number specified by the expression minus one (inclusive). For example:

```
NUMBER = RND(201)
```

This statement generates a random number between 0 and 200 (inclusive), and assigns its numeric value to the variable NUMBER.

The value of expression must be a positive number no larger than 32,767. If a larger number is used, a value of 0 is returned by the function. The expression value is truncated if necessary (not rounded) to the nearest integer before being used as the upper limit in the random number range.

```
RND(expression)
```

Figure A. General Form of the RND Function

CORRECT USE	EXPLANATION
`Z = RND(11)`	Assigns a random number between 0 and 10 (inclusive) to variable Z.

`R = 100`
`Q = 50`
`B = RND(R+Q+1)`
	Assigns a random number between 0 and 150 (inclusive) to the variable B.

`Y = RND(ABS(051))`	Assigns a random number between 0 and 50 (inclusive) to the variable Y.

Figure B. Examples of Correct Usage of the RND Function

RQM
Statement

The RQM (release quantum) statement suspends program execution for one second.

The general form of the RQM statement is:

```
RQM
```

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. Thus, RQM may be used to cause pauses in program execution.

CORRECT USE	EXPLANATION

`* PROGRAM SEGMENT TO SOUND`
`* TERMINAL "BELL" FIVE TIMES.`
`*`
`BELL=CHAR(7)`
`FOR I=1 TO 5`
`PRINT BELL:`
`RQM`
`NEXT I`
`END`
	RQM statement causes "beeps" at one-second intervals.

Figure B. Example of Correct Usage of RQM Statement

SADD
Function

The SADD (string addition) function adds two string numbers and returns the result as a string number.

The general form of the SADD function is:

```
SADD (X,Y)
```

X and Y may be any valid numbers or string numbers of any magnitude and precision. For speed considerations, string numbers are preferable to standard numbers.

If either X or Y contains non-numeric data, an error message is generated; the result of the addition will be zero.

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.

```
SADD (X,Y)
```

Figure A. General Form of SADD Function

CORRECT USE	EXPLANATION
TOTAL=SADD(SUBTOT1,SUBTOT2)	Assigns sum of variables SUBTOT1 and SUBTOT2 to variable TOTAL.
PRINT (SADD(X,".004")	Prints sum of variable X and string constant (.004).
A=SADD("1.030476",B)	Assigns to variable A the sum of string constant (1.030476) and variable B.
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.

Figure B. Examples of Correct Usage of SADD Function

SCMP
Function

The SCMP (string compare) function compares two string numbers and returns a result of -1 (less than), 0 (equal), or 1 (greater than).

The general form of the SCMP function is:

```
SCMP (X,Y)
```

X and Y may be any valid numbers or string numbers. For speed considerations, string numbers are preferable to standard numbers.

If either X or Y contains non-numeric data, an error message is generated; the result of the comparison will be zero (0).

The result of the SCMP function is a number: -1, 0, or 1. If X is less than Y, the result is -1. If they are equal, the result is 0. If X is greater than Y, the result is 1. The function can be used in any expression where a number or string would be valid.

```
SCMP (X,Y)
```

Figure A. General Form of SCMP Function

CORRECT USE	EXPLANATION
`IF SCMP(FX,FY) = 0 THEN GOTO 100`	The result of the comparison determines whether program execution branches to statement 100 or continues in sequence.
`IF SCMP(FX,FY) < 0 THEN PRINT X:" IS LESS THAN ":Y`	The PRINT operation is executed only if the result of the IF statement is true (-1 was the result of the SCMP function).
`IF SCMP(FX,FY) > 0 THEN PRINT X:"IS GREATER THAN ":Y`	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.

Figure B. Examples of Correct Usage of SCMP Function

SDIV
Function

The SDIV (string division) function divides the first string number by the second and returns the result as a string number.

The general form of the SDIV function is:

```
SDIV (X,Y)
```

X and Y may be any valid numbers or string numbers. Although the two arguments may be of any magnitude and precision, the precision of the result will be limited to 14 significant digits. (See examples in Figure B.)

If either X or Y contains non-numeric data, an error message is generated; the result of the division will be zero. If Y is zero, an error message will state that division by zero is illegal; the result will be zero.

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.

```
SDIV (X,Y)
```

Figure A. General Form of SDIV Function

CORRECT USE	EXPLANATION
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") Y=SDIV("1","3") Y-SDIV("0.1","3")	Result is 3.3333333333333. Result is .33333333333333. Result is .033333333333333.

Figure B. Examples of Correct Usage of SDIV Function

SEEK
Statement

The SEEK statement allows a program to find data stored in either the ARG. or MSG. redirection variable. When used with ARG., it finds an argument in the argument list passed to the program. When used with MSG., it finds a system message resulting from the last EXECUTE statement.

The general forms of the SEEK statement are:

```
SEEK ( ARG. {,arg#} ) var {THEN stmt} {ELSE stmt}

SEEK ( MSG. {,arg#} ) var {THEN stmt} {ELSE stmt}
```

The SEEK statement is always used with one of two system predefined redirection variables: ARG. and MSG. ARG. refers to the list of arguments (if any) following the program name in the TCL command which invoked the program. For example:

```
RUN BP MYPROG ARG1 ARG2
```

This invokes program MYPROG, which can find the strings "ARG1" and "ARG2" using SEEK (ARG.) statements.

MSG., as used in the SEEK statement, refers to the list of input messages generated by the most recent EXECUTE statement.

The arg# may be any expression that evaluates to an integer that specifies the argument to be positioned to; if arg# is not present, the next argument on the argument list is positioned to. (If this is the first SEEK statement executed, the first argument on the list is used.) If an argument is present in the position specified, the internal pointer is set to that position and the THEN branch, if specified, is taken. If no argument is present in that position, the internal pointer is not changed and the ELSE branch, if specified, is taken.

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 MSG. redirection variable, please refer to the GET statement, listed alphabetically in this chapter.

```
SEEK ( ARG. {, arg#} ) {THEN stmt} {ELSE stmt}

SEEK ( MSG. {, arg#} ) {THEN stmt} {ELSE stmt}
```

Figure A. General Forms of the SEEK Statement

CORRECT USE	EXPLANATION
SEEK (ARG., 3) THEN GOSUB 10000 ELSE PRINT "NOT ENOUGH ARGS" STOP END	If the argument is found, the subroutine is executed; otherwise, the message is printed and program execution terminates.

Figure B. Example of Correct Usage of the SEEK Statement

SELECT
Statement

The SELECT statement builds a select list from a file or a dynamic array for use with the READNEXT statement. When a file is selected, the select list will be a list of items in the file. When a dynamic array is selected, the list will contain one list element from each attribute in the array.

The general form of the SELECT statement is:

SELECT {variable} {TO select-variable} {ON ERROR stmt}

If variable is present, it must be either a file variable previously initialized in an OPEN statement, or a variable whose current value is a dynamic array. 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.

If select-variable is present, the select list generated will be assigned to that variable. If select-variable is omitted, the list will be assigned to the program's internal default select variable. A select variable, in general, has meaning only in SELECT and READNEXT statements; its value outside of these statements is undefined.

Select lists are discussed in more detail under the READNEXT statement, listed alphabetically in this chapter. Also see the OPEN statement for more information on this statement.

The statement(s) after ON ERROR, if present, are executed only if a file is selected and it (1) is a remote file (accessed via UltiNet) and (2) cannot be accessed due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 accessing local files.

The ON ERROR clause 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.

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 accessed due to network errors, the program may terminate with an error message if no ON ERROR clause is present.

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, when executed from the TCL level. 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.

As an example, the following BASIC program will print the item-ids of all the items in the file named BP.

```
    OPEN 'BP' ELSE STOP
    SELECT
10  READNEXT ID ELSE STOP
    PRINT ID
    GOTO 10
```

Each of the six possible formats of the SELECT statement is illustrated below:

SELECT

Creates a select list of item-ids from the file most recently opened without a file variable.

SELECT file-variable

Creates a select list of item-ids from the file opened to the file-variable.

SELECT var

Creates a select list from the attributes of the item currently stored in the variable var.

SELECT TO select-variable

Creates a select list from the file most recently opened without a file variable; assigns the selected list to the select-variable.

SELECT file-variable TO select-variable

Creates a select list of item-ids from the file opened to the file-variable; assigns the selected list to the select-variable.

SELECT var TO select-variable

As above, except the list is created from the attributes of the item in var.

```
SELECT {variable} {TO select-variable} {ON ERROR stmt}
```

Figure A. General Form of SELECT Statement

CORRECT USE	EXPLANATION
SELECT	Builds 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.

Figure B. Examples of Correct Usage of SELECT Statements

SEQ
Function

The SEQ function converts an ASCII character to its corresponding numeric value.

The general form of the SEQ function is:

```
SEQ(expression)
```

The value of the expression is the ASCII character string to be converted. The first character of the string is converted to its corresponding numeric value.

The following example will print the number 49:

```
PRINT SEQ('1')
```

NOTE: For a complete list of ASCII codes, refer to the appendix of this manual.

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

```
SEQ(expression)
```

Figure A. General Form of SEQ Function

CORRECT USE	EXPLANATION
DIM C(50) S = 'THE GOOSE FLIES SOUTH' FOR I=1 TO LEN(STRING) C(I) = SEQ(S[I,1]) NEXT I	Encodes in vector C elements the decimal equivalents of individual characters of character string S.

Figure B. Examples of Correct Usage of SEQ Function

SIN
Function

The SIN trigonometric function returns the sine of an angle expressed in degrees.

The general form of the SIN function is:

SIN(expression)

The value of the expression specifies the number of degrees in the angle. The function generates the sine of the angle.

In the following summary M is used to denote the largest allowable number in BASIC, which is 14,073,748,835.5327 with PRECISION 4.

FUNCTION	RANGE	DESCRIPTION
COS(X)	-M <= X <= M	Returns the cosine of an angle of X degrees.
SIN(X)	-M <= X <= M	Returns the sine of an angle of X degrees.
TAN(X)	-M <= X <= M	Returns the tangent of an angle of X degrees.
LN(X)	0 <= X <= M	Returns the natural (base e) logarithm of the expression X.
EXP(X)	-M <= RESULT <= M	Raises the number 'e' (2.7183) to the value of X.
PWR(X,Y)	-M <= RESULT <= M	Raises the first expression to the power denoted by the second expression.

Figure A. Summary of Trigonometric Functions

SMUL
Function

The SMUL (string multiplication) function multiplies two string numbers and returns the result as a string number.

The general form of the SMUL function is:

```
SMUL (X,Y)
```

X and Y may be any valid numbers or string numbers of any magnitude and precision.

If either X or Y contains non-numeric data, an error message is generated; the result of the multiplication will be zero.

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.

```
SMUL (X,Y)
```

Figure A. General Form of SMUL Function

CORRECT USE	EXPLANATION
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.

Figure B. Examples of Correct Usage of SMUL Function

SPACE
Function

The SPACE function generates a string value containing a specified number of blank spaces.

The general form of the SPACE function is:

```
SPACE(expression)
```

The expression specifies the number of blank spaces to be generated in the string.

For example:

```
PRINT SPACE(10):"HELLO"
```

This statement prints 10 blanks followed by the string "HELLO".

```
SPACE(expression)
```

Figure A. General Form of SPACE Function

CORRECT USE	EXPLANATION
B = 14 A = SPACE(B)	Assigns to variable A the string value containing 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.
S = SPACE(5) L = "SMITH" C = "," F = "JOHN" N = S:L:S:C:S:F	Assigns to variable N a concatenated string consisting of 5 blanks, the name "SMITH", 5 blanks, a comma, 5 blanks, and the name "JOHN".

Figure B. Examples of Correct Usage of SPACE Function

SQRT
Function

The SQRT function returns the positive square root of a positive number.

The general form of the SQRT function is:

```
SQRT(expression)
```

The value of the expression is the positive number for which to generate the square root. If the expression evaluates to less than or equal to zero, the function returns a value of 0.

For example:

```
Y=SQRT(X)
```

This statement assigns to variable Y the positive square root of the positive number X.

```
SQRT(expression)
```

Figure A. General Form of the SQRT Function

CORRECT USE	EXPLANATION
Y = SQRT(36)	Assigns the value 6 to variable Y.

Figure B. Example of Correct Usage of the SQRT Function

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.

The general form of the SSUB function is:

```
SSUB (X,Y)
```

X and Y may be any valid numbers or string numbers of any magnitude and precision.

If either X or Y contains non-numeric data, an error message is generated; the result of the subtraction will be zero.

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.

```
SSUB (X,Y)
```

Figure A. General Form of SSUB Function

CORRECT USE	EXPLANATION
TOTAL=SSUB(SUBTOT1,SUBTOT2)	Assigns difference of variables SUBTOT1 and SUBTOT2 to variable TOTAL.
PRINT (SSUB(X,".004")	Prints difference of variable X and string constant (.004).
A=SSUB("1.030476",B)	Assigns to variable A the difference of string constant 1.030476 and variable B.
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.

Figure B. Examples of Correct Usage of SSUB Function

STOP
Statement

The STOP statement terminates program execution.

The general form of the STOP statement is:

```
STOP {errnum{,param, param, ...}}
```

Upon the execution of a STOP statement, the BASIC program will terminate.

The STOP statement may be placed anywhere within the BASIC program to indicate the end of one of several alternative paths of logic.

The STOP statement may optionally include an errnum expression, which identifies the error message name, and one or more param expressions, which are error message parameters. The errnum is a reference to an item in the ERRMSG file. The param(s) are variables or literals to be used within the error message format.

Note: The ABORT statement may also be used to terminate program execution. Unlike STOP, however, ABORT also terminates PROC execution if the program was invoked from a PROC. STOP terminates a BASIC program but allows a PROC to continue executing.

A sample BASIC program illustrating the correct use of the STOP statement is presented in Figure B.

```
STOP {errnum{,param, param, ...}}
```

Figure A. General Form of STOP Statement

```
      *
      *
      *
      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,
10    PRINT "POSITIVE PROFIT"          program will terminate
      END <------------------------ Physical end of program
```

Figure B. Program Showing Correct Usage of STOP Statement

STORAGE
Statement

The STORAGE statement allows a program to change the three buffer sizes used for storing string data in variables.

The general form of the STORAGE statement is:

STORAGE small-buffer, med-buffer, large-buffer

Each parameter is a numeric expression that specifies a buffer size. Small-buffer, med-buffer, and large-buffer must all be multiples of 10. Also med-buffer must not be less than small-buffer, and large-buffer must not be less than med-buffer.

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 which the descriptor then points to. 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. Strings up to 32266 characters in length can be accommodated.

The default small buffer size is 50 bytes, the default medium buffer size is 150 bytes, and the default large buffer unit size is 250 bytes. These three values may be altered by the STORAGE statement, where they correspond to the small-buffer, med-buffer, and large-buffer expressions.

```
STORAGE small-buffer, med-buffer, large-buffer
```

Figure A. General Form of STORAGE STATEMENT

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).

Figure B. Examples of Correct Usage of STORAGE Statement

STR
Function

The STR function generates a string value containing a specified number of occurrences of a specified string.

The general form of the STR function is:

```
STR(string,count)
```

Both string and count may be any valid expression. The value of string specifies the string to be used. The count contains the number of occurrences to be generated. The STR function returns a string value that contains string repeated count number of times.

The following statement, for example, assigns a string value containing 12 asterisk characters to variable X:

```
X=STR('*',12)
```

As a further example, the following statement will cause the string value "ABCABCABC" to be printed:

```
PRINT STR('ABC',3)
```

```
STR(string,count)
```

Figure A. General Form of STR Function

CORRECT USE	EXPLANATION
`VAR = STR("A",5)`	Assigns to variable VAR a string containing five A's.

`A = 'BBB'`
`B = STR("B",3)`
`C = B CAT A`
	Assigns to variable C the string containing six B's.

`N = STR("?%?",4)`	Assigns to variable N the string value containing 4 consecutive occurrences of the string "?%?".

Figure B. Examples of Correct Usage of STR Function

SUBROUTINE
Statement

The SUBROUTINE statement provides external subroutine capabilities for a BASIC program. An external subroutine is a subroutine that is compiled separately from the program or programs that call it.

The general form of the SUBROUTINE statement is:

```
SUBROUTINE {name} {(argument list)}
```

The optional name may be used to indicate the name of the subroutine, but this is ignored by the compiler. An external subroutine is invoked by specifying the item-id of the subroutine program item, or the item-id of a cataloged subroutine pointer, in a CALL statement.

The optional argument list consists of one or more variables, separated by commas, that take on the actual values passed to the subroutine. The arguments may be continued on multiple lines; each line except the last must end with a comma (after an argument expression).

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 statement, listed alphabetically in this chapter.)

The SUBROUTINE statement is used to identify the program as a subroutine and must be the first statement in the program.

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 are the values of the arguments. External subroutines may call other external subroutines, including themselves. A sample external subroutine that involves two arguments together with correctly formed CALL statements, is shown below.

```
CALL Statements          Subroutine ADD

CALL ADD (A,B,C)         SUBROUTINE ADD (X,Y,Z)
CALL ADD (A+2,F,X)       Z=X+Y
CALL ADD (3,495,Z)       RETURN
                         END
```

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 will be updated to reflect the most recent values of the

corresponding variables in the subroutine. Constants and other expressions used as subroutine arguments will not be 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.

NOTE: An external subroutine must begin with a SUBROUTINE statement and contain a RETURN statement. GOSUB and RETURN 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.

```
SUBROUTINE {name} {(argument list)}
```

Figure A. General Form of SUBROUTINE Statement

CORRECT USE	EXPLANATION
CALL REVERSE (A,B) SUBROUTINE REVERSE (I,X)	Subroutine REVERSE has two arguments.
CALL REPORT SUBROUTINE REPORT	Subroutine REPORT has no arguments.
CALL VENDOR (NAME, ADDRESS, NUMBER) SUBROUTINE VENDOR (NAME, ADDR,NUM)	Subroutine VENDOR returns three values.
CALL DISPLAY (A,B,C) SUBROUTINE DISPLAY (I,J,K)	Subroutine DISPLAY accepts (and returns) three argument values.
CALL COPY (MAT X, MAT Y) SUBROUTINE COPY (MAT A, MAT B) DIM A(10) DIM B(10)	Subroutine COPY has two dimensioned arrays passed to it from the calling program.

Figure B. Examples of Correct Usage of CALL and SUBROUTINE Statements

Arrays may be passed to external subroutines.

The general form for specifying an array in an argument list for the SUBROUTINE statement is:

```
SUBROUTINE name (MAT variable {,MAT variable ...} )
```

The variable is the name of an array given in a DIM statement. Multiple arrays may be passed, as needed. The array(s) must be dimensioned in both the calling program and the subroutine.

Individual array dimensions may be different, as long as the total number of elements matches. Arrays are copied in row major order. Consider the following example:

Calling Program	Subroutine
DIM X4,5)	SUBROUTINE COPY (MAT A)
CALL COPY (MAT X)	DIM A(10,2)
END	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.

Additional examples of array passing, both correct and incorrect, are shown in Figure B.

```
SUBROUTINE name (MAT variable {,MAT variable ...} )
```

Figure A. General Form of SUBROUTINE Statement with Array Passing

CORRECT USE	EXPLANATION
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)	
INCORRECT USE	**EXPLANATION**
DIM FOUR (2,2) CALL GOF (MAT FOUR)	Corresponding arrays must have the same number of elements in the calling program and the subroutine.
SUBROUTINE GOF(MAT NIX) DIM NIX(5)	

Figure B. Examples of Array Parameters

SYSTEM
Function

The SYSTEM function allows the user to obtain certain pre-defined values from the system. The value returned may either be an error status code (generated as a result of a previous BASIC statement), or a parameter such as the page-number or page-width.

The general form of the SYSTEM function is:

```
SYSTEM(expression)
```

The value of expression must be in the range 0 through the maximum value as defined in table A. If the value of expression is outside the allowable range, the SYSTEM function will return a value as if the expression evaluated to zero (the error function).

If the expression used in the SYSTEM function 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, WRITET, etc. 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.

For example, the sequence of BASIC instructions:

```
READT TAPERECORD ELSE
    BEGIN CASE
      CASE SYSTEM(0) = 1; PRINT "ATTACH THE TAPE UNIT"; STOP
      CASE SYSTEM(0) = 2; PRINT "END OF FILE; DONE!"; STOP
    END CASE
END
```

will result in one of the messages being printed if there is either an EOF read from the tape, or if the tape unit was not attached to the line running the BASIC program.

FOR SYSTEMS USING ULTINET: When an error occurs during file transfers causing the ON ERROR clause in a statement to be executed, SYSTEM(0) returns the UltiNet error message number. This number is the item-id of the error message in the ERRMSG file. The ON ERROR clause can then test SYSTEM(0) and take appropriate action (e.g., via the PUT or STOP statements). For more information about using the ON ERROR clause, see the specific statement (any READ or WRITE statement, OPEN, CLOSE, DELETE, or SELECT), listed alphabetically in this chapter.

The SYSTEM function, with non-zero values of the expression, returns parameters that have been set external to the BASIC program. Figure A shows the general format and the SYSTEM function expressions. Figure B shows the error codes returned by SYSTEM(0).

SYSTEM(expression)

Value of expression	Value returned
0	Error function value; see Figure B.
1	1 if PRINTER ON or (P) option used in RUN; 0 if 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.
10	Code for machine BASIC program is running on: H0: for Honeywell-WCS-based system H1: for Honeywell-HPP-based system D0: for DEC-based systems without typeahead D1: for DEC-based systems with typeahead and regular memory D2: for DEC-based systems with typeahead and dual-ported memory
11	Number of characters in typeahead buffer.
12	Terminating character of last INPUT statement

Figure A. General Form and Expressions for SYSTEM Function

Previously executed BASIC statement.	Error code returned.	Meaning
READT, WRITET, WEOF or REWIND	1	Tape unit is not attached.
	5	Tape unit is off-line.
	6	Cartridge is not formatted correctly.
READT	2	EOF read from tape unit.
WRITET	3	Attempted to write null string.
	4	Attempted to write variable longer than tape record length.
Any statement with ON ERROR clause	2001-2339	(Only applicable to systems using UltiNet). UltiNet error code; see UltiNet User Guide for specifics.

Figure B. Values Returned by the Error Function: SYSTEM(0)

TAN
Function

The TAN trigonometric function returns the tangent of an angle expressed in degrees.

The general form of the TAN function is:

TAN(expression)

The expression specifies the number of degrees in an angle. The TANGENT function returns the tangent of the angle.

In the following summary M is used to denote the largest allowable number in BASIC, which is 14,073,748,835.5327 with PRECISION 4.

FUNCTION	RANGE	DESCRIPTION
COS(X)	-M <= X <=M	Returns the cosine of an angle of X degrees.
SIN(X)	-M <= X <= M	Returns the sine of an angle of X degrees.
TAN(X)	-M <= X <= M	Returns the tangent of an angle of X degrees.
LN(X)	0 <= X <= M	Returns the natural (base e) logarithm of the expression X.
EXP(X)	-M <= RESULT <= M	Raises the number 'e' (2.7183) to the value of X.
PWR(X,Y)	-M <= RESULT <= M	Raises the first expression to the power denoted by the second expression.

Figure A. Summary of Trigonometric Functions

TIME
Function

The TIME function returns the internal time of day.

The general form of the TIME function is:

```
TIME()
```

This function returns the internal time of day. The internal time is the number of seconds past midnight. For example:

```
X = TIME()
```

This statement assigns the internal time to variable X.

```
TIME()
```

Figure A. General Form of TIME Function

CORRECT USE	EXPLANATION
A = TIME()	Assigns current internal time to variable A.
IF TIME() > 100 THEN GOTO 10	Branches to label 10 if more than 100 seconds have passed since midnight.

Figure B. Examples of Correct Usage of TIME Function

TIMEDATE
Function

The TIMEDATE function returns the current time and date in external format.

The general form of the TIMEDATE function is:

```
TIMEDATE()
```

The TIMEDATE function returns the string value containing the current time and date in external format. This format is:

```
HH:MM:SS  DD MMM YYYY
```

where: HH=hours
MM=minutes
SS=seconds
DD=day
MMM=month
YYYY=year

For example, the following statement assigns the string value of the current time and date to variable B:

```
B = TIMEDATE()
```

The string value assigned to variable B could then be, for example:

```
"08:30:23  06 MAY 1985"
```

```
TIMEDATE()
```

Figure A. General Form of TIMEDATE Function

CORRECT USE	EXPLANATION
PRINT TIMEDATE()	Prints the current time and date in the external format.

Figure B. Example of Correct Usage of TIMEDATE Function

TRIM
Function

The TRIM function removes extraneous blank spaces from a specified string.

The general form of the TRIM function is:

```
TRIM (expression)
```

The expression specifies the string to be trimmed. The TRIM function deletes preceding, trailing, and redundant blanks from the expression. For example:

```
A="       GOOD MORNING,       MR. BRIGGS"
A=TRIM(A)
PRINT A
```

The PRINT statement will print:

```
GOOD MORNING, MR. BRIGGS
```

```
TRIM(expression)
```

Figure A. General Form of TRIM Function

CORRECT USE	EXPLANATION
N=" SMITH , JOHN " M = TRIM(N)	Where N is the above string variable, assigns to variable M a string consisting of the name SMITH, 1 blank, a comma, one blank, and the name JOHN.

Figure B. Example of Correct Usage of TRIM Function

UNLOCK
Statement

The UNLOCK statement, in conjunction with the LOCK statement, provides a file and execution lock capability for BASIC programs. The UNLOCK statement releases the specified execution lock(s) set by the LOCK statement.

The general form of the UNLOCK statement is:

```
UNLOCK {expression}
```

The value of the expression is an integer between 0 and 47, inclusively, that specifies which execution lock is to be released (cleared). If the expression is omitted, then all execution locks which were previously set by the program will be released.

The UNLOCK statement operates in conjunction with the LOCK statement, which sets an execution lock. (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.

Once a LOCK statement has set an execution lock, it can be released only by the same program that set it.

An attempt to UNLOCK an execution lock which the program did not LOCK has no effect. All execution locks set by a program will automatically be released upon termination of the program, even if it is terminated by the END command from the BASIC Debugger.

The following is an example of the complete execution lock capability: Process A sets execution lock 42 before executing a non-reentrant section of code (that is, code which should not be executed by more than one process simultaneously). Process B executing the same program reaches the "LOCK 42" instruction, but cannot lock that section of code until Process A has unlocked 42. Thereby, the code is rendered non-reentrant.

```
UNLOCK {expression}
```

Figure A. General Form of UNLOCK Statement

CORRECT USE	EXPLANATION
UNLOCK 47	Resets execution lock 47.
UNLOCK	Resets all execution locks previously set by the program.
UNLOCK (5+A)*(B-2)	The current value of the expression (5+A)*(B-2) specifies which execution lock is released.

Figure B. Examples of Correct Usage of UNLOCK Statements

UNTIL
Statement

The UNTIL statement is an optional statement within the FOR/NEXT or LOOP statement sequences.

The general form of the UNTIL statement is:

```
UNTIL expression DO statements
```

Please refer to the FOR statement or the LOOP statement for information about the entire statement sequence.

WEOF
Statement

The WEOF statement writes an end of file mark on a magnetic tape. The tape unit is as specified by the most recent T-ATT command executed at the TCL level.

The general form of the WEOF statement is:

WEOF {THEN statements} {ELSE statements}

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

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. A tape unit must have been previously attached; if it is subsequently set off line, the system detects the condition and allows the user to correct it and proceed.

If this statement is the first tape instruction (READT, REWIND, WEOF, or WRITET) in the BASIC program, the tape unit must have previously been attached. If the tape unit has not been attached, then the ELSE statements, if any, will be executed, and the system function SYSTEM(0) will return 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, for an alternative to printing error messages.)

If, however, the tape drive is adversely 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. (Thus, the ELSE statements are not executed in either case, and the BASIC program has no way to detect such adverse action.

IMPORTANT: 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.) 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 the user to (C)ontinue, it is not guaranteed that valid data is then read or written.

Note: The WEOF statement is used in conjunction with the READT, WRITET, and REWIND statements. (For additional information, see each statement listed alphabetically in this

chapter.) See the ULTIMATE System Commands manual for more information on the T-ATT command.

```
WEOF {THEN statements} {ELSE statements}
```

Figure A. General Form of WEOF Statement

CORRECT USE	EXPLANATION
WEOF ELSE STOP	Writes two EOF marks, then backspaces over the second one.

Figure B. Examples of Correct Usage of WEOF Statement

WHILE
Statement

The WHILE statement is an optional statement within a FOR/NEXT or LOOP statement sequence.

The general form of the WHILE statement is:

```
WHILE expression DO statements
```

Please refer to the FOR statement or LOOP statement for information about the entire statement sequence.

WRITE
Statement

The WRITE statement is used to update a file item. It also unlocks the item if it was initially locked.

The general form of the WRITE statement is:

```
WRITE expression ON {file-var,} item-id {ON ERROR stmt}
```

The WRITE statement replaces the contents of the item specified by the item-id expression with the string value of the first expression. If the item-id expression specifies an item which does not exist, then a new item will be created. The optional file-var specifies the file variable; if it is used, the item will be replaced in the file previously assigned to that variable via an OPEN statement. If the variable is omitted, then the internal default file variable is used (i.e., the file most recently opened without a file variable).

The following statements, for example, replace the current contents of the item named XYZ in the file opened and assigned to variable F5 with the string value "THIS IS AN EXAMPLE":

```
VALUE = "THIS IS AN EXAMPLE"
WRITE VALUE ON F5,"XYZ"
```

Alternatively, this example may have been specified as follows:

```
WRITE "THIS IS AN EXAMPLE" ON F5,"XYZ"
```

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.)

The statement(s) after ON ERROR, if present, are executed only if the file (1) is a remote file (accessed via UltiNet) and (2) cannot be written on due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 to local files.

The ON ERROR clause 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.

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 written to due to network errors, the program may terminate with an error message if no ON ERROR clause is present.

The WRITE statement unlocks the item lock associated with the item being written, if it was initially locked. Item locks may be set with the READU, READVU, or MATREADU statements to prevent simultaneous updates of the same item by more than one program. For more information on item locks, please see the READU, READVU, or MATREADU statements, listed alphabetically in this chapter.

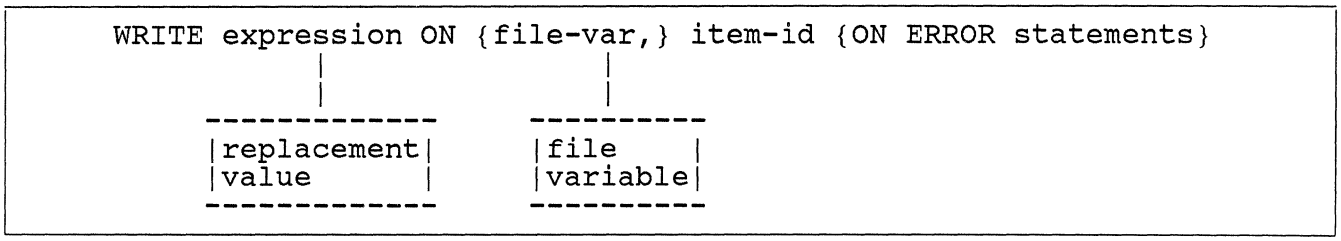


Figure A. General Form of WRITE Statement

CORRECT USE	EXPLANATION
WRITE "XXX" ON A, "ITEM5"	Replaces the current contents of item ITEM5 (in the file opened and assigned to variable A) with string value "XXX".
A="123456789" B="X55" WRITE A ON FN1,B	Replaces the current contents of item X55 (in the file opened and assigned to variable FN1) with string value "123456789".
WRITE 100*5 ON "EXP"	Replaces the current contents of item EXP (in the file opened without a file variable) with string value "500".
WRITE 10*5 ON "EXP" ON ERROR ERRNUM=SYSTEM(0) GOSUB PROCESSERR GOTO TOP END	Writes the string value "50" to item EXP, as above, or retrieves error number and performs local subroutine on UltiNet error number.

Figure B. Examples of Correct Usage of WRITE Statement

WRITET
Statement

The WRITET statement writes a record on magnetic tape. The tape unit and record length (block size) on the tape is as specified by the most recent T-ATT statement executed at the TCL level.

The general form of the WRITET statement is:

```
WRITET expression {THEN statements} {ELSE statements}
```

Either the THEN clause or the ELSE clause may be omitted, but not both; at least one of them must be present.

The WRITET statement writes the string value of the expression, if non-null, onto the next record of the "current" magnetic tape unit. If the length of the string is less than the current tape block size, the tape record (block) will be padded with trailing blanks. If the length of the string is greater than the current block size, the record will be truncated to the current block size and trailing characters in the string will not be written; in this case a warning message will be printed on the terminal, and the system function SYSTEM(0) will return a value of 4. After the write operation, the THEN statements, if any, are executed.

If the string value of the expression is the empty string (''), then the ELSE statements, if any, are executed, and the system function SYSTEM(0) will return a value of 3. If the tape unit has not been attached, the ELSE statements, if any, are executed. For example:

```
WRITET A ELSE STOP
```

This writes the string value of A onto the tape. If A is the empty string or no tape is attached, the program terminates. (See below for the exception to this procedure.)

A tape unit must have been previously attached; if it is subsequently set off line, the system detects the condition and allows the user to correct it and proceed.

If this statement is the first tape instruction (READT, REWIND, WEOF, or WRITET) in the BASIC program, the tape unit must have been previously attached. If the tape unit has not been attached, then the ELSE statements, if any, will be executed, and the system function SYSTEM(0) will return 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.)

NOTE: Refer to the SYSTEM Function for an alternative to printing the error messages.

If, however, the tape drive is adversely 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. (Thus, the ELSE statements are not executed in either case, and the BASIC program has no way to detect such adverse action.)

IMPORTANT: 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.) 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 the user to (C)ontinue, it is not guaranteed that valid data is then read or written.

Note: The WRITET statement is used in conjunction with the READT, WEOF, and REWIND statements. (For additional information, see each statement listed alphabetically in this chapter.) See the System Commands manual for more information on the T-ATT command.

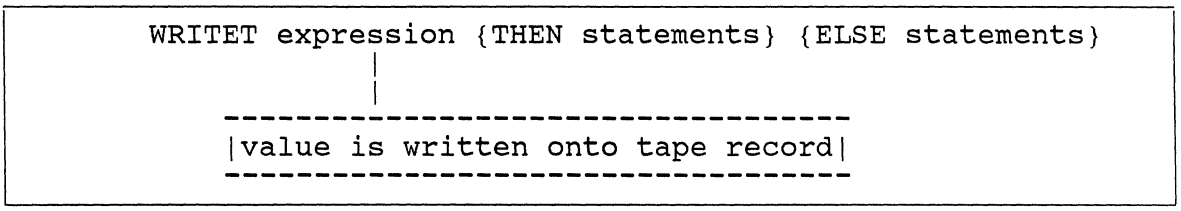


Figure A. General Form of WRITET Statement

CORRECT USE	EXPLANATION
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 5 tape records. If one of the array elements has a value of '' (or if the tape unit is not attached, the program will terminate).

Figure B. Example of Correct Usage of WRITET Statement

WRITEU
Statement

The WRITEU statement writes a file item. The item remains locked after execution of the WRITEU statement. The letter "U" is appended to the statement name to imply "update", not "unlock".

The general form of the WRITEU statement is:

```
WRITEU expression ON {file-var,} item-id {ON ERROR stmt}
```

The WRITEU statement functions the same as the WRITE statement except for the locking feature. It does not unlock the item lock, if it was initially locked, after completing the write operation. This variation on the WRITE statement is used primarily for master file updates when several transactions are being processed and an update of the master item is made following each transaction update.

If the item is not locked before the WRITEU statement is executed, it will be locked afterwards. For more information on item locks, please see the READU, READVU, or MATREADU statement, listed alphabetically in this chapter.

NOTE: The RELEASE statement can be used to unlock the item. (Please refer to the RELEASE statement, listed alphabetically in this chapter.)

```
WRITEU expression ON {file-var,} item-id {ON ERROR stmt}
```

Figure A. General Form of WRITEU Statement

CORRECT USAGE	EXPLANATION
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 GOTO PROCESSERR	Writes as above, or branches to local subroutine to process UltiNet error number.

Figure B. Example of Correct Usage of WRITEU Statements

WRITEV
Statement

The WRITEV statement is used to write a single attribute value to an item in a file. The item is unlocked if it was previously locked.

The general form of the WRITEV statement is:

```
WRITEV expression ON {file-var,} item-id, attr#
      {ON ERROR statements}
```

The value of expression replaces the current value of the attribute number specified by attr# (attribute number expression), in the item specified by the item-id expression. If a file-var is specified, the update is to the file previously assigned to that variable via an OPEN statement. If the file-var is omitted, then the internal default file variable will be used (i.e., the file most recently opened without a file variable).

If a non-existent item name (or attribute number) is specified, then a new item (or attribute) will be created. If a new attribute is created, all attributes between it and the former last attribute will be null.

The statement(s) after ON ERROR, if present, are executed only if the file (1) is a remote file (accessed via UltiNet) and (2) cannot be written on due to a network error condition. In this case, the value of SYSTEM(0) will indicate 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 to local files.

The ON ERROR clause 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.

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 written on due to network errors, the program may terminate with an error message if no ON ERROR clause is present.

Consider this example:

```
X1 = "XXX"
WRITEV X1 ON A2,"ABC",4
```

These statements replace the 4th attribute of item ABC (in the file opened and assigned to variable A2) with the string value "XXX".

The WRITEV statement will also allow the attribute number to have a value of either zero or minus one, thus inserting data prior to the first attribute or following the last attribute, respectively. For example:

```
WRITEV XX ON FILE, ITEM-ID, AMC
```

When AMC=0, the attribute XX is inserted at the begining of the item. All attributes in the item are shifted by 1 attribute and the attribute XX becomes attribute 1.

When AMC = -1, the attribute XX is appended to the end of the item. The number of attributes in the item increases by 1 and all previously existing attributes are undisturbed.

The WRITEV statement unlocks the item lock associated with the item being updated, if it was initially locked. Item locks may be set with the READU, READVU, or MATREADU statements to prevent simultaneous updates to the same item by more than one program. For more information on item locks, please see the READU, READVU, or MATREADU statement, listed alphabetically in this chapter.

The BASIC program will abort with an appropiate error message if the specified file has not been opened prior to the execution of the WRITEV Statement.

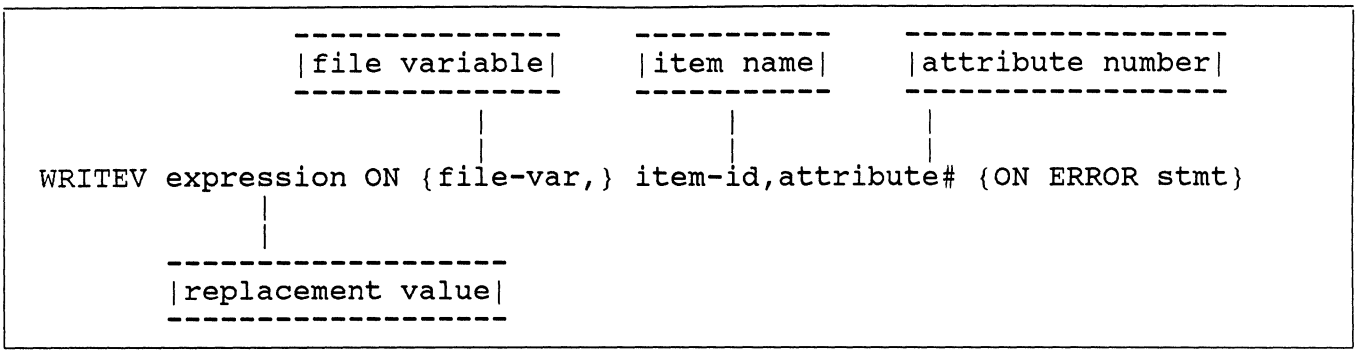


Figure A. General Form of WRITEV Statements

CORRECT USAGE	EXPLANATION
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.

Figure B. Examples of Correct Usage of WRITEV Statements

WRITEVU
Statement

The WRITEVU statement writes an attribute value to a file item. The item remains locked after execution of the WRITEVU statement. The letter "U" is appended to the statement name to imply "update", not "unlock".

The general form of the WRITEVU statement is:

```
WRITEVU expression ON {file-var,} item-id, attr#
       {ON ERROR statements}
```

The WRITEVU statement functions the same as the WRITEV statement except for the locking feature. It does not unlock the item lock, if it was initially locked, after completing the write operation. This variation on the WRITEV statement is used primarily 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 will be locked afterwards. For more information on item locks, please see the READU, READVU, or MATREADU statements, listed alphabetically in this chapter.

NOTE: The RELEASE statement can be used to unlock the item. (Please refer to the RELEASE statement, listed alphabetically in this chapter.)

```
WRITEVU expression ON {file-var,} item-id, attr# {ON ERROR statements}
```

Figure A. General Form WRITEVU Statement

CORRECT USAGE	EXPLANATION
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.
WRITEVU NAME ON CFILE,ID,3, ON ERROR	GOTO PROCERR Writes as above, or branches to local subroutine to process UltiNet error number.

Figure B. Example of Correct Usage of WRITEVU Statement

NOTES

CHAPTER 4

TESTING AND DEBUGGING BASIC PROGRAMS

4.1 BASIC Symbolic Debugger

The BASIC Symbolic Debugger facilitates the debugging of new BASIC programs and the maintenance of existing BASIC programs.

The BASIC Debugger may be entered at execution time by 1) depressing the BREAK key or 2) using the 'D' (debug) option with the RUN verb. It is also entered automatically under some error conditions. Once the BASIC Debugger has been entered, it will indicate the source code line number about to be executed and will prompt for commands with an asterisk (*) as opposed to the Assembly Debugger prompt: '!' and the TCL prompt: '>'.

The user has at his disposal the following general capabilities:

1. Controlled stepping through execution of program by way of single or multiple steps.

2. Transferring control to a specified step (line number).

3. Breaking (temporary halting) of execution on specified line number(s) or on the satisfaction of specified logical conditions.

4. Displaying and/or changing any variable(s), including dimensioned array variables.

5. Tracing variables.

6. Conditional entry to the Assembly Debugger.

7. Directing output (terminal/printer).

8. Stack manipulation (displaying and/or popping the stack).

9. Displaying of specified (or all) source code line(s).

NOTE: Variables cannot be referenced in programs compiled with the S option, which inhibits saving the symbol table. 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".)

Figure A shows a summary of the Symbolic Debugger features and related commands. The following sections describe in detail the commands and their use.

During program test runs, SYS2 privileges are required for all commands other than G, END, and OFF. This prevents users from making unauthorized changes to data during reporting and data entry.

BASIC DEBUGGER FEATURE	RELATED COMMAND
1. Set breakpoint on logical condition where 'o' is logical operator <,>,=,#; 'v' is variable; 'c' is condition to meet; or 'n' is line number where preceded by B$o.	Bvoc{&voc} B$on
2. Display breakpoint table	D
3. Escape to Assembly Debugger	DEBUG or DE
4. Single/multiple step execution	En
5. End program execution and return to TCL	END
6. Proceed from breakpoint	G
to specified line 'n'	Gn
7. Remove all breakpoints	K
specified breakpoint 'n'	Kn
8. Display source code current line	L
'n' number of lines from current one	Ln
number of lines from m-n	Lm-n
all lines	L*
9. Switch output from terminal to printer/ from printer to terminal	LP
10. Clear debug entry delay counter	N
Delay entry until 'n' breakpoints/steps encountered	Nn
11. Logoff	OFF
12. Inhibit/enable output	P
13. Printer-close output to spooler	PC
14. Pop return stack	R
Display return stack	S
15. Turn trace table on/off	T
Trace specified variable 'v'	T{/}v
16. Remove all traces	U
specified trace	U{/}v
17. Request source (if not in same file/same name)	Z
18. Display current program name and line number; verify object code	$ or ?
19. Print value of variable 'v'	/v
of element 'x' in array 'm'	/m(x)
of element 'x,y' in matrix 'm'	/m(x,y)
of entire array 'm'	/m
entire symbol table	/*
20. Set window	[x,y]
remove window setting	[

Figure A. Summary of BASIC Debugger Features and Commands

4.2 The Symbol Table

The symbol table is used to reference variables.

The symbol table contains all variable names defined in a program. 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 / command. (For details, please refer to Section 4.8, Displaying and Changing Variables: the / Command).

4.3 Displaying Source Code: L and Z Commands

The L command displays one or more lines of source code from the same file as the object code program. 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.

L command

The L command displays source code. The general form is:

```
L{m-{n}} {*}
```

If the form L is used, only the current line is displayed. If the form Ln is used, then the line number specified by n is displayed. If the form Lm-n is used, then the line number range specified by m-n is displayed. If the form L* is used, then all lines in the source program is displayed.

The Z command is used when the source code is from another file or stored under a different program name. The form of the Z command is:

```
Z
```

The BASIC Debugger then displays the following message:

```
File/prog name?
```

The user then enters the filename and the program name (item-id), separated by a blank. If the file and program are found, the BASIC Debugger returns with a prompt (*). If the program is not found, the message "No Source" is displayed before returning to the prompt. If either the filename or program name is not specified, or the filename is invalid, the "File/prog name?" message is repeated until a valid filename and program name are entered, or a carriage return <CR> is entered, aborting the Z command.

L{m-{n}}	Displays program source lines specified by values of m and n.
L*	Displays all source lines of a program.
Z	Prompts for source code filename and program name.

Figure A. General Form of L and Z Commands

4.4 The Trace Table: T and U Commands

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.

Up to six variables may be entered in the trace table associated with each program or external subroutine. The values of these variables will be printed whenever an execution break occurs. These breaks occur if the BREAK key is pressed, or a breakpoint is encountered, or after each of 'n' statements is executed where 'n' is a non-zero number specified with the E command. (For details on the E command, see Section 4.7, Execution Control.)

The trace table may be alternately turned on and off by use of the T command. If the command is accepted, a plus sign is printed next to it. If the variable does not exist or the wrong symbol table is assigned, the message 'Sym not fnd' is displayed.

Examples of use of the trace table are shown below:

T{/}var	The value of the 'variable' will be printed out at each execution break.
T	Turns trace off if it was on, or turns trace on if it was off. The word 'ON' or 'OFF' will be displayed.

The U command is used to delete variables from the trace table. A minus sign is printed next to the command to indicate that an entry has been removed. Its two forms are:

U{/}variable	Deletes variable from the trace table
U	Deletes the entire trace table.

The D command is used to display the trace and breakpoint tables for the currently executing program or external subroutine. (For details, please refer to Section 4.6.)

TNAME	Sets a trace for variable 'NAME'
TDISC(2)	Sets a trace for the second element of the array DISC. Note that only individual array elements may be traced.
T	Turns trace on or off.
D	Displays the trace and breakpoint tables.
UNAME	Deletes the variable NAME from the trace table.
U	Deletes all variables from the trace table.

Figure A. Trace Table Commands

4.5 Breakpoint Table: B and K Commands

The breakpoint table is used to establish conditions that will cause a break in program execution for test purposes.

The B command sets breakpoints, and the K command kills (removes) breakpoints. The breakpoint table for each program or external subroutine may contain up to four conditions, any one of which will, when satisfied, cause a break in execution. Logical expressions are used to set the break conditions.

The general form of the B command is:

```
B condition(s)
```

The logical operators and special symbols used to set the break conditions are:

```
<    less than
>    greater than
=    equal to
#    not equal to
&    a logical connector between conditions (AND).
$    a special symbol to specify a line number condition
```

The basic forms of breakpoint are shown below:

```
B variable-name operator expression {& another condition}

B $ operator line-number {& another condition}
```

where variable-name is a simple variable or an explicitly stated array element and expression is a variable, constant, or array element. If the variable does not exist the message "Sym not fnd" will be printed. String constants must be enclosed in quotes using the same rules that apply to BASIC literals. Consider the following examples:

```
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 if X is equal to 3.
```

A plus sign will be printed next to the command if it is accepted. When the condition is met, an execution break will occur and the debugger will halt execution of the program and print:

```
*Bn m
```

where n is one of the 4 breakpoint table entries and m is the program line number that caused the break.

The D command is used to display the trace and breakpoint

tables for the currently executing program or external subroutine. (For details, please refer to Section 4.6.)

The K command is used to delete breakpoint conditions from the table. A minus sign will be printed next to the command to indicate that an entry has been removed. The command has two forms:

Kn Deletes the 'n'th breakpoint condition where 'n' is in the range 1-4. All other breakpoints remain the same.

K Deletes all breakpoint conditions.

Bvariable-name operator expression
Sets breakpoint(s) by condition of a variable
B$ operator line-number
Sets breakpoint(s) by condition of a line-number
D Displays the trace and breakpoint tables
Kn Deletes specified breakpoint
K Deletes all breakpoints

Figure A. Breakpoint Table Commands

CORRECT USAGE	EXPLANATION
BX<42	Sets a break condition to halt execution when X is less than 42.
BADDRESS=''	Breaks when ADDRESS is null.
BDATE=INV.DATE&$=22	Breaks when variable DATE is equal to variable INV.DATE and if the line number is 22.
K2	Kills the second breakpoint condition.
BPRICE(3)=24.98	Sets a break condition to halt execution when the third element of the array PRICE is equal to 24.98. Only individual array elements may be specified.
K	Kills all breakpoint conditions.

Figure B. Correct Examples of B and K Commands

4.6 Displaying Tables: D command

The D command is used to display the trace and breakpoint tables.

The general form of the D command is:

```
D
```

This command displays all trace and breakpoint tables for the currently executing program or external subroutine.

Up to six trace tables and four breakpoint tables can be displayed.

Figure A shows sample output from a D command.

```
T1    TKN
T2    LINE
T3    ADDR
T4
T6
T6
B1    $=356
B2    ADDR>100
B3
B4
```

Figure A. Sample Output from a D Command

4.7.Execution Control: E, G, and N Commands

The commands E, G, and N, in conjunction with the breakpoint table, control the execution of the program under debug control.

The general forms of the E command are:

En	will execute 'n' lines and then cause an execution break.
E	execution continues until interrupted by the user, by a breakpoint, or until the program ends.

The En form allows execution of 'n' lines in the program before causing an execution break. This condition will be overridden if any entry in the breakpoint table is satisfied, which also causes an execution break. The E form turns off the E function so that breaks will occur only if an entry in the breakpoint table is satisfied or the BREAK key is pressed.

The general forms of the N command are:

Nn	bypass 'n' execution breaks
N	do not bypass execution breaks

The Nn form causes the BASIC Debugger to proceed through 'n' execution breaks before printing a prompt and returning control to the user. The variables being traced will still be printed at each breakpoint. The N form resets this function so that control is passed to the user on every execution break.

The general forms of the G command are:

Gn	go to line number 'n'
G	go to next line number
linefeed	go to next line number

The Gn form specifies that execution is to resume on that line number within the program. If the line number specified is greater than the number of lines within the program, the message 'Nstat' is displayed. The G form specifies that execution is to resume with the very next line in the program. The linefeed command is equivalent to G with no 'n' option. In any case, control is returned to the user when a condition in the breakpoint table, if any, is satisfied or when the BREAK key is pressed.

E	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.
N	Returns Debugger to the single execution break mode (normal mode).
N2	Allows two execution breaks to pass before returning control back to the user.
G	Program execution is resumed at the very next line in the program.
G37	Execution is resumed at line 37 of the program.

Figure A. Examples of E, N and G commands

4.8 Execution Control: END and OFF Commands

The END command allows for program termination and the OFF command logs the user off.

The general form of the END command is:

```
END
```

This will terminate program execution and return the user to TCL.

The general form of the OFF command is:

```
OFF
```

This will terminate program execution and will log the user off.

4.9 Displaying and Changing Variables: the / Command

Variables and arrays can be displayed and changed in either decimal or string formats during program execution.

The general forms of the / command are:

```
/v      where v is variable name

/*      displays all variables
```

where v is a variable name. The variable may be a simple variable, array name, explicitly stated array element or a COMMON variable. If the variable does not exist, then the message 'Sym not fnd' is displayed. If the variable is found, the value is displayed and the user will be prompted with an equals (=) sign. A new value may then be entered, followed by a carriage return. All values are entered as strings (without surrounding quotes). Entering a carriage return alone causes the variable to retain its current value. If displaying an entire array, each element is displayed until all elements are exhausted or the BREAK key is pressed.

The form /* will display all variables, but no changing of the values of the variables is permitted.

Figure A shows examples of using the / command.

/NAME	Displays the value of the variable NAME.
/Y(3)	Displays the value of the third element of the array Y.
/X(2,3)	Displays the value of the 4th row, 5th column of the dimensioned array X (a matrix).
/Z	Displays the value of each element of the array Z.
/*	Displays the values of all variables in the program. No changing of these values is permitted.

Figure A. Examples of the / command.

4.10 Special Commands

Special commands allow the user to control input/output, display of program name and line number, and to pass control to the Assembly Debugger.

P command

The P command suppresses all output from the program to the terminal so that the user may look at only the Debugger output. Any subsequent P commands issued will toggle this function and the words 'Off' or 'On' will be printed next to the command.

LP command

The LP command, which is similar to a PRINTER ON command in BASIC, sends all output to the printer. Any subsequent LP commands issued will toggle this function and the words 'Off' or 'On' will be printed next to the command.

PC command

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 outputted.

? command and $ command

Either the ? command or the $ command may be used to display the current program or external subroutine name and current line number, and to perform a check-sum calculation to verify the integrity of the object code.

[] command

The general form of the [] (string window) command is:

```
[s,l]
```

where s is the starting character position and l is the number of characters. This command will cause the values of all variables printed by the BASIC Debugger to be limited to the specified substring or "window". Setting l to 0 has the same effect as entering the [command.

[command

The [command resets the effect of a previous [] command, causing values of variables to be printed in their entirety.

DEBUG command

The DEBUG (or DE) command passes control to the Assembly Debugger. The Assembly Debugger G or line-feed commands may

be used to return to the BASIC Debugger. (For more information about the Assembly Debugger, please refer to the Assembly Language Manual.)

P	Suppression of program terminal output. Debugger output is not affected. Additonal P's toggle this command.
LP	All program output is to the printer. Additional LP's toggle this command.
PC	Forces printing of output data prior to program termination.
? or $	Displays program name and current line number.
[s,l]	Sets string window as specified by values of s and l.
[	Resets string window
DE{BUG}	Passes control to the Assembly Debugger.

Figure A. Summary of Special BASIC Debugger Commands

4.11 Example of Using the BASIC Debugger

This section shows a sample of using the BASIC Debugger.

Figure A is the BASIC program source listing and Figure B is a sample of using the BASIC Debugger.

The text <CR> indicates the user pressing the carriage return.

```
      TEST3
001   A=123456.7891
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
```

Figure A. Sample Program

Dialogue	Explanation
>RUN BP TEST3 (D)	Run program with 'D' option to break before first line is executed.
***E1**	Indicates execution halted before line 1.
***/X <CR>** X(1) 0=**<CR>** Unasgn var X(2) 0=**<CR>** Unasgn var X(3) 0=**<CR>** Unasgn var	Display array X. Did not change any elements; value is zero since no lines have been executed. Variable not changed.
***B$=5 <CR>** + ***G <CR>** *B1 5	Break when line number is 5. Go Indicates break condition satisfied; about to execute line 5.
***/X(1) <CR>** 123456 **<CR>** ***TX(2) <CR>** + ***E1 <CR>** ***G <CR>** *E6	Display X(1). Leave unchanged. Trace X(2). Print at each break. Set single step. Break at each statement. Go Indicates execution break caused by E1; program is about to execute line 6.
X(2) HELLO THERE ***G <CR>** *E7	X(2) automatically displayed by trace. Go Indicates execution break caused by E1; program is about to execute line 7.
X(2) HELLO THERE	X(2) automatically displayed by trace.
***E <CR>** ***$ <CR>** BP TEST3 L 7 Object verifies ***/A <CR>** 123456.7891= **<CR>**	Set execution to normal mode. E1 off. Find what line is about to be executed. Display variable A. Leave unchanged.
***P <CR>** Off ***B$=10 <CR>** + ***D <CR>** T1 X(2) T2 T3 T4 T5 T6 B1 $=5 B2 $=10 B3 B4	Turn terminal print off. Break when line number is 10. Display trace and break tables
***K1 <CR>** 1 -	Kill first break condition ($=5).
***/A <CR>** 123456.7891=**356.71<CR>** ***/A <CR>** 356.71 **<CR>** ***END <CR>**	Display variable A; change to 345.71. Display variable A. Leave unchanged. End execution of program.

Figure B. Sample of Terminal Session Using BASIC Symbolic Debugger

CHAPTER 5

REFERENCE FOR PROGRAMMERS

5.1 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. In both portions, individual records (items) are retrieved through random access by record key (item identifier, or "item-id"). Within a record, multiple values can be stored in a single field. A program can access any level of data within a record: attributes (fields), values, and subvalues.

ULTIMATE data files are designed for flexibility with efficient use of system resources. Each file is composed of two parts: a dictionary section that may be used to define the attributes (fields) of the file, and a data section that contains one item (record) for each instance of data (e.g., each customer in a Customer file).

Many BASIC programs work with ULTIMATE data files to access and retrieve, or maintain and update, information. The ULTIMATE BASIC language is designed to allow programmers to access and update any level of data: dictionary, records (items), attributes, values, and subvalues.

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 defined explicitly by a set of items (attribute definition items) in the dictionary of the Customer File, but this is not required.

The dictionary is a reference tool, but 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 dictionary can be searched for the attribute name, 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 manuals.

Each record 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. 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.

Handling System Delimiters

The ULTIMATE system uses three levels of field data: attributes, values, and subvalues. Figure A shows a sample of attributes, values, and subvalues within a file.

The ULTIMATE system uses standard attribute, value, and subvalue delimiters. For efficiency and minimalization of errors, these should be defined once in the initialization portion of a program with = (Assignment) or EQUATE statements, and then referenced by variable name:

```
 AM = CHAR(254) or EQUATE AM TO CHAR(254)  Attribute Mark
 VM = CHAR(253) or EQUATE VM TO CHAR(253)  Value Mark
SVM = CHAR(252) or EQUATE SVM TO CHAR(252) Subvalue Mark
```

A 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:

```
AM is shown as an up-arrow (^), as in NAME^
VM is shown as a right bracket (]) as in 100]
SVM is shown as a backslash (\) as in 1\2\
```

In addition, a fourth delimiter, Segment Mark, is used internally by the system to mark the end of every string referenced by a BASIC program. Consequently, no BASIC program can refer to a Segment Mark character, usually abbreviated SM. The value of a Segment Mark in BASIC would be CHAR(255); it is sometimes shown as an underscore or back-arrow (_) as in XYZ_. However, when discussing data which is understood to be delimited by a Segment Mark, the (trailing) SM is often not shown, as in the following examples.

Figure B shows a sample file with attributes, values, and subvalues. Note that only one delimiter is needed between data elements. A subvalue is delimited by "\" if another subvalue within the current value follows, or by "]" if another value follows, or by "^" if an attribute follows, or by "_". A value is delimited by "]" if another value follows, or by "^" if an attribute follows, or by "_". An attribute can only be delimited by an attribute mark "^", or by "_".

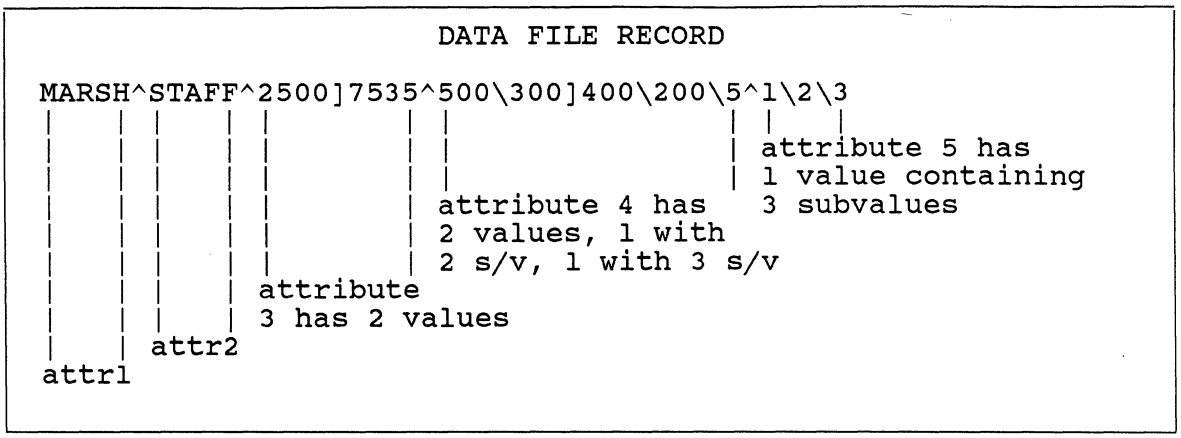


Figure A. Diagram of Item in a Data File

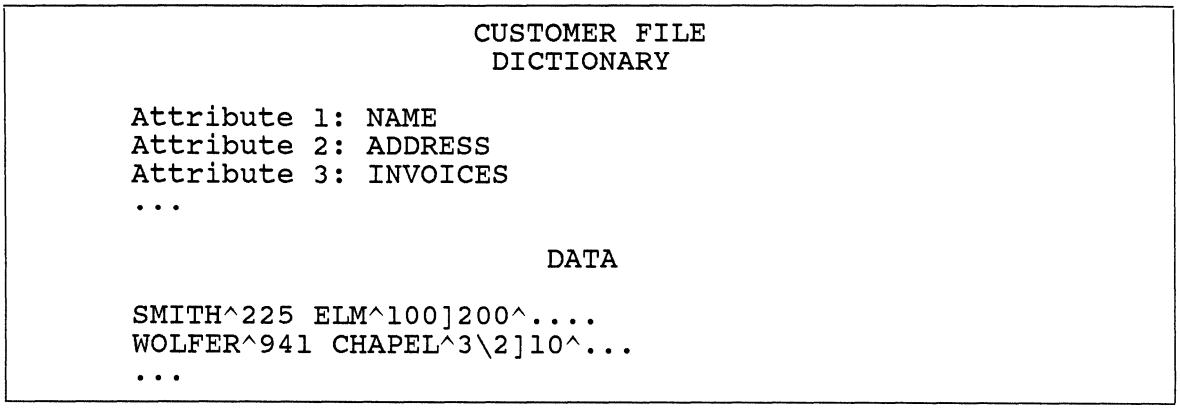


Figure B. Sample of an ULTIMATE File: Dictionary/Data

5.2 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; each has a particular purpose and "correct" usage:

Name	Purpose and Usage
OPEN	to open a file; required for any I/O functions
CLOSE	to close a file; recommended for UltiNet users
SELECT	to create a select list containing the item-ids of all records in a specified file or the attributes of a specified dynamic array
READNEXT	to sequentially retrieve item-ids from a select list so that the actual record can be read from file
READ(U)	to read a record specified by an item-id (key) into a variable
MATREAD(U)	to read a record specified by an item-id (key) into a dimensioned array
READT	to read the next record on magnetic tape; note that tape can only be read sequentially, record by record.
READV(U)	to read a specified attribute number in a record specified by an item-id (key) into a variable. NOTE: This statement should only be used when a single attribute is to be accessed from an item.
DELETE	to delete a record with a specified item-id
MATWRITE(U)	to write a record with a specified item-id from a dimensioned array
WRITE(U)	to write a record specified by an item-id (key) from a variable
WRITET	to write the next sequential record to tape
WRITEV(U)	to write a specified attribute number from a variable to that attribute in a record specified by an item-id (key). NOTE: This statement should only be used when a single attribute is to be written to an item.
RELEASE	to unlock item lock(s) set by the program
CLEARFILE	to delete all data items in a file

File Handling

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.

Item Selection - Select Lists

To access an item (record) in a file, a BASIC program must specify the item-id (key). A program cannot sequentially read through each record in a file without specifying the keys. It can, however, create a list of item-ids, and then use READNEXT statements in conjunction with READ statements to access items 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.

The READNEXT statement is designed to sequentially retrieve item-ids from a select list, thereby making them available for a subsequent READ statement. The READNEXT statement requires a select list to be available. Select lists can be created by a SELECT statement in BASIC or a (S)SELECT command in Recall. The BASIC SELECT statement does not use any selection criteria; all items/elements are included on the list. The (S)SELECT Recall command, however, is very flexible; it can select (and sort, if desired) items according to a condition(s) that a specified attribute(s) must meet. The list usually contains only item-ids, but it can contain attributes and values as well.

Before running a BASIC program, items can be selected and sorted according to user specifications, creating a select list of just the item-ids (not the entire records). The select list can then be used by the program to retrieve only the file records needed for processing. The Recall (S)ELECT command can also be used in an EXECUTE statement to generate a select list from within a BASIC program, instead of prior to running the program.

Figure A shows a simple sequence of statements to process all items in a file.

Using Recall SELECT Commands

The Recall SELECT and SSELECT commands allow sophisticated pre-processing of a file due to selection criteria which the user may specify in order to retrieve only those items meeting certain conditions. The SSELECT command, moreover, allows access to items in a sorted sequence. For example:

```
     SSELECT CUST BY DATE BY AMOUNT WITH CITY="LOS ANGELES"
OR "NEW YORK"
```

This statement sorts into date sequence (ascending order) the items in CUST representing customers in Los Angeles or New York. If more than one item has the same DATE, these items are sorted into AMOUNT sequence (ascending order).

A Recall (S)SELECT command can be executed from the terminal at the TCL level, or within a PROC, immediately before running a BASIC program. Or, the (S)SELECT command can be included within the BASIC program by inserting it in an EXECUTE statement. (For details, please refer to the EXECUTE statement, listed alphabetically in Chapter 3.)

If the (S)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. Both can be accessed by using these READNEXT statements:

```
READNEXT CUST FROM NAMES
READNEXT INV FROM ORDERS
```

```
     OPEN INV ELSE STOP
     SELECT
10   READNEXT REC ELSE STOP
     READ A FROM REC
     process
     WRITE A ON REC
     GOTO 10
```

Figure A. Sample I/O Loop

5.3 Programming Considerations about I/O 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.

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. It is the responsibility of the application (i.e., the BASIC program), however, 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 (see Section 5.2 for a listing) 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 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. It is recommended that:

1. All I/O statements include an ON ERROR clause;

2. The ON ERROR statement(s) use the SYSTEM(0) function to retrieve the current error number;

3. Statements such as PUT or STOP are used to display the error number and associated text before either resuming the program or terminating its execution.

5.4 Programming Techniques for Handling File Items

When a record 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 an item called 1234 with these attributes:

```
          1234 (NUMBER; the key/item-id)
1         STERN (LAST-NAME)
2         JEFF (FIRST-NAME)
3         125 MORNINGSIDE (ADDRESS)
4         ORANGE (CITY)
5         92667 (ZIP)
6         10]12 (ORDERS)
```

Assume further that the following BASIC statements have been executed:

```
OPEN 'CUST' TO FVAR ELSE STOP 201,'CUST'
SELECT FVAR
READNEXT REC FROM FVAR ELSE PRINT 'DONE';STOP
```

At this point REC = 1234. The next statement reads the item:

```
READ A FROM FVAR,REC ELSE GOTO 9999
```

After the READ, the item is stored in variable A:

```
A =  STERN^JEFF^125 MORNINGSIDE^ORANGE^92667^10]12
```

Several methods are available to access the attributes in this variable. Another assignment statement could be used:

```
LAST.NAME = A<1>
```

Angle brackets enclose the attribute number. If a value is to be assigned, the angle brackets would enclose the attribute number and value number; for example:

```
VAL = A<6,2>
```

would assign the second value of the sixth attribute to the variable VAL.

The assignment statement could use an intrinsic function to do the same operation:

```
LAST.NAME = EXTRACT(A,1)
```

A number of intrinsic functions are available for file update functions. Other functions besides EXTRACT are REPLACE,

INSERT, and DELETE. (For details, please refer to the appropriate function, listed alphabetically in Chapter 3.)

An array string can be built from two or more attributes:

```
NAME = 'STERN':CHAR(254):'JEFF'
```

To write an updated item to the file, a WRITE statement may be used:

```
WRITE A ON FVAR,REC ELSE GOTO 8888
```

Using 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:

```
DIM A (6)
MATREAD A FROM FVAR,REC ELSE GOTO 9999
```

result in the assignment to array A of the following:

```
A(1) = STERN
A(2) = JEFF
A(3) = 125 MORNINGSIDE
A(4) = ORANGE
A(5) = 92667
A(6) = 10]12
```

Each attribute is then accessed by its corresponding dynamic array element:

```
LAST.NAME = A(1)
```

An assignment statement for A(1):

```
A(1) = "STEER"
```

would change the original value of the LAST.NAME attribute.

When an item read with MATREAD is ready for updating, a MATWRITE statement may be used:

```
MATWRITE A ON FVAR,REC
```

Dealing with an Unknown Number of Values

The DCOUNT intrinsic function may be used to determine the number of values (including null values) in an attribute. For example:

```
VM=CHAR(253); *OR EQUATE VM TO CHAR(253)
READV ATTR FROM ID, ATTNO ELSE STOP
VALCOUNT=DCOUNT(ATTR,VM)
FOR I=1 TO VALCOUNT
        PRINT ATTR<1,I>
NEXT I
```

5.5 Guidelines for Cursor Positioning

The @ function should be used so that the correct control characteristics are sent to the terminal regardless of terminal type, which is specified by the TERM command.

Cursor positioning should be controlled by the following PRINT Statements using the @ Functions. These functions are detailed in Chapter 3; see the @ Function.

```
PRINT @(-1)  =   ERASE SCREEN
PRINT @(-2)  =   HOME
PRINT @(-3)  =   CLEAR TO END OF SCREEN
PRINT @(-4)  =   CLEAR TO END OF LINE
PRINT @(-5)  =   START BLINK
PRINT @(-6)  =   STOP BLINK
PRINT @(-7)  =   START PROTECT
PRINT @(-8)  =   STOP PROTECT
PRINT @(-9)  =   CURSOR LEFT 1 CHARACTER
PRINT @(-10) =   CURSOR UP 1 LINE
PRINT @(-11) =   CURSOR DOWN 1 LINE
PRINT @(-12) =   CURSOR RIGHT 1 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 UNDERLINING
PRINT @(-18) =   STOP UNDERLINING
PRINT @(-19) =   START INVERSE VIDEO
PRINT @(-20) =   STOP INVERSE VIDEO
PRINT @(-21) =   DELETE LINE
PRINT @(-22) =   INSERT LINE
PRINT @(-23) =   SCROLL SCREEN UP 1 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
```

5.6 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 pre-defined rather than repetitively performed.

Minimizing Program Size

An example of handling 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
```

These statements should have been written as follows:

```
    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.

Avoiding Repetitive Operations

As an example of handling repetitive operations, this statement:

```
X=SPACE(9-LEN(OCONV(COST,'MCA'))):OCONV(COST,'MCA')
```

should have been written as follows:

```
E=OCONV(COST,'MCA')
X=SPACE(9-LEN(E)):E
```

In the same context, the following operation:

```
FOR I=1 TO X*Y+Z(20)
  ...
  ...
  ...
NEXT I
```

should have been written as follows:

```
TEMP=X*Y+Z(20)
FOR I=1 TO TEMP
  ...
  ...
  ...
NEXT I
```

5.7 Programming Example: PRIME

This program finds prime numbers.

Figure A shows a sample run which finds the prime number closest to 44.

```
      *
      *  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 # to test ':
      INPUT NUM
10    NULL
      IF REM(NUM,2) = 0 THEN PRINT NUM: ' is even!'; NUM=NUM+1
20    NULL
      FOR N=3 TO SQRT(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
```

```
>RUN BP PRIME

Enter # to test ?44

44 is even!
45 is divisible by 3
47 is prime!

>
```

Figure A. Sample Run of PRIME

5.8 Programming Example: COLOR

This program allows a user with an ADDS Viewpoint Color (terminal type C) terminal to set the screen foreground and background colors.

Figure A shows a sample run which selects a green foreground on a black background.

```
*
* PROGRAM TO SELECT FOREGROUND AND BACKGROUND COLORS
* FOR VIEWPOINT COLOR TERMINALS
*
EQU AM TO CHAR(254)
C='X':AM:'B':AM:'C':AM:'R':AM:'M':AM:'W':AM:'Y':AM:'G'
D='Black':AM:'Blue':AM:'Cyan':AM:'Red':AM:'Magenta':AM:
                    'White':AM:'Yellow':AM:'Green'
AMC=DCOUNT(C,AM)
FOR I=1 TO AMC
PRINT C<I>: ' - ':D<I>
NEXT I
*
PRINT 'Foreground Color ':
INPUT F
PRINT 'Background Color ':
INPUT B
*
PRINT CHAR(27): '7@':F:B
END
```

```
>RUN BP COLOR

X - Black
B - Blue
C - Cyan
R - Red
M - Magenta
W - White
Y - Yellow
G - Green
Foreground Color ?G
Background Color ?X
```

Exhibit A. Sample Run of COLOR

5.9 Programming Example: P0000 (File Update)

This program uses terminal input to update a master file.

Figure A shows a sample of the terminal display.

```
      *
      *** 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- Prom Master Update ***':
      @(70,1):'P0000':@(4,4):'Enter Prom Part Number: '
      PROMPT ''
      PS=@(0,12):CLRL:BELL
      PRMPT=@(0,12):CLRL:"Enter Line # to change/'D' to delete item/-NL-
      to update"
      SCR='COMPANY:'L15:'PROM WIDTH;'L15:'PROM DEPTH:'L15:
      'F/P CODE:'L15:'FILL CHAR:'L15
      *
      OPEN 'PROMMASTER' TO PM ELSE STOP 201,'PROMMASTER'
10    FLG1=0; PRINT TL:; INPUT PID
      IF PID = "END" ! 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 200
100   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 210
      GO 100
      *
200   * ADD NEW ITEM *
      FOR W=1 TO 4
210   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 210
      CASE W=2
        IF P(2)#4 & P(2)#8 THEN
           PRINT PS:'Must be a 4 or 8 in this field'; GO 210
        END
      CASE W=3
        IF NOT(NUM(P(3))) THEN
 215       PRINT PS:'Invalid response, must be decimal/K units'
           GO 210
           END
        IF REM(P(3),32)#0 THEN GO 215
      CASE W=4
        IF NOT(P(4) MATCH FPMSK) THEN
           PRINT PS:'Must be 4 decimal digits'; GO 210
           END
```

```
CASE W=5
  IF P(5)#0 & P(5)#"F" THEN
     PRINT PS:'Must be "0" or "F" '; GO 210
     END
END CASE
IF NOT(FLG1) THEN GO 100
NEXT W
FLG1=0 ; * ITEM NOW EXISTS
GO 100
```

```
     ***CCARM Corp - Prom Master Update***     P0000

Enter Prom Part Number: 222

1. Company:
2. Prom Width:
3. Prom Depth:
4. F/P Code:
5. Fill Char:

Enter Line # to change/'D' to delete item/-NL- to update _
```

Figure A. Sample Terminal Output

5.10 Programming Example: 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.

Figure A shows a sample of the program's output to the terminal.

```
      *** PROGRAM USING THE EXECUTE STATEMENT***
      *
      OPEN "DICT","MD" TO MD ELSE
         PRINT "CAN'T OPEN THE FILE CALLED MD"
         STOP
      END
      CLEAR = @(-1)
      CES = @(-3)
      CEL = @(-4)
100   PRINT "ENTER THE DICTIONARY CODE FOR THE SEARCH OR 'END' ":CES:
      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:'"',
        //SELECT. > ID, //OUT. > XXX
      IF XXX[2,3] = "401" THEN
         PRINT "NO DICTIONARY ITEMS FOR CODE = ":CODE
         PRINT @(0,22):
         GOTO 100
      END
      PRINT CLEAR:
      PRINT "MASTER DICTIONARY ITEMS WITH A DICTIONARY CODE OF - ":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 TO DISPLAY":CES:
            PRINT " THE REMAINING ITEMS, "
            PRINT " PRESS <RETURN> TO CONTINUE, OR (C)ANCEL ":
            INPUT ANS:
            IF ANS[1,1] = "C" THEN PRINT @(0,22): ; GOTO 100
            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>)
         I = I + 1
         Y = Y + 1
      REPEAT
      PRINT @(0,22):
      GOTO 100
      STOP
   END
```

```
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'?_
```

Figure A. Sample Terminal Output

5.11 Programming Example: SUMMARY.REPORT (Menu/Report Generator)

This program illustrates sample coding from a menu-driven set of report generation programs. The operator selects the desired report option from a "D&B Prospect Selector" menu, which is produced by a PROC. The PROC calls the appropriate application to print the selected report. The application prints the report and returns to the PROC to re-display the menu.

Figure A shows the menu of reports. The BASIC program listing below contains the coding for the "Summary Prospect Report" option. An abbreviated listing of a subroutine called GET.CRITERIA, which is called by SUMMARY.REPORT, is included below the main program.

NOTE: In the program below, some lines are too long to fit within the margins of this document. A right arrow symbol (-->) is used to signify that the same program line is continued on the next display line, indented ten spaces.

```
      *** 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,SIC.CODES,SIC.SELECT,SORT.BY
      COMMON TITLE,NAME,FLAG
      PROMPT ''
      DIM REPORTS(11)
      MAT REPORTS = ''
      REPORT = 1
100   PRINT @(-1):@(10,0):"Selective Prospect Summary Report -->
                          Report #":REPORT:
      ***
      *** CALL SUBROUTINE TO PROMPT FOR SELECTION CRITERIA
      ***
      CALL GET.CRITERIA
      IF FLAG = 'X' THEN GO 100
      IF FLAG = '-' THEN GO 900
      ***
800   *** BUILD REPORT RECALL STATEMENT
      ***
      REPORTS(REPORT) = "SORT USC.PROSPECT "
      REPORTS(REPORT) = REPORTS: " WITH STATE "
      IF STATES = "ALL" THEN GO 810
      X = 1
      LOOP
        STATE = STATES<X>
      UNTIL STATE = '' DO
        REPORTS(REPORT) = REPORTS(REPORT):' "':STATE: '" '
        X = X + 1
      REPEAT
810   REPORTS(REPORT) = REPORTS(REPORT):" AND WITH COUNTY-CODE "
      IF COUNTIES = "ALL" THEN GO 820
      X = 1
      LOOP
▒       COUNTY = COUNTIES<X>
```

```
      UNTIL COUNTY = '' DO
        REPORTS(REPORT) = REPORTS(REPORT): ' "':COUNTY: '" '
        X = X + 1
      REPEAT
820   REPORTS(REPORT) = REPORTS(REPORT):' AND WITH SALES >= "':SALES: '"'
830   REPORTS(REPORT) = REPORTS(REPORT):" AND WITH "
      IF SIC.SELECT = "P" THEN
        REPORTS(REPORT) = REPORTS(REPORT):" PRIMARY-SIC "
      END ELSE
        REPORTS(REPORT) = REPORTS(REPORT):" SIC-CODES "
      END
      IF SIC.CODES = 'ALL' THEN GO 840
      X = 1
      LOOP
        FROMSIC = SIC.CODES<X,1>
        TOSIC   = SIC.CODES<X,2>
      UNTIL FROMSIC = '' DO
        IF X > 1 THEN REPORTS(REPORT) = REPORTS(REPORT):" OR "
        REPORTS(REPORT) = REPORTS(REPORT):' >= "':FROMSIC:'"'
        REPORTS(REPORT) = REPORTS(REPORT):' AND <= "':TOSIC:'"'
        X = X + 1
      REPEAT
840   ***
      BEGIN CASE
        CASE SORT.BY = 1
          REPORTS(REPORT) = REPORTS(REPORT):" BY ZIP BY COMPANY "
        CASE SORT.BY = 2
          REPORTS(REPORT) = REPORTS(REPORT):" BY COMPANY "
        CASE SORT.BY = 3
          REPORTS(REPORT) = REPORTS(REPORT):" BY PRIMARY-SIC BY COMPANY "
      END CASE
      REPORTS(REPORT) = REPORTS(REPORT):" COMPANY OFFICER TELEPHONE -->
                DMI-LINE SALES "
      REPORTS(REPORT) = REPORTS(REPORT):' HEADING " ':" 'T' -->
                                        D&B Prospect Report for - ":NAME
      REPORTS(REPORT) = REPORTS(REPORT):"                        Page 'P' -->
                'C' 'L' 'L' ":TITLE:" 'C' 'L' 'L' ":'"'
      REPORTS(REPORT) = REPORTS(REPORT): "DBL-SPC ID-SUPP LPTR "
      ***
900   *** BUILD ANOTHER REPORT?
      ***
      REPORT = REPORT + 1
      IF REPORT > 10 THEN GO 1000
910   PRINT @(5,23):"Do you wish to generate another report (Y/N)? -->
                #  ":@(51,23):; INPUT RSP,1:_
      PRINT @(51,23):SPACE(2):
      PRINT @(51,23):RSP:
      IF RSP = 'X' THEN
        REPORT = REPORT - 1
        IF REPORT < 1 THEN REPORT = 1
        GO 100
      END
      IF RSP = 'Y' THEN GO 100
      IF RSP # 'N' THEN GO 910
      ***
1000  *** EXECUTE REPORTS
      ***
```

```
      PRINT @(-1):@(10,0):"Now processing reports....."
      REPORT = 1
      LOOP
        STATEMENT = REPORTS(REPORT)
      UNTIL STATEMENT = '' DO
        PRINT;PRINT
        PRINT STATEMENT
        EXECUTE STATEMENT
        REPORT = REPORT + 1
      REPEAT
      STOP
      END

SUBROUTINE GET.CRITERIA
      COMMON STATES,COUNTIES,SALES,SIC.CODES,SIC.SELECT,SORT.BY
      COMMON TITLE,NAME,FLAG
      FLAG = ''
      ***
200   *** GET STATES
      ***
      X = 1
      STATES = ''
      PRINT @(5,2):"Enter State code - ":
202   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 799
      END
      IF STATE = '' THEN GO 300
      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 202
      END
      PRINT @(21+(X*3),2):STATE 'L#3':
      IF STATE = 'ALL' THEN STATES = 'ALL'; GO 300
      IF NOT(STATE MATCHES '2A') THEN GO 202
      STATES = REPLACE(STATES,X,0,0,STATE)
      X = X + 1
      IF X > 12 THEN GO 300
      GO 202
      ***
300   *** GET COUNTY CODE
      ***
      ...
      ...
      ...
      ***
400   *** GET MINIMUM SALES VOLUME
      ***
      ...
      ...
```

```
      ...
      ***
500   *** GET SIC CODE RANGES
      ***
      ...
      ...
      ...
      ***
600   *** SELECT ON PRIMARY SIC CODES OR ALL SIC CODES
      ***
      ...
      ...
      ...
      ***
700   *** GET FREE FORM HEADING
      ***
      ...
      ...
      ...
      ***
730   *** GET OPERATORS NAME
      ***
      ...
      ...
      ...
      ***
750   *** GET SORT CRITERIA
      ***
      PRINT @(5,21):"Sort by 1) Zip 2) Company name 3) SIC code # ": -->
            @(51,21):; INPUT SORT.BY,1:_
      PRINT @(51,21):SPACE(1):
      IF SORT.BY = 'X' OR SORT.BY = 'END' THEN FLAG = 'X'; GO 799
      IF SORT.BY = '' THEN GO 750
      IF SORT.BY = '-' THEN GO 730
      IF SORT.BY < 1 OR SORT.BY > 3 THEN GO 750
      PRINT @(51,21):SORT.BY 'R#1':
      ***
799   *** RETURN
      ***
      RETURN
```

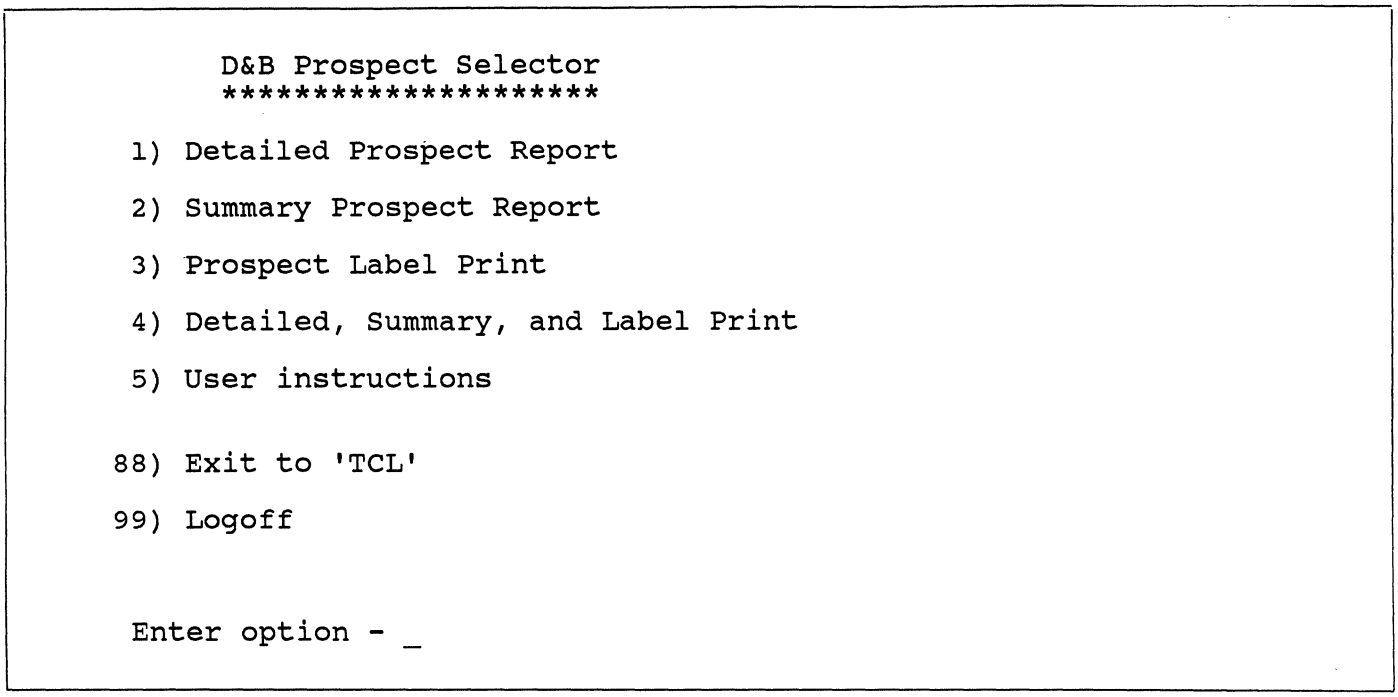


Figure A. Sample Menu

NOTES

APPENDIX A

BASIC COMPILER ERROR MESSAGES

This appendix presents a list of the error messages which may occur as a result of compiling a BASIC program. The error number and message are printed in bold-faced type. The cause and explanation (if needed) are in regular type.

ERROR#	ERROR MESSAGE AND CAUSE
B100	**Compilation aborted; no object code produced** Compilation errors present.
B101	**Ambiguous ELSE clause** Statement with optional ELSE clause used in single-line IF statement.
B102	**Bad statement** Unrecognizable statement.
B103	**Label 'label' is missing** Label indicated by GOTO or GOSUB was not found or MATREAD statement uses simple variable instead of dimensioned array.
B104	**Label 'label' is doubly defined** More than one statement was found beginning with the same label.
B105	**'variable' has not been dimensioned** Subscripted variable was not dimensioned.
B106	**'variable' has been dimensioned and used without subscripts** Dimensioned array used without subscripts.
B107	**LOOP statement nested too deep** LOOP statement nested within too many outer LOOP statements.
B109	**Variable missing in NEXT statement** Iteration variable from FOR statement is missing in NEXT statement.
B110	**END statement missing** The END statment is missing in a multi-line IF statement, or invalid use of END statement.
B111	**EXIT used outside of LOOP statement** EXIT statement not between LOOP and REPEAT
B112	**REPEAT missing in LOOP statement** REPEAT is missing in a LOOP statement.

ERROR#	ERROR MESSAGE AND CAUSE
B113	**Terminator missing** Garbage is following a legal statement, or a quote is missing.
B114	**Maximum number of variables exceeded** More than 3200 variables (including array elements) used.
B115	**Label 'label' is used before the EQUATE stmt.** The symbol is referenced before it has been defined.
B116	**Label 'label' is used before the COMMON stmt.** A common variable is referenced before it is put in common.
B117	**Label 'label' is missing a subscript list** A dimensioned array is referenced without a subscript list.
B118	**Label 'label' is the object of an EQUATE statement and is missing** Variable after TO clause in EQUATE statement has not been declared or used elsewhere in program.
B119	**Warning - precision value out of range - ignored** A precision not in the range of 0-9.
B120	**Warning - multiple precision statements - ignored** More than one precision statement was found.
B121	**Label 'label' is a constant and cannot be written into** The symbol after EQUATE has been assigned a constant value (literal number or string), but also is used as the object of an assignment statement. The symbol cannot be both a constant and a variable in the same program.
B122	**Label 'label' is improper type** Expression after TO in EQUATE is illegal.
B124	**Label 'label' 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** Null source item.
B126	**ELSE clause missing** Required ELSE clause missing.
B127	**NEXT missing** NEXT statement is missing in FOR-NEXT loop.

ERROR#	ERROR MESSAGE AND CAUSE
B128	**Item 'name' not found** Object of $INCLUDE or $CHAIN directive is missing.
B129	**Illegal: program name same as dictionary item name** Program file dictionary already contains an item (other than object code pointer) with the same name as the program.
B199	**Source file must have separate DICT and DATA sections** Program file data section defined as same as dictionary section.
B220	**'CSYM' is not a file name or needs a data level** File CSYM not properly defined on the account for cross-reference purposes.

APPENDIX B

BASIC RUN-TIME ERROR MESSAGES

This appendix presents a list of the error messages which may occur as a result of executing a BASIC program. 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.

ERR#	ERROR MESSAGE AND CAUSE
B1	**Runtime abort at line n** Caused by BASIC statement ABORT.
B10	**Variable has not been assigned a value; zero used** An unassigned variable was referenced, so a value of 0 is assumed.
B11	**Tape record truncated to tape record length** An attempt was made to write more onto a tape record than the tape record length. (The record is truncated to tape record length.)
B12	**File has not been opened** File indicated in I/O statement has not been opened via an OPEN statement.
B13	**Null conversion code is illegal; no conversion done** A string variable that should have a value is actually null.
B14	**Bad stack descriptor** Number of parameters in CALL statement different from that in SUBROUTINE statement, or file variable used as an operand.
B15	**Illegal opcode: n** Program code contains garbage.
B16	**Non-numeric data when numeric required; zero used** A non-numeric string was encountered when a number was required, so a value of 0 is assumed.
B17	**Array subscript out of range** Array subscript is less than 1 or greater than value in DIM statement.
B18	**Attribute number less than -1 is illegal** Attribute less than -1 is specified in READV or WRITEV statement.
B19	**Illegal pattern** Illegal pattern used with MATCH or MATCHES operator.

ERR#	ERROR MESSAGE AND CAUSE
B20	**COL1 or COL2 used prior to executing a FIELD stmt; zero used** COL1 or COL2 function used before FIELD function, so a value of 0 is assumed.
B22	**Illegal value for STORAGE statement** STORAGE parameter less than 10 or not a multiple of 10.
B23	**Program 'name' must be recompiled** Object code not compatible with current operating system.
B24	**Divide by zero illegal; zero used** Division by zero attempted, so a value of 0 is assumed.
B25	**Program 'name' has not been cataloged** A subroutine specified in a CALL statement was not found in the program file and was not cataloged or not found in the file specified by the M/DICT catalog item.
B26	**'UNLOCK n' attempted before LOCK** An attempt was made a unlock a lock which had not been locked by the program.
B27	**RETURN executed with no GOSUB** RETURN statement executed prior to GOSUB.
B28	**Not enough work space** Not enough user work space to hold all data values.
B30	**Array size mismatch** Array sizes do not match in MAT Copy statement, or in CALL and SUBROUTINE statements.
B31	**Stack overflow** The program has attempted to call too many nested subroutines.
B32	**Page heading exceeds maximum of 1400 characters** Page heading is too long.
B33	Precision declared in subprogram 'name' is different from **that declared in the mainline program** Precision must be the same in both calling programs and subroutines.
B34	**File variable used where string expression expected** Illegal use of file variable.
B41	**Lock number greater than 47** LOCK parameter not in range 0-47.

ERR#	ERROR MESSAGE AND CAUSE
B209	**File is update protected.** An attempt was made to update a file that has an update lock.
B210	**File is access protected.** An attempt was made to read a file that has a retrieval lock.

APPENDIX C

LIST OF ASCII CODES

This appendix presents a list of ASCII codes for decimal number values from 0 through 255 (see DECIMAL column). The hexadecimal equivalent value and ASCII character generated are given (see HEX and CHARACTER columns).

Note that decimal values 0-31 are assigned as non-printable functions. Decimal values 1-26 may be specified by control key sequences (see TERMINAL KEY column). A "control key sequence" is entered by holding down the <CTRL> key while pressing a second key (e.g., <CTRL>A). Some of the non-printable characters have a special use in Ultimate systems (see SPECIAL USE IN ULTIMATE column).

Decimal values above 127 (Hex '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, in the ULTIMATE system, special file structure functions and control key sequences have been assigned to decimal values 251 through 255 (Hex 'FB' through 'FF').

DECIMAL	HEX	CHARACTER	SPECIAL USE IN ULTIMATE	TERMINAL KEY
00	00	NUL		
01	01	SOH		<CTRL>A
02	02	STX		<CTRL>B
03	03	ETX		<CTRL>C
04	04	EOT		<CTRL>D
05	05	ENQ		<CTRL>E
06	06	ACK		<CTRL>F
07	07	BEL	Bell on terminal	<CTRL>G
08	08	BS	Backspace	<CTRL>H
09	09	HT	Tab	<CTRL>I
10	0A	LF	Line feed on terminal	<CTRL>J
11	0B	VT		<CTRL>K
12	0C	FF		<CTRL>L
13	0D	CR	Carriage return on terminal	<CTRL>M
14	0E	SO		<CTRL>N
15	0F	SI		<CTRL>O
16	10	DLE		<CTRL>P
17	11	DC1		<CTRL>Q
18	12	DC2	Retype entire line	<CTRL>R
19	13	DC3		<CTRL>S
20	14	DC4		<CTRL>T
21	15	NAK		<CTRL>U
22	16	SYN		<CTRL>V
23	17	ETB		<CTRL>W
24	18	CAN	Cancel line on terminal	<CTRL>X
25	19	EM		<CTRL>Y
26	1A	SUB		<CTRL>Z

DECIMAL	HEX	CHARACTER	SPECIAL USE IN ULTIMATE	TERMINAL KEY
27	1B	ESC		
28	1C	FS		
29	1D	GS		
30	1E	RS		
31	1F	US		
32	20	SPACE		
33	21	!		
34	22	"		
35	23	#		
36	24	$		
37	25	%		
38	26	&		
39	27	'		
40	28	(		
41	29	)		
42	2A	*		
43	2B	+		
44	2C	'		
45	2D	-		
46	2E	.		
47	2F	/		
48	30	0		
49	31	1		
50	32	2		
51	33	3		
52	34	4		
53	35	5		
54	36	6		
55	37	7		
56	38	8		
57	39	9		
58	3A	:		
59	3B	;		
60	3C	<		
61	3D	=		
62	3E	>		
63	3F	?		
64	40	@		
65	41	A		
66	42	B		
67	43	C		
68	44	D		
69	45	E		
70	46	F		
71	47	G		
72	48	H		
73	49	I		
74	4A	J		
75	4B	K		
76	4C	L		
77	4D	M		
78	4E	N		
79	4F	O		

DECIMAL	HEX	CHARACTER	SPECIAL USE IN ULTIMATE	TERMINAL KEY
80	50	P		
81	51	Q		
82	52	R		
83	53	S		
84	54	T		
85	55	U		
86	56	V		
87	57	W		
88	58	X		
89	59	Y		
90	5A	Z		
91	5B	[		
92	5C	\		
93	5D	]		
94	5E	^		
95	5F			
96	60	'		
97	61	a		
98	62	b		
99	63	c		
100	64	d		
101	65	e		
102	66	f		
103	67	g		
104	68	h		
105	69	i		
106	6A	j		
107	6B	k		
108	6C	l		
109	6D	m		
110	6E	n		
111	6F	o		
112	70	p		
113	71	q		
114	72	r		
115	73	s		
116	74	t		
117	75	u		
118	76	v		
119	77	w		
120	78	x		
121	79	y		
122	7A	z		
123	7B	{		
124	7C	:		
125	7D	}		
126	7E	~		
127	7F	DEL		
128	80			
129	81			

DECIMAL	HEX	CHARACTER	SPECIAL USE IN ULTIMATE	TERMINAL KEY
130	82			
131	83			
132	84			
133	85			
134	86			
135	87			
136	88			
137	89			
138	8A			
139	8B			
140	8C			
141	8D			
142	8E			
143	8F			
144	90			
145	91			
146	92			
147	93			
148	94			
149	95			
150	96			
151	97			
152	98			
153	99			
154	9A			
155	9B			
156	9C			
157	9D			
158	9E			
159	9F			
160	A0			
161	A1			
162	A2			
163	A3			
164	A4			
165	A5			
166	A6			
167	A7			
168	A8			
169	A9			
170	AA			
171	AB			
172	AC			
173	AD			
174	AE			
175	AF			
176	B0			
177	B1			
178	B2			
179	B3			

DECIMAL	HEX	CHARACTER	SPECIAL USE IN ULTIMATE	TERMINAL KEY
180	B4			
181	B5			
182	B6			
183	B7			
184	B8			
185	B9			
186	BA			
187	BB			
188	BC			
189	BD			
190	BE			
191	BF			
192	C0			
193	C1			
194	C2			
195	C3			
196	C4			
197	C5			
198	C6			
199	C7			
200	C8			
201	C9			
202	CA			
203	CB			
204	CC			
205	CD			
206	CE			
207	CF			
208	D0			
209	D1			
210	D2			
211	D3			
212	D4			
213	D5			
214	D6			
215	D7			
216	D8			
217	D9			
218	DA			
219	DB			
220	DC			
221	DD			
222	DE			
223	DF			
224	E0			
225	E1			
226	E2			
227	E3			
228	E4			
229	E5			

DECIMAL	HEX	CHARACTER	SPECIAL USE IN ULTIMATE	TERMINAL KEY
230	E6			
231	E7			
232	E8			
233	E9			
234	EA			
235	EB			
236	EC			
237	ED			
238	EE			
239	EF			
240	F0			
241	F1			
242	F2			
243	F3			
244	F4			
245	F5			
246	F6			
247	F7			
248	F8			
249	F9			
250	FA			
251	FB	SB	Start buffer	<CTRL>[
252	FC	SVM	Subvalue Mark	<CTRL>\
253	FD	VM	Value Mark	<CTRL>]
254	FE	AM	Attribute Mark	<CTRL>^
255	FF	SM	Segment Mark	<CTRL>_

APPENDIX D

SUMMARY OF THE BASIC DEBUGGER COMMANDS

The following is a summary of all the BASIC DEBUGGER commands and their descriptions.

Bx	Set breakpoint condition table where 'x' is a simple logical expression, which may be composed of < (less than), > (greater than), = (equal to), # (not equal to), & (and), and the special operator $ (line number).
D	Display breakpoint and trace tables.
DEBUG	Escape to Assembly Debugger.
DE	Short form of DEBUG.
En	Execute 'n' instructions. E <CR> turns mode off.
END	End execution of BASIC program and return to TCL.
G	Proceed from breakpoint.
Gn	Go to line n.
K	Kill all breakpoint conditions in table set by 'B' command.
Kn	Kill breakpoint condition 'n' where 'n' is the breakpoint number from 1-4.
L{m-{n}}	Display program source lines.
LP	All output forced to printer; reverses status each time LP is selected.
N	Stop on every execution break.
Nn	Continue thru n execution breaks before stopping.
OFF	Log off.
P	Inhibit/enable BASIC program output.
PC	Printer close - output to spooler.
R	Pop return stack.
S	Display return stack.
T	Turn breakpoint trace table off/on.

T{/}v	Set variable 'v' in trace breakpoint table.
U	Remove all breakpoint trace table variables set by 'T' command.
U{/}v	Remove breakpoint trace variable 'v' from table.
Z	Request symbol table.
?	Print current program name and line number; verify object.
$	Same as "?".
/v	Print value of a variable 'v'.
/m(x)	Print value of a point 'x' in array 'm'.
/m(x,y)	Print value of point 'x,y' in array 'm'.
/m	Print the entire array where 'm' is the array.
/*	Dump entire symbol table.
[x,y]	Set string window where 'x' equals the start of the string and 'y' equals the length of the string. This command effects all Debugger output of variables and has no effect on input.
[	Remove string window (setting string length to zero has the same effect).
line feed	Same as G <CR>.

NOTES:

1. An equals sign (=) prints out after the printing of a variable in any slash command. The value of the variable may be changed at this point.

2. A carriage return <CR> terminates all commands.

3. Pressing the BREAK key breaks to the BASIC Debugger from BASIC program at end of line.

4. The BASIC Debugger prompts with '*'.

APPENDIX E

BASIC DEBUGGER MESSAGES

The following informative, warning or error messages are used by the BASIC DEBUGGER.

*E n	Single step execution break at line# 'n'.
*Bm n	Table breakpoint at line# 'n'; 'm' is breakpoint number.
*I n	Break key execution break at line# 'n'.
*v=x	Value of variable 'v' at execution break.
Cmnd?	Command not recognized.
Nstat	Statement number out of range of program.
Sym not fnd	Symbol not found in table.
Unassigned var	Variable not assigned a value.
Stk emp	The subroutine return stack is empty.
Ilstk	Illegal subroutine return stack format.
Tbl full	Trace or break table full.
Illgl sym	Illegal symbol.
Not in tbl	Not in trace break table.
No source	No source code found for program.

INDEX

ULT
THE ULTIMATE CORP.
717 RIDGEDALE AVENUE, EAST HANOVER, NEW JERSEY 07936
(201) 887-9222 TWX 710-996-5862 Telecopier (201) 887-6139