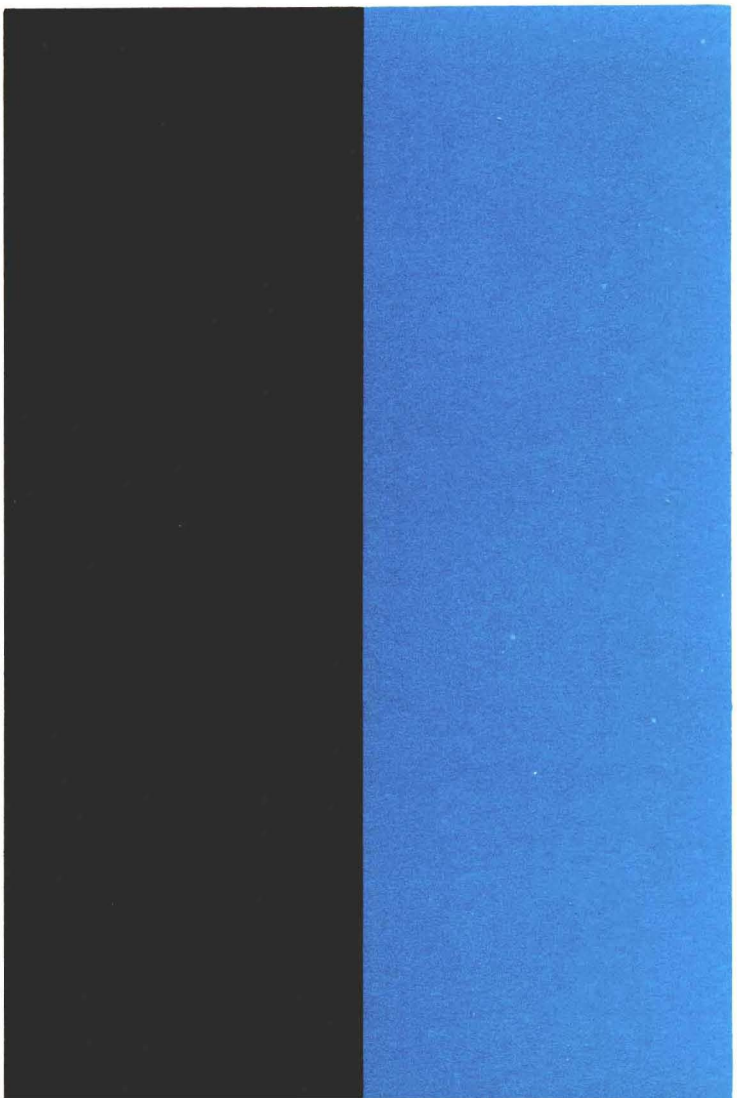
SPERRY RAND

UNIVAC 1100 SERIES

BASIC



PROGRAMMER REFERENCE

UP-7925

ACKNOWLEDGMENT

The authors of this manual and the UBASIC Processor would like to express their appreciation to the Systems Programming Group of the University of Maryland who supplied the basis for this project.

UBASIC is an extension of the BASIC Processor that was developed by the University of Maryland.

This document contains the latest information available at the time of publication. However, the Univac Division reserves the right to modify or revise its contents. To ensure that you have the most recent information, contact your local Univac Representative.

UNIVAC is a registered trademark of the Sperry Rand Corporation.

PRINTED IN U.S.A.

PAGE STATUS SUMMARY

ISSUE: UP-7925 Original

The following table lists the status of each page in this document and indicates the update package (if applicable).

Section	Page Number	Page Status	Update Package
Cover	None	Original	
Disclaimer	None	Original	
PSS	1	Original	
Acknowledgment	1	Original	
Preface	1	Original	
Contents	1 thru 4	Original	
1	1	Original	
2	1 thru 10	Original	
3	1 thru 47	Original	
4	1 thru 7	Original	
Appendix A	A1 thru A3	Original	
Appendix B	B1	Original	
Appendix C	C1 thru C6	Original	
Appendix D	D1 thru D10	Original	

Section	Page Number	Page Status	Update Package

PREFACE

This manual describes the UNIVAC 1100 Series BASIC (UBASIC) programming language. UBASIC is an extended version of the BASIC (Beginners All-purpose Symbolic Instruction Code) language developed at Dartmouth College. These extensions make UBASIC an extremely powerful language for advanced users as well as a simple language for beginners.

This manual is designed as a reference manual, not as a self-instructing guide. Therefore a fundamental knowledge of BASIC is assumed.

Only the rudimentary knowledge of the use of the UNIVAC 1100 Series Operating System contained in Appendix A Operating Procedures is necessary for the beginning user. More sophisticated use of UBASIC will require increasing levels of understanding of the Operating System.

CONTENTS

1. INTRODUCTION

The UNIVAC 1100 Series BASIC Processor (UBASIC) is designed primarily for conversational use. Written with the beginning programmer in mind, the UBASIC language consists of instructions which are largely self-explanatory, and the syntax is closely related to normal English. The UBASIC Processor can be used for batch processing, and is powerful enough to solve a large class of problems.

This document provides information concerning the UBASIC language and system as implemented by Univac for the 1100 series system operating under the Executive.

The UBASIC Processor reads statements (or lines) and immediately examines each one for syntactic correctness. On remote devices in conversational (demand) mode, syntactic errors in a line cause a diagnostic message to be printed immediately, so that such errors can be corrected before the next line is accepted. When a complete program has been processed, then a RUN command may be given, at which time the program is translated to machine language (compiled), and execution of the program is initiated.

There are several examples of UBASIC programs in Appendix D, and operation procedures are described in Appendix A. The simple examples and the operating procedures should be studied before reading this manual (see Appendices A and D).

2. PRELIMINARY DEFINITIONS

In order to understand the material in this reference manual, it is necessary to define certain terms which may appear repeatedly.

2.1. PROGRAM

A program under the UBASIC system consists mainly of a series of statements in the UBASIC language which perform algebraic and/or string manipulations. Each program is terminated by the END statement. The simplest, but not practical, UBASIC program would be as follows:

```
n END
```

where n is any statement number less than 100000. A UBASIC program may also consist of all the system commands as well as the individual source language (UBASIC) programs. This is especially true with non-conversational runs, where all system commands must be included in a deck of punched cards.

2.2. STATEMENTS

A UBASIC statement is a single line of information consisting of a series of characters. The line is terminated in various ways, depending on the device which is used for transmission of the information. The sequence of statements in the UBASIC language begin and proceed as follows:

(1) A statement number (unsigned and less than 100000),
(2) An instruction word,
(3) An expression(s) needed for instructions,
(4) Comments, and
(5) A statement terminator.

There are also UBASIC system control statements such as RUN which are not preceded by a number. The word *statement* used alone, however, will refer only to UBASIC language statements and *system commands* will refer to UBASIC control statements.

When submitted at the central computer site or from a high speed remote device, each statement consists of all 80 columns of a punched card.

2.3. INSTRUCTION WORDS

The instruction word in UBASIC is the heart of every statement, signifying what the statement does in a program, and providing the basis for classifying the statement. UBASIC statements as determined by their instruction words fall into six categories:

(1) Input and output of data
(2) Algebraic and string manipulations
(3) Matrix operations
(4) Logical tests
(5) Control transfers
(6) UBASIC directives

The individual UBASIC statements will be discussed in detail in Section 3.

2.4. CONSTANTS

2.4.1. ARITHMETIC CONSTANTS

An arithmetic constant in UBASIC represents a numeric value and may take several forms. Any number of digits may be used in a constant (only 8 digits of accuracy are available) as long as the magnitude of the constant is between 10^{-39} and 10^{38}. A constant may also be represented in exponential form which consists of a signed integer or decimal number followed by the letter E and an integer power of ten exponent. The following are permissible arithmetic constants in UBASIC:

12E+2 (equivalent to 12E2 or 1200)
1.32
59
−.3233333333
+5.2
12.E−2 (equivalent to .12)
10E5 (equivalent to 100000)
−0E120 (equivalent to 0)
0.0E−400 (equivalent to 0)

UBASIC also accepts the percent sign in constants and interprets it as E−2. Thus, 2% is equivalent to 2E−2 or .02.

NOTE: 2E-2% is not acceptable since the expression 2E−2E−2 is not valid.

UBASIC recognizes no difference between fixed point and floating point constants. All numbers are processed as floating point, whether they have a decimal point or not. Thus the numbers 2 and 2.0 will be equivalent.

2.4.2. STRING CONSTANTS

A UBASIC string constant is a sequence of zero or more characters. In a program a string constant is represented by a character string between single quotes, with two quotes in the body of the string representing one quote in the constant. Thus, the character string AB'CD would appear as the constant 'AB''CD'. When used as data or input items, the first and last quotes and the doubling of internal quotes may be omitted unless the omission would create an ambiguity (i.e., a comma in the string or a quote as the first character would require it to be enclosed in quotes).

2.5. VARIABLES

A variable represents a value which may be changed using certain UBASIC instructions. The value of a variable may be changed at any time, and the variable will represent that value until changed again. In UBASIC, variables have two types:

(1) simple and
(2) subscripted;

and two modes:

(1) string and
(2) arithmetic.

A simple variable is a letter, or a letter followed by a digit, and may be designated as a string variable (one which has as its value a character string) by a dollar sign ($) following it.

Arrays, which are groups of numbers in the form of a matrix or a list (a vector), may also be referred to by a variable. An array variable, consisting of a letter or a letter followed by a dollar sign, may either refer to a single variable of that name or to a whole array, depending on the context of the statement in which it appears. Subscripts are used to reference individual elements of an array. An array name followed by a subscript(s) refers to a single value, and is referred to as a subscripted variable. An algebraic subscripted variable may have one to four subscripts, but a string subscripted variable (with a dollar sign after the array name) may only have one. The following are examples of valid UBASIC variables:

(a) A (a simple algebraic variable and/or array name)
(b) A1 (a simple algebraic variable)
(c) B$ (a simple string variable and/or string array name)
(d) A(1) (a singly subscripted algebraic variable)
(e) B(J,2) (a doubly subscripted algebraic variable)
(f) A$(K) (a subscripted string variable)

NOTE: A function name (e.g., FNA) is also used as a variable within the range of a multi-line defined function, and represents the value of the function (see 3.10.)

2.6. OPERATIONS

There are two primary types of operators in UBASIC as follows:

(1) relational and
(2) algebraic.

A **relational operator** is one of the following:

=	(equals)
<	(is less than)
>	(is greater than)
<> or ><	(is not equal to)
<= or =<	(is less than or equal to)
>= or =>	(is greater than or equal to)

These relational operators can occur only in the IF statement between two expressions, as will be explained in the description of that statement. Between two algebraic expressions, they have their normal meaning in the ordering of real numbers. Between string expressions, they apply to alphabetical order (< meaning "come before") or, more precisely, alphanumeric order as shown in the list given with the CHANGE statement description (see 3.14.).

An **algebraic operator** is one of the following:

+	(unary plus)
↑ (Δ onsite) or **	(exponentiation)
–	(unary minus)
* or /	(multiplication, division)
+ or –	(addition, subtraction)

These operators are used in algebraic expressions and are evalated in the order in which they appear in the preceding table. Operators with the same precedence are evaluated from left to right. Thus, –2**2+3*3 equals (–(2**2))+(3*3) equals 5. Parentheses may be used to override the above order. The order of precedence with unary operators and exponentiation depends on the form of the expression. If the unary operator is needed to evaluate the exponent, it is used first.

For example, in the expression A**–2, the – is evaluated first and the expression becomes A**(–2).

2.7. FUNCTIONS

In UBASIC there are two types of functions:

(1) system defined and
(2) program defined (DEF functions).

The system defined functions are divided into five classes. These classes are:

(a) Numeric Functions
(b) String Functions
(c) Logical Functions
(d) Relational Functions
(e) Special Functions

2.7.1. NUMERIC FUNCTIONS

These functions may be used in any algebraic expression and return a numeric value as a result.

Several of these functions do not require an argument. In such cases if an argument is included, it is ignored.

ABS(E) – This function returns the absolute value of the expression E.

ATN(E) – This function returns the arctangent in radians of E.

CLK or CLK(E) – This function returns the time of day in military hours, minutes, and seconds. This function will ignore the argument E, if it is included.

Example: If the statement "20 PRINT CLK" was executed at 2:15:15 P.M., then the value 141515 would be returned.

CNT (String 1, String 2) – This function which matches the substrings in string 1 with string 2 returns a count of the number of substrings in string 1 which match string 2. If no match is found, zero is returned.

COL or COL(E) – The next print position of the current line is returned by this function. When COL(E) is used the next print position of file E is returned.

COS(E) – This function returns the cosine of E (E in radians).

COT(E) – This function returns the cotangent of E (E in radians).

CSF – The status code of the last EXEC statement executed is returned by this function. These status codes are described in the *UNIVAC 1100 Series Operating System Programmer Reference, UP-4144* (current version).

DAT or DAT(E) – This function returns the current date in month, day, and year form. This function will ignore the argument E, if it is included.

Example: If the statement "20 PRINT DAT" was executed on March 15, 1971, then the value 31571 would be returned.

DET – The value of the determinant of the most recently inverted matrix is returned by this function. The value zero would be returned, if the last attempt to invert a matrix failed because it was singular.

DIG(E) – The digital part of E, where E is to be expressed in scientific notation, is returned by this function. (See the XPT function.)

$$1 \leqslant | \text{ DIG(E) } | < 10 \quad \text{or} \quad \text{DIG(E)} = 0 \quad \text{if} \quad E = 0$$

Example: 1.4372 is returned for DIG(143.72), since 143.72 in scientific notation equals 1.4372×10^2.

EXP(E) – This function returns e raised to the power E, where e is approximately 2.7182818.

FRP(E) – The fractional part of E is returned by this function.

Example: .443 is returned for FRP (852.443).

GET (String, Character) – This function returns, from the string specified, the FIELDATA equivalent of the character indicated by the character field.

INP(E) – The integer part of E is returned by this function.

Example: 456 is returned for INP (456.234).

INT(E) – The greatest integer which is less than or equal to E is returned by this function.

LEN (String) – The number of characters currently in the string specified is returned by this function.

LGT(E) – This function returns log to the base 10 of E.

LOG(E) – This function returns log to the base e of E.

MAX(E1,E2) – This function returns the greater of the values E1 and E2.

MIN(E1,E2) – This function returns the lesser of the values E1 and E2.

MOD(E1,E2) – This function returns E1 modulo E2 (remainder of E1/E2).

Example: 1 is returned for MOD(5,2).

MXL (String) – The maximum length of the string specified is returned by this function. If a string constant is specified zero is returned.

NUM – This function returns the number of items read in on the last MAT INPUT statement.

RND or RND(E) – This function returns a random number, $0 < \text{RND} < 1$ (uniformly distributed).

SEP (String 1, String 2,E) – This function returns the character position within string 1 of the first character of the substring which matches string 2. E determines which character position of string 1 is used as the starting position. If no match is found zero is returned.

SER (String 1, String 2) – This function returns the character position within string 1 of the first character of the substring which matches string 2. If no match is found zero is returned.

SGN(E) – This function returns 0 if E is zero, –1 if E is negative or, +1 if I is positive.

SIN(E) – This function returns the sine of expression E (E in radians).

STD (String, Vector) – This function stores the characters of the specified string into the vector indicated starting at vector (1). The number of words required for the storage is returned. The remainder of the last word is blank filled if the character string is not a multiple of six. The data is not converted, but merely moved six character per word from the string to the vector.

SQR(E) – This function returns the square root of E.

TAN(E) – This function returns the tangent of E (E in radians).

TIM or TIM(E) – The execution time in seconds as measured from the time of the last RUN or RUNNH command is returned by this function. E is ignored if specified.

TIS or TIS(E) – The time of day in milliseconds since midnight is returned by this function. E if specified is ignored.

XPT(E) – The exponent part of E is returned by this function, when E is represented in scientific notation. (See the DIG function.)

Example: XPT (3471.88) would return a value of three since 3471.88 in scientific notation equal 3.47188×10^3.

VAL (String) – The numeric value of the constant in the string specified is returned.

Example: The number 4.34 is returned for VAL ('4.34').

VAS (String, E1, E2) – The numeric value of the constant in the character positions E1 through E2 is returned by this function.

2.7.2. STRING FUNCTIONS

The following functions can occur only in LET statements and cause the assignment of string values to the string variable on the left of the equals sign. Character positions within strings begin with one. Strings specified as arguments in calls to these functions are left unaltered. As in the case of the normal assign statement, the keyword LET may be omitted.

n LET String 1 = CAT$ (String 2, String 3) — The CAT$ function concatenates string 2 and string 3 and stores the result in string 1.

n LET String 1 = DTS$ (Vector,E) — This function converts E words stored in the vector specified into a six bit character string and stores this in string 1. The data is not converted, but merely moved six characters per word from the vector to the string.

n LET String 1 = PAD$ (String 2,E) — This function adjusts string 2 to E length and stores the result, blank padded on the right, if necessary, in string 1.

n LET String 1 = TRM$ (String 2) — This function deletes all blanks from the right of string 2 and stores the result in string 1.

n LET String 1 = EXT$ (String 2, E1,E2) — This function deletes from string 2 a substring of E2 characters beginning with the character position specified by E1 and stores the result in string 1.

n LET String 1 = CPY$ (String 2, E1,E2) — This function stores a substring of string 2 in string 1. This substring starts with character position E1 and is E2 characters in length.

n LET String 1 = ADD$ (String 2, String 3, E1) — This function inserts string 3 into string 2 and stores the results in string 1. String 3 is inserted into string 2 starting at character position E1+1.

n LET String 1 = PUT$ (String 2, E1,E2) — This function replaces character position E2 in string 2 with a FIELDATA character. This FIELDATA character is obtained by converting the value in E2 to a FIELDATA character. String 2 is then stored in string 1 padded to the right with blanks through position E1—1.

n LET String 1 = STR$(E) — Following the rules of the PRINT statement this function converts E to a string and stores the result in string 1.

2.7.3. LOGICAL FUNCTIONS

The logical functions all return one of two values, 0 for false and 1 for true. The function arguments are left unaltered by these functions.

NOT(E) — Logical negation. This function returns 1 if E is zero, 0 if E is non-zero.

AND(E1,E2) — Logical and. This function returns 1 if both E1 and E2 are non-zero; 0 otherwise.

IOR(E1,E2) — Logical inclusive or. This function returns 1 if E1 or E2, or both E1 and E2 are non-zero; 0 otherwise.

XOR(E1,E2) — Logical exclusive or. This function returns 1 if E1 or E2, but not both E1 and E2 are non-zero; 0 otherwise.

EQV(E1,E2) – Logical equivalence. This function returns 1 if both E1 and E2 are zero, or if both E1 and E2 are non-zero; 0 otherwise.

IMP(E1,E2) – Logical implication. This function returns 1 if E1 and E2 are both non-zero or if E1 is zero; 0 if E1 is non-zero and E2 is zero.

2.7.4. RELATIONAL FUNCTIONS

The relational functions all return one of two values, 0 for false and 1 for true. These functions may be used in algebraic expressions where relational operators are not allowed. The function arguments are left unaltered by these functions.

LSS (E1,E2) – The less than function returns 1 if E1 is less than E2, 0 otherwise.

LEQ (E1,E2) – The less than or equal to function returns 1 if E1 is less than or equal to E2; 0 otherwise.

GTR (E1,E2) – The greater than function returns 1 if E1 is greater than E2; 0 otherwise.

GEQ (E1,E2) – The greater than or equal to function returns 1 if E1 is greater than or equal to E2; 0 otherwise.

EQU (E1,E2) – The equal to function returns 1 if E1 is equal to E2; 0 otherwise.

NEQ (E1,E2) – The not equal to function returns 1 if E1 is not equal to E2, 0 otherwise.

2.7.5. SPECIAL FUNCTIONS

The following functions do not fall into any of the above classes;

FLD (S,N,E) – This function selects N bits of the 36 bit algebraic expression E, starting with bit position S, and returns these bits as a floating point number. Bits of the word E are numbered 0 through 35 from left to right.

Example: 100 A = FLD (0,9,1.0) would set A equal to 129, the result of converting the first nine bits (0201) of, the binary representation of 1.0 (0201400000000) to floating point representation.

INS (S'N'E1,E2) – This function converts the value of E2 to a binary integer, and inserts this value, right justified with leading binary zeros, into a field of N bits starting with bit S of E1. This function can be used as a function or as a CALL statement. The changed value of E1 is returned when INS is used as a function.

TAB(E) – This function sets the next print position in the current print line equal to the arithmetic expression E. This function is only used in the PRINT statement. Print positions are numbered 0 through 71, from left to right.

2.8. EXPRESSIONS

An algebraic expression is any meaningful combination of:

- arithmetic constants,
- variables,
- functions, and
- algebraic operators.

The syntax rules are presented in Appendix C. UBASIC Syntax In Backus Normal Form. A string expression is either a string constant or a string variable.

The two types of variables and constants (string and algebraic) may be combined in any expression only if the string item is an argument to a defined function which may accept such an argument. The following are all valid expressions:

```
A+B/C1*FNA(SGN(2),RND)
C(A(B(FNB(1+LOG(10),FNC,A,B),1.5E+2*A),0))
F*(A-B**-C(F)+(((A))))-314159E-5
```

even though some parentheses may be redundant. The following expression may or may not be valid, depending on how the function FNA is defined:

```
FNA(A$,'ABC',2)
```

2.9. COMMENTS

In order to make programs more understandable, it is often desirable to include comments. For this reason, two methods of commenting are available in UBASIC as follows:

(1) The REM statement (see 3.15.).

(2) If a comment is desired, it may be inserted after the statement by using a master space character to signify the end of the actual statement. Comments are printed, but otherwise ignored by the UBASIC processor.

3. DESCRIPTION OF STATEMENTS

In discussing the various statements in this section, several symbols will be employed to give the general form of some statements. The meaning of the symbols are as follows:

n	a statement number
v	a variable
e	an expression
c	a numeric constant
r	a relational operator
m	a statement number to which control may be transferred
ae	an algebraic expression
sc	a string constant
v$	a string variable
i,j	unsigned integers
sv	a simple variable
a	an array name

3.1. LET

The LET statement is used to assign a value to one or more variables. The value of the given expression is computed and assigned to the specified variable(s).

The general form of the LET statement is as follows:

```
n  LET  v1=v2=...vN=e
```

Example:

```
10  LET  A(12)=B*(K+C1)
15  LET  N$='ABC'
20  LET  A=B=D*E+F
```

The mode of the expression must agree with the mode of the variables to which it is being assigned, i.e., if the expression is arithmetic, the variable(s) must be arithmetic.

Values are assigned from right to left. In other words,

```
10 LET A(N)=N=2
```

will assign the value 2 to N and to A(2), in that order.

The keyword LET may be omitted in the LET statement:

```
30 A=B=SQR(E)
```

is equivalent to:

```
30 LET A=B=SQR(E)
```

Concatenation of two or more string values may be accomplished by combining them in order with plus signs and assigning the result to the specified string variable. Thus, if the current value of A$ is 'CAT' and the current value of B$ is 'NATION', then

```
30 C$ = 'CON' + A$ + 'E' + B$
```

will assign a value of 'CONCATENATION' to C$.

3.2. READ AND DATA

The READ statement is used to assign values to variables. The general form of the READ statement is: follows:

```
n READ  v1,v2,...,vN
```

Example:

```
10 READ A, B, C1, V(1,5)
```

The values to be assigned to the specified variables must be included in a DATA statement.

The general form of the DATA statement is:

```
n DATA  c1,c2,...,cN,sc1,...,scN
```

Example:

```
90 DATA 3.14, 10.E-24, 'ABCD(E', NOPQR, 47
```

Each time a READ statement is encountered, the specified arithmetic variables are assigned the next unread numbers in DATA statements, and the specified string variables are assigned the next unread string quantities in DATA statements.

Program execution will be terminated if a READ is encountered for which there are no DATA items left (see 3.3. RESTORE).

String quantities may or may not be enclosed in quotes in the DATA statement. If commas, leading quotes, or leading or trailing blanks are to be included in the string, quotes must be used to enclose the quantity. A single quote in a string enclosed in quotes is represented by *two* single quotes.

The arithmetic and string constants are independent of each other, and may appear itermixed in a DATA statement.

3.3. RESTORE, RESTORE*, RESTORE$

The RESTORE statement is used in conjunction with the READ and DATA statements to reuse a list of data within a program. When the RESTORE statement (REST for short) is encountered, all data listed in DATA statements is restored for reuse so that the next READ statement will read the first item of data. RESTORE$ should be used to restore string data without affecting numeric data. Similarly, if only numeric data is to be reused, RESTORE* should be employed.

Example:

```
10 RESTORE      (will restore all data)
20 RESTORE*     (will restore only numeric data)
30 RESTORE$     (will restore only string data)
```

3.4. PRINT

The PRINT statement is used to print out the values of algebraic or string expressions.

The general form of the PRINT statement is:

```
n  PRINT  e1,e2;  sc1  e3,...,TAB(ae),...,eN
```

Example:

```
PRINT 'A='A; SQR(B); 'THE'; A$, TAB(40), C
```

Any algebraic expression may be used in a PRINT statement and its value will be printed when the program is run. Similarly, any string appearing in the PRINT statement, either as a string constant or a string variable may be printed. Where a value is printed on a line, this line may be controlled by the use of commas, semicolons, and the TAB function. A print line is normally composed of five print zones, the first four of these zones are 15 characters long with the remaining 12 positions making up the fifth zone, and a comma is interpreted as "skip to the next print zone". A semicolon, however, is interpreted as "don't skip". After an item followed by a semicolon has been printed, printing of subsequent items will begin immediately following it. The lack of both a comma and a semicolon after the last item in a print statement will result in the next item's being printed on the next line by a subsequent

n PRINT

statement which prints a new line consisting of any previous items not yet printed, or a blank line if no such items exist.

The TAB function is provided in UBASIC to allow flexibility in printing. TAB(ae), where ae is an arithmetic expression, results in a skip (forward only) to column INT(ae) in the line being printed. (The 72 columns are numbered from 0 through 71.) If printing has already passed column INT(ae), the TAB is ignored.

The following rules concerning the printing of items should be remembered:

(1) Only 72 characters can be printed on one line; any excess is printed on the next line.

(2) If printing has extended into the last print zone, the next separate item printed will be on the next line, even if a semicolon was used.

(3) Numbers are printed, left justified, according to the following rules:

(a) If the number is an integer with magnitude less than 100,000,000, it is printed out in integer format.

(b) A decimal format is used (up to 8 significant digits) if the number has any fractional part.

(c) A number with magnitude greater than 10^{+8} or less than 10^{-5} is printed out in exponential format – a mantissa following rules a and b and an integer exponent prefixed by the letter E and a sign.

The following examples show printed numbers and their interpretation as follows:

629	an integer
123456	an integer
123456.	a rounded number
6E7	an integer – 60 million
6.0E7	rounded off
.0625	exactly 1/16
1.0E–3	rounded off to 1/1000

NOTE: An algebraic expression may directly follow a string constant in a PRINT statement: follows:

```
10 PRINT 'THE ANSWER IS 'X
```

3.5. GO TO

Within a UBASIC program, statements are executed in order according to their statement numbers from lowest to highest, until a transfer type statement is executed.

The GO TO statement is used to transfer control within a UBASIC program to another line. The general form of the GO TO statement is

```
n GO TO m
```

which causes the program to transfer control to statement m rather than to the statement immediately following statement n.

UBASIC includes another statement reference which is independent of line numbers.

Example:

```
14 GO TO *+4
```

Would transfer control to the fourth line following line 14 after sorting by line number, regardless of how the fourth line was numbered. Similarly

```
25 GO TO *-3
```

would transfer control to the third statement preceding statement 25.

It should be noted that the GO TO statement may not transfer control from within the range of a multi-line defined function to anywhere outside the range of that DEF, or from outside a DEF into its range. It is legal to jump to a DEF statement itself. Program control will resume immediately after the FNEND for that DEF. For such purposes, a DEF statement may be considered part of the main program, outside the range of the function it defines.

3.6. IF

3.6.1. CONTROL TRANSFER

In its simplest form, the IF statement is used to test values in the program, and to transfer control if certain conditions are met. When so used, the IF statement has two parts:

(1) a relation and
(2) a transfer control.

The relation consists of two expressions (both arithmetic or both string) and one of the following relational operators: =, <,<=, =<, >,>=, =>, <>, ><, which are: equal to, less than, less than or equal to, less than or equal to, greater than or equal to, greater than or equal to, not equal to, and not equal to, respectively. The control transfer is of the form GO TO and a statement number or a statement reference, or THEN and a statement number or statement reference (see 3.5. GO TO).

```
10 IF A+B < C**D   THEN 70
```

would transfer control to statement 70 if (A+B) is less than (C**D). Otherwise, control would pass to the next statement.

When strings are compared in the relation, comparison is done alphabetically and according to the list shown with the CHANGE statement (see 3.14.). The relation 'A'<'B' is true, because A comes before B alphabetically. Should the strings be of different lengths, the shorter string is extended with blanks during the comparison to make the strings of equal length. Thus 'A' < 'AA' is true, because 'A' is treated as 'A' for the comparison, and the blank is alphanumerically ahead of the A.

Example:

```
20 IF A$ = 'ABC$F' THEN 35
14 IF 'NRP' > S2$ GO TO 43
15 IF A$ < S$(12) THEN *+5
```

3.6.2. CONDITIONAL STATEMENT EXECUTION

In addition to causing conditional transfer, the IF statement may cause the conditional execution of a statement. This statement may be an input/output statement, such as assign statement, STOP, or another IF.

Example:

```
10 IF A>=0 THEN B=SQR(A)
20 IF A$=B THEN READ A,B,N
30 IF A=1 THEN IF B=2 THEN STOP
```

3.6.3. IF-THEN-ELSE SEQUENCE

Following the conditional statement or transfer of an IF statement as previously described, the word ELSE followed by a statement may be added. This form allows the statement following the THEN to be executed if the condition is true, but executes the ELSE statement if the condition if false. If the executed statement does not transfer control, then the program continues to the next statement.

Example:

```
10 IF A=0 THEN B=0 ELSE B=C/A
20 IF A=B THEN GO TO 100 ELSE C=100
30 IF N=0 THEN GO TO 200 ELSE GO TO 300
```

3.6.4. COMBINING IF STATEMENTS

Any number of IF-THEN and/or IF-THEN-ELSE sequences may be used together. The rule for associating the various clauses in such a combined statement is to associate the inner-most IF-THEN-ELSE and treat the IF-THEN as a left parentheses and the ELSE as a right parentheses in a manner similar to nested parentheses. Thus

IF ... THEN ... IF ... THEN ... ELSE ... ELSE ...

The number of ELSE's in a combined statement must be less than or equal to the number of IF ... THEN's. Thus, every ELSE must have a corresponding IF ... THEN, but the converse is not true. In this example:

IF ... THEN IF ... THEN IF THEN ... ELSE

only the last IF ... THEN has a corresponding ELSE. The next statement in the program becomes the "default ELSE" for the first two IF ... THEN's (i.e., if either of the first two conditions is false, the next program statement will be executed next.)

3.6.5. LOGICAL OPERATORS

Two or more relational conditions may be combined to form a complex relation in an IF statement by the words AND,OR,NOT. These three words have their normal logical meaning. All AND's are sequentially grouped with corresponding operands and then all OR's.

Example:

```
10 IF A>0 AND B>0 THEN C=SQR(A**2+B**2) ELSE C=0
20 IF I >0 OR J=4 THEN 200
```

3.7. ON

The ON statement is used to transfer program control to one of several statements in a program, depending on the value of an expression. The ON statement has two parts: an expression and a control transfer. The expression is any valid arithmetic expression. The control transfer has a THEN or a GO TO followed by two or more statement numbers and/or statement references separated by commas (see 3.5. GO TO).

The general form of the ON statement is as follows:

```
n ON e GO TO m1,m2,m3,... ,mN
```

or

```
n ON e THEN m1,m2,... ,mN
```

where $mi = \begin{cases} n \\ *+k \\ *-L \end{cases}$ for i = 1 to N.

If the value of the expression is 1, control will be transferred to the first statement specified after the THEN or GO TO; if the expression value is 2, control goes to the second statement, etc. The integer part of the expression value is used to determine which statement will receive control.

Example:

```
21 ON A+B*C GO TO 10, 20, *+3, 70
```

If A+B*C has a value of 2.5, control will transfer to the second statement specified, statement 20. If the value of the expression is less than 1 or greater than the number of statements given, an error message will be given.

The special form of the ON statement ON ENDFILE is described in 3.24.6.

3.8. PROGRAM LOOP CONTROL

3.8.1. FOR AND NEXT

The FOR statement, combined with the NEXT statement, is used to allow the programmer to execute statements a number of times without the need to write these statements more than once. Together, a FOR and a NEXT statement form a "loop". The statements in the loop are executed a specified number of times.

The FOR statement consists of an index variable, an intial value which the index variable assumes when the loop is entered, a step-size by which the value of the index variable is increased each time the NEXT statement is encountered, and a final value which causes termination of execution of the statements in the loop when the index variable exceeds this value.

The general form of the FOR and NEXT statements is as follows:

```
n1 FOR sv=e1 TO e2 STEP e3
 .
 .      Statements to be repeatedly executed
 .
n2 NEXT sv
```

Example:

```
10  FOR  X1=1  TO  10  STEP  1
 .
 .
 .
40  NEXT  X1
```

The instruction between statements 10 and 40 would be executed 10 times with X1 assuming the values 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 consecutively.

If the STEP clause is missing, a step-size of 1 is assumed. Negative step-sizes are permitted, as follows:

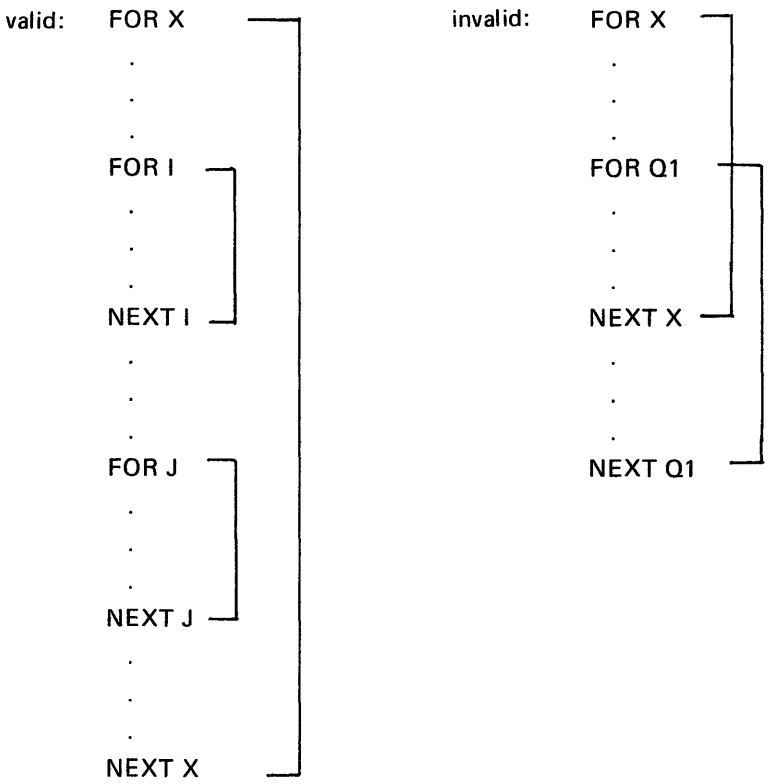
```
10  FOR  A=B/C  TO  N+1  STEP  -1
 .
 .
 .
 .
 .
30  NEXT  A
```

In this case, execution of the loop would continue until A became less than N+1. Note that, if the terminating condition is met upon entry to the loop, the statements in the loop will not be executed. The keyword BY may be used interchangeable with STEP and are exactly equivalent. Note that a single FOR statement may not contain both the STEP and the BY keywords.

The value of the index variable may be used in calculations in the loop. In addition, the value of the index variable may be changed in the loop. If the value of the index variable is changed, the new value replaces the old value and the step-size is added to the new value to obtain the value for the next loop iteration.

Transfer-type statements may be included in a FOR loop (i.e., IF, ON, GO TO) to provide a means of exit from the loop, other than normal completion of the loop. It is permissible to jump out of or into a loop. Caution should be used when jumping into a FOR loop because the values of the initial value, terminal value, and step-size are calculated only if the FOR statement itself is executed; and whatever value the index value has when execution transfers into the loop will be used to determine how many iterations, if any, will be performed.

FOR loops may be nested; that is, one FOR loop may be placed inside another FOR loop. Note that associated FOR and NEXT statements must be completely inside, or completely outside, any other FOR loops. FOR loops may not cross or overlap.



3.8.2. FOR AND IMPLIED NEXT

If the program loop to be performed can be stated in a single statement, a simplified form of the FOR can be used with no NEXT statement involved:

```
n FOR sv=e1 TO e2 STEP e3 STATEMENT
```

where STATEMENT may be an assignment, input/output, or another FOR statement.

Example:

```
10 FOR I = 1 TO 10 A(I) = 0
20 FOR J = 1 TO K STEP 2 PRINT A(J)
```

No NEXT statement is necessary as program control will loop, performing the single statement until the condition of the FOR is satisfied, then perform the next statement in the program sequence.

3.8.3. WHILE

In a manner similar to the FOR with no NEXT, the assignment and input/output statements may be interpreted as a single statement loop under control of a WHILE condition. The difference in the WHILE statement and the FOR statement is that WHILE allows a more general relation to be evaluated (similar to the IF). Its form is follows:

```
n WHILE e1 r1 e2  STEP sv BY e3  STATEMENT
```

Example:

```
10 WHILE A+B<10 STEP A BY 2 C(A/2)=A+B-5
20 WHILE A<1 A=A+SQR(B)
```

Unlike the FOR statement, there is no assumed step size, thus this statement should be used with caution. In the first example, the conditional statement C(A/2)=A+B–5 does not alter the value of the expression A+B. Without the phrase "STEP A BY 2" the expression A+B would always be less than 10 which gives a true condition always. Thus the program would go into a one line loop. In the second example above, although there is no STEP clause, the conditional assignment will alter the value of A (if B is non-zero). Eventually A will exceed one, at which time control will be passed to the next sequential statement.

3.8.4. UNTIL

The UNTIL statement is identical in form to the WHILE statement

```
n UNTIL e1 r1 e2  STEP sv BY e3  STATEMENT
```

but has the opposite effect. That is, the expressions are evaluated and compared giving a true or false condition and the conditional statement is performed if the condition is false. The whole process is then repeated (after the STEP, if applicable). When the condition is evaluated as true, control passes to the next sequential statement.

3.8.5. FOR COMPOUNDED WITH WHILE OR UNTIL

In the FOR statement (with or without the NEXT) the conditional limit (expressed by TO e) may be replaced by WHILE or UNTIL with a relational test (e1 r1 e2) as in the following example:

```
10 FOR I=1 WHILE A(I)<B STEP 1
20 A(I) = A(I--1)+1
30 PRINT I,'A(';I;')=',A(I)
40 NEXT I
```

As in the case of the simple FOR, if the loop is a single statement, it may appear on the line with the FOR without a corresponding NEXT. Also, the STEP may be omitted, in which case it is assumed to be 1.

3.9. STOP

The STOP statement is used to terminate execution of a program. The general form of the STOP statement is:

```
n STOP
```

When the STOP statement is encountered, the program terminates exactly as if the END statement at the end of the program had been encountered. Thus,

```
.                                   .
.                                   .
.                                   .
30 GO TO 999                        30 STOP
.                                   .
.                                   .
.                    and            .
999 END                             999 END
```

both cause termination of program execution. The STOP statement may be used anywhere in the program. However, the END statement must be the last program statement.

3.10. DEF AND FNEND

The Statement DEF allows the programmer to define his own functions. There are two types of DEF-defined functions as follows:

(1) the single-line DEF statement function, and
(2) the multi-line function which begins with a DEF and ends with an FNEND.

The general form of the DEF statement is:

```
n DEF FNg(f1,...,fN) = e
```

or

```
n DEF FNg(f1,...,fN)
```

In the latter case, the DEF statement is followed by a block of statements which carry out the desired function. The last statement of the block must have the form:

```
n  FNEND
```

In the above forms, g may be any single letter of the alphabet, the fi are formal parameters to the function and must be simple non-subscripted variables, and e is any valid expression.

The single-line DEF is used to allow a compact, flexible expression for a formula which is used several times in the program. For example, if the programmer wishes to calculate the sum of the products of two pairs of numbers, he can define a function as follows:

```
10  DEF  FNQ(A,B,C,D1)=A*B+C*D1
```

In order to use this function, the programmer must give (in any arithmetic expression) the function name and a list of expressions to be used as arguments to the function. The function above must have four arguments, and these arguments must be valid arithmetic expressions.

Example:

```
S+N1*(FNQ(A1*B,S-S1,A(1,2),10.721)/2
```

is a valid expression. The system-defined functions may also be used in any arithmetic expression.

Note that the formal parameters (A,B,C,D1) used in the previous DEF example must be simple, non-subscripted variables, although they may be subscripted in the expression.

For greater flexibility, UBASIC allows the programmer to define more complex functions by the use of the multi-line DEF. A multi-line DEF may include any UBASIC command with the following restrictions:

(1) Transfer of control statements may not refer to statements outside the range of the DEF; that is, statements after the FNEND, before the DEF or the DEF itself.

(2) DIM statements will be assumed to refer to main program variables, and not formal parameters of the DEF. Formal parameters are "dimensioned" by their appearance in a context which implies dimensioning. For instance, the occurrence of a variable with subscripts or in a MAT statement.

(3) All UBASIC functions are assumed to be "normal." This means that the function does not change the value of any of its arguments, and the function will always return the same value when given the same arguments. Functions which are not normal should be used with care, since if they occur in expressions, the point at which the function is called may not be what the programmer expects.

Example:

If FNA(N) changes the value of N, the statement

```
10 LET A=N+FNA(N)
```

will cause FNA to be evaluated, changing the value of N, then the new value of N will be added to the value of FNA. However,

```
10 LET A=FNB(N)+FNA(N)
```

will cause FNB to be evaluated first, using the old value of N, then FNA will be evaluated and added to FNB.

A multi-line DEF may return a numerical value, or it may perform a series of computations based on its arguments. If a DEF returns a numerical value, it is used exactly as the single-line DEF statement. Note that the mode of the parameters passed to a function must match the mode of the formal parameters. It is important that the user ensure that the string variables are given string arguments when the function is used in an arithmetic expression. The function name (FN followed by a letter) in a multi-line DEF is used exactly as a simple variable. The value that this function name is given in the DEF becomes the value of the function in the arithmetic expression which uses that function. (A "function value variable" is assigned the value zero on entry to the DEF.)

The DEF statement at the beginning of a multi-line DEF has the function name and formal parameter list as follows:

```
20 DEF FNC(A,B1,Q,V1$)
```

Note that this DEF does not contain an '=' or an expression.

Example:

```
10 DEF FNN(A,B,C$)
20 LET FNN=A
30 IF C$='YES' THEN 50
40 LET FNN=B
50 FNEND
```

If the function FNN is used in an arithmetic expression as FNN(1,10,'YES') the function would have the value 1. If the third argument had not been 'YES' (or a string variable with the value 'YES') the function would have been assigned the value of the second argument, 10.

The multi-line DEF may also be used to perform calculations or manipulations, where a numerical result is not the desired result.

Example:

```
10 DEF FNZ (A,B,R$)
15 IF B=0 THEN 35
20 FOR S=1 TO A
25 PRINT R$, B*S
30 NEXT S
35 FNEND
```

If the second argument (B) is not zero, then the above function will print the third argument and a multiple of the second argument a number of times determined by the first argument. If the second argument is zero, the function will do nothing.

Since this function does not assign a value to FNZ, its value, if used in an arithmetic expression, will be zero. To use this function, the following CALL statement may be used:

```
70 CALL FNZ(10,N1*L,'MULTIPLE')
```

This causes ten lines of 'MULTIPLE' followed by the first 10 multiples of N1*L to be printed, if N1*L was not equal to zero.

The DEF statement defining a function must appear in the program before that function is used.

UBASIC includes a feature known as recursion. This means that a function may call itself. The recursive feature provides a more powerful facility for use in computations.

For instance, the factorial function may be defined in UBASIC as follows:

```
10 DEF FNF(A)
15 LET FNF=1
20 IF A<=1 THEN 30
25 LET FNF=A*FNF(A-1)
30 FNEND
```

This function is based upon the fact that $0! = 1$, $1! = 1$, $2! = 2 \cdot 1! = 2 \cdot 1 = 2$, and generally $N! = N \cdot (N-1)!$.

The function will continue calling itself, until the argument is less than or equal to one, and then will return function values until the final value is reached.

3.11. CALL

The CALL statement is used to call a multi-line DEF which performs calculations and manipulations rather than returning a numeric value, as would be done by a single-line DEF function. The CALL statement includes the name of the function being called and the list of arguments to be passed to that function.

Example:

```
100 CALL FNA(N+B*C,10.5,A1)
```

might be used to call the following function:

```
10 DEF FNA(A,B,C)
20 IF A <> B THEN 40
30 LET C=0
40 FNEND
```

This would be equivalent to writing

```
100 IF N+B*C <> 10.5 THEN 102
101 LET A1=0
102 ....
```

In this example, the second form is more efficient, but in many cases a function is preferable.

The same general form may be used in calling system defined subroutines.

NOTE: If the CALLed function returns a value, it is ignored.

An additional form of the CALL statement may be used to call a second program for execution as a subroutine.

```
100 CALL Program, m
```

In this case, the program named would be compiled with the current program, and when the CALL statement is encountered, a GOSUB is executed to the statement designated by m. If no m is specified, then the first statement of the named program is used.

3.12. GOSUB AND RETURN

The GOSUB instruction is used to call a section of a program designed as a subroutine instead of a function. The GOSUB statement transfers control to the line referenced in the statement, and records the location being transferred from, so that the subroutine can return control to the next line. This return of control is accomplished by the use of the RETURN statement in the subroutine. When this statement is encountered, control is transferred to the line after the GOSUB from which the routine was called. GOSUB's may be nested in almost any way, as long as one RETURN is encountered for each GOSUB executed.

NOTE: GOSUB's, like functions, may be used recursively.

The following is an example of the recursive use of GOSUB and RETURN:

```
 5 A=2
 7 B=0
10 GOSUB 40
20 PRINT B
30 STOP
40 B=A+B
50 IF B> 6 THEN 70
60 GOSUB 40
70 RETURN
80 END
```

The first GOSUB transfers to statement 40. Since B only equals 2, the GOSUB in 60 will keep calling the routine and adding A to B until B=8. Then, control will reach the RETURN in 70, which will return to the last GOSUB used (line 60). The second time RETURN is encountered, it will return control to the second to the last GOSUB encountered, and so on until control transfers back to the statement after the GOSUB (line 20).

3.13. INPUT

The general form of the INPUT statement is:

```
n INPUT v1,...,v$1,...,vN,v$N
```

The INPUT statement is similar in form and function to the READ statement. However, instead of using DATA lists, INPUT requests data at RUN time, printing out a question mark and expecting the number and mode of information corresponding to the variable list in the statement.

Example:

```
10 INPUT A, A(1), A$, S$(N)
```

would print a question mark, and the user would supply two arithmetic constants and two strings, separated by commas, followed by a line termination. The first constant would be the value of A, the second of A(1), etc. If a data item is followed by the comma and ampersand (&), then the next data item will be taken from the following line. Data supplied in answer to the above INPUT statement could be on two lines.

Example:

```
3.14,14 'STRINGA',&
'STRING FOR S(N)'
```

This statement may also be used to input information from symbolic files. The file handling capabilities are described in 3.24.

3.14. CHANGE

The CHANGE statement is used to translate strings into arithmetic arrays or vice-versa. The first form of the statement is:

```
n CHANGE v$ TO a
```

In this form, the CHANGE statement takes the number of characters in v$ and puts this number into a(0). Then each character in v$ is translated into its arithmetic equivalent according to the following table, and stored into consecutive positions of the vector a.

The second form of the statement is:

```
n CHANGE a TO v$
```

The number of elements specified in a(0) is translated into characters and stored in v$.

The following table shows the characters and their numeric equivalents:

@	0				
[	1	(capital K from teletypewriter)			
]	2	(capital M from teletypewriter)			
#	3				
Δ	4	(on-site, ↑ is teletypewriter equivalent)			
	5	(blank)			
A	6	T	25	?	44
B	7	U	26	!	45
C	8	V	27	,	46
D	9	W	28	\	47
E	10	X	29	0	48
F	11	Y	30	1	49
G	12	Z	31	2	50
H	13	)	32	3	51
I	14	–	33	4	52
J	15	+	34	5	53
K	16	<	35	6	54
L	17	=	36	7	55
M	18	>	37	8	56
N	19	&	38	9	57
O	20	$	39	'	58
P	21	*	40	;	59
Q	22	(	41	/	60
R	23	%	42	.	61
S	24	:	43		

⟨ 62 (on-site only) ≠ 63 (on-site only)

3.15. REM

The REM statement in UBASIC provides a method for inserting comments or remarks into a program. Since the REM statement is non-executable (specifies no action to be performed by the computer), it may be placed anywhere before the END statement in a program. The form of the statement is:

n REM ANY COMMENTS

or

n REMARK COMMENTS

or

n REMINDER--COMMENTS

Any character may follow REM.

3.16. RANDOMIZE

The function RND generates pseudo-random numbers. Although these numbers have almost all the properties of true random numbers, any RUN will produce the same sequence of numbers. In order to start this sequence (which is very long) at a "random" point, the statement

n RANDOMIZE

should be used. This command initializes the starting point of the RND function using the time of day in milliseconds, thus providing a relatively random starting point.

3.17. END

The END statement is the last in every UBASIC program. It signifies the end of the program. If the END statement is executed, it terminates a run and prints out the elapsed running time. A program should have only one END and it should have the largest statement number in the program.

3.18. DIM

The DIM statement is used to reserve space for arrays and to record their maximum dimensions. Using the DIM statement, arrays can be dimensioned singly if string, and up to four subscripts if algebraic. The dimensions specified become the largest subscripts which can be used with the array variable. The statement consists of the keyword DIM and a list of arrays with unsigned integer subscripts. The general form of the DIM statement is:

n DIM a(i,j),a$(i),a(i),...

a(i) reserves space for the algebraic elements a(0), a(1),...,a(i). a(i,j) reserves space for the algebraic elements a(0,0),a(1,1),a(1,2),...,a(1,j),...,a(i,1),a(i,2),...,a(i,j). The string dimension a$(i) reserves space for i+1 sixty character strings referred to by a$(0),a$(1) ,...,a$(i). Note that storage of matrices is by row.

NOTE: (1) A zero dimension is legal, but impractical for matrix manipulation, since row and column zero are ignored. (See MAT, Section 3.23.)

(2) Function parameters used as arguments cannot be referenced in a DIM statement within the function. The DIM statement is for use only with main program variables (variables other than function parameters).

(3) If not referred to in DIM statements, vectors are assumed to be dimensioned 10, and arrays are assumed to be dimensioned 10X10, 10X10X1, or 10X10X1X1 depending on the number of subscripts. It is often advisable to use a DIM statement to exactly fix their length.

Dynamic dimensioning can be done using several matrix functions, which redimension an array as long as its size does not exceed the specifications of the DIM statement for that array.

The following are valid DIM statements:

```
10 DIM A(20),B(20,20),C(3,10),A$(5)
20 DIM B$(30),D(0,20),E(5,0),F(5,1)
```

3.19. EXCHANGE

The EXCHANGE statement, which has two forms, reverses the values of two variables. The forms of the EXCHANGE statement are as follows:

```
n EXCHANGE v1,v2
```

or

```
n v1==v2
```

3.20. CHAIN

The execution of the current program is terminated and the start of execution of the specified saved program is initiated by this statement. The form of the CHAIN statement is as follows:

```
n CHAIN Program-name,n
```

In the above form, program name specifies the name of a saved program and n specifies the statement number of the named program where execution is to begin. If n is not specified, then the first executable statement of the named program is used.

There are two modifiers which may be attached to the CHAIN statement, the colon (:) and the asterisk (*). When the colon (:) is used all values of single arithmetic variables in the current program are saved, and may be used by the chained program. The asterisk (*) suppresses the printing of the execution time for individual chained programs, but does not affect the printing of the total execution time. The general forms of these modified statements are:

```
n CHAIN* Program name , n
n CHAIN: Program name , n
n CHAIN*: Program name , n
```

Note that in the last form the order of the asterisk and the colon are unimportant. Also, there is no limitation on the number of programs which may be chained. Thus, a program which is chained may contain a CHAIN statement.

3.21. STRINGS

This statement specifies the maximum number of characters which are allowed in any string variables. The general form is as follows:

```
n STRINGS j
```

where j is an integer constant with a maximum range of 511. If no STRINGS statement appears in the current program, then the maximum number of characters per string is 60.

3.22. ARRAYS

This statement is used to specify the default dimensions of any arrays which are referenced in the current program, but not previously dimensioned in a DIM statement.

The general form of this statement is:

```
n ARRAYS D(i,j) S$(i), S(i), Q(i,j,k), Q(i,j,k,l)
```

D(i,j) specifies that all algebraic matrices, not included in a DIM statement be given the maximum dimensions (i,j).

S$(i) specifies that all string arrays, not included in a DIM statement, be given the maximum dimensions (i).

S(i) specifies that all algebraic arrays, not included in a DIM statement, be given the maximum dimension (i).

Q(i,j,k) specifies that all three dimensioned matrices, not included in a DIM statement be given the maximum dimensions (i,j,k).

Q(i,j,k,l) specifies that all four dimensioned matrices, not included in a DIM statement be given the maximum dimensions (i,j,k,l).

A default value of 10 will be given for i and j and 1 for k and l in the event that an ARRAYS statement is not included. There is no definite order in which D(i,j),S$(i),S(i),Q(i,j,k) and Q(i,j,k,l) must appear in the ARRAYS statement, and any of these may be omitted.

3.23. MAT

UBASIC incorporates matrix manipulation as an integral part of the language. The MAT instruction signifies that matrices are to be operated on, and the remainder of the statement specifies exactly what is to be done. The various types of MAT statements are explained following. They are applicable only to vectors and two-dimensioned arrays.

3.23.1. MAT ARRAY=MATRIX EXPRESSION

This MAT statement corresponds to the LET statement. It performs the operation indicated on the right of the equals sign and puts the elements into the array on the left. The matrix expression has several forms:

(1) It may be another matrix as

```
10 MAT A=B
```

which puts an exact copy of the matrix B into A.

(2) It may be a scalar multiple of a matrix (the scalar expression must be in parentheses) as

```
20 MAT A=(Z+2)*B
```

which makes A a scalar multiple of B.

(3) It may be the sum or difference of two similarly dimensioned matrices as

```
30 MAT A=B+A
```

which increases every element in A by the corresponding element in B.

(4) It may be the product of two compatibly dimensioned matrices as

```
40 MAT A=B*C
```

NOTE: The matrix on the left of the equals sign cannot be one of the matricies in the product on the right.

(5) It may be one of the matrix functions:

(a) INV(a) finds the inverse and the determinant of the square matrix a. The system function DET is used to obtain the determinant of the last matrix inverted.

(b) TRN(a) finds the transpose of the matrix a.

(c) ZER zeros the matrix on the left of the equals sign. ZER(ae1,ae2) or ZER(ae1) dimensions and zeros the array on the left of the equals sign according to the values of the algebraic expressions ae1 and ae2.

(d) CON fills the matrix with ones. CON(E1,E2) or CON(E) dimensions and fills the matrix with ones.

(e) IDN makes the square matrix the identity matrix. IDN(ae1,ae2) dimensions the array and makes it the identity matrix(ae1=ae2).

(f) DIM(ae1,ae2) or DIM(ae) redimensions an array without changing the elements.

NOTE: For matrices other than vectors, elements are not destroyed, but are almost impossible to reference due to the relative shift of most rows and columns. Vectors, however, do not have this disadvantage and are left still usable.

The following are valid MAT statements:

```
10 MAT A=C
20 MAT A=INV(C)
30 MAT A=TRN(A)
40 MAT A=ZER(A9,5)
50 MAT C=IDN
60 MAT A$=C$
```

3.23.2. MAT PRINT

The MAT PRINT instruction is used to print out a matrix in its array form. The instruction is followed by a list of array names, separated by commas or semicolons. A comma after an array name spaces the elements into zones, while a semicolon close-spaces the matrix. The following are valid MAT PRINT statements:

```
10 MAT PRINT A,B;C
20 MAT PRINT C
30 MAT PRINT A$;
40 MAT PRINT A,B$
```

3.23.3. MAT INPUT

The MAT INPUT statement is similar to the regular INPUT statement, in that it prints out a question mark and reads in the response. The MAT INPUT statement, however, reads in a whole vector and records the number of items of input for reference by the function NUM. The instruction itself has only one form:

```
n MAT INPUT a
```

3.23.4. MAT READ

The MAT READ statement, like the READ statement, takes data from the lists of DATA items. More than one array may be listed in the statement and the list will be processed from left to right. The arrays are filled, by rows, according to their dimensions or according to dimensions specified in the statement (any appropriate expressions).

The following are valid MAT READ statements:

```
10 MAT READ A
20 MAT READ A, A$, C(5*X,10)
30 MAT READ D$(10)
```

3.24. FILE I/O

3.24.1. OPEN

This statement which is used to open data files has the following forms:

```
n OPEN filename FOR { SYMBOLIC } { RANDOM } { INPUT  }
                    { BINARY   }   { or   }   { OUTPUT } AS FILE f
                    {   or     }   { NULL }   { PRINT  }
                    { NULL     }              { PUNCH  }
                                              {   or   }
                                              {  IO    }
```

or

n OPEN filename , { SYMBOLIC / BINARY / or / NULL } { RANDOM / or / NULL } { INPUT / OUTPUT / PRINT / PUNCH / or / IO } , f

where filename is an EXEC 8 filename — up to twelve letters, digits, dashes, slashes, or dollar signs and file number f is any positive numeric expression within the range one to nine.

NOTE: If the letters FOR are used in the filename in that particular sequence, the second of the above forms must be employed. Also, filename in the OPEN statement is optional. (See 3.24.9.)

Data files must be opened before they can be read or written. When the OPEN statement is executed the UBASIC file mechanism checks to see if a file with the same name has previously been catalogued, in which case the OPEN statement will apply to the catalogued file. If the file had not been previously catalogued, the system will assign a catalogued file with the designated filename. When the filename is omitted in the OPEN statement, the UBASIC file mechanism will assign an internally named temporary file.

Although there is no difference in file structure, BINARY or SYMBOLIC may be specified or both words may be omitted. If RANDOM is not specified, sequential structure is assumed.

IO means the file may be read or written without reopening, and thus has meaning only for RANDOM. Conversely PRINT or PUNCH files cannot be random.

3.24.2. INPUT

The INPUT FROM f: statement is used to read information from a specific file. The form of this statement is:

n INPUT FROM f : variable list

The INPUT FROM f: statement works identically to the general INPUT statement with the exception that line images are read from the file f.

NOTE: If the value of the file designator f is 0, input will be from the teletypewriter or an on-site card reader.

3.24.3. OUTPUT

To write on a file either of the following equivalent forms may be used:

WRITE ON f: expression list
PRINT ON f: expression list

where f, the file designator is any valid arithmetic expression and the expression list appears identically to the general PRINT statement (see 3.4.).

The PRINT ON/WRITE ON statements work the same as the PRINT statement except for printing on a specified file, and writing a comma between specific items on the print file. File f will consist of records which correspond to line images produced by the PRINT statement. A null record will be written to the specified file, if the expression list is blank. Note that the TAB, COL functions and the end punctuations may be used the same as with the general PRINT statement. (See Section 3.4.)

NOTE: If the value of the file designation f is 0, output will be to the teletypewriter or the on-site printer, if the run is batch.

3.24.4. PRINT

The print option of the OPEN statement allows the user to direct output to an on-site high speed printer. The full 132 characters of the printer are usable. All of the normal print rules apply to the use of a print file. If the OPEN statement contains a file name, the print file will remain cataloged after being printed. If a file name is not used in the OPEN statement, the file will be decatalogued after printing. A print file may not be used for input.

3.24.5. PUNCH

The characteristics of the punch file are the same as the print file (see 3.24.4.) with the exception that the record length is 80 characters and the output is directed to an on-site punch.

3.24.6. ON ENDFILE

The ON ENDFILE statement has the form:

n ON ENDFILE f GO TO statement number.

The ON ENDFILE is used to specify the next statement which is to be executed after an end of file is encountered on an input file. The file f may have multiple ON ENDFILE statements. For a file f the last ON ENDFILE statement encountered during execution determines the control path.

3.24.7. LINES

The LINES statement has the following format:

n LINES X,Y

Where X = File No.
Y = Record length in characters.

The LINES statement allows the user to set the record length of a file. The maximum record length is 512 characters. File 0 may be set to 72 or 132 characters. The default values for record length are 72 characters for file 0 and 132 characters for files 1 through 9.

3.24.8. CLOSE

The CLOSE statement is used to write an end of file on the specified file(s), when it is in write mode. The file f will be closed if it is in read mode when a CLOSE statement is encountered for the file specified.

The form of the CLOSE statement is:

n CLOSE F1,F2,...FN

3.24.9. TEMPORARY FILES

If the file name has been omitted on an OPEN statement, the file will exist only for the duration of the execution of the UBASIC program. This extension allows the user the use of scratch files during the operation of a program.

The file number associated with a temporary file should not be used for any cataloged file within the same program. The first operation on a temporary file must be an output operation.

The above information on temporary files does not apply to files opened for print or punch.

3.24.10. RANDOM FILES

Files may be established for random input and/or output operations by the OPEN statement (see 3.24.1.). Such a file must be OPENed in a run prior to any input or output operation, even though it was established and written in a prior run. Data must, of course, be written into the file before meaningful data can be read from the file. A CLOSE statement should be used after operations on the file are completed.

Reading and writing a random file is accomplished by CALLing the appropriate function below. F represents an algebraic expression which will designate a file number 1 through 9 corresponding to the OPENed RANDOM file (see 3.24.1.). a and n are algebraic expressions designating character counts.

RRD(f,a,n,V)

This function does a random read of information from a FASTRAND or simulated FASTRAND file.

For random reads, n FIELDATA characters (6 bits each) are read from file f, starting at character a, and stored in vector V starting at V(1). V must be singly dimensioned. Characters in file f are addressed starting at one.

Example:

RRD(1,101,100,A) causes the second hundred characters on file 1 to be read into A(1),A(2),...through the first four characters of A(17).

The RRD function returns a function value which describes the result of the read request. This value is usually called the status code, and a full description may be found in the *UNIVAC 1100 Series Operating System Programmer Reference UP-4144* (current version). The most common status code is: 0, if the read request was completed normally.

RWR(f,a,n,V)

This function does a random write to a FASTRAND or simulated FASTRAND file.

For random writes, n FIELDATA characters (6 bits each) are written on file f, starting at character a. The n characters are taken from vector V, starting with V(1). V must be singly dimensioned.

Characters in file f are addressed starting at one.

Example:

RWR(1,101,100,V) causes the 100 characters stored in V(1),V(2),...,through the first four characters of V(17) to be written on file 1 as the 101st through the 200th characters on the file.

The RWR function returns a function value which describes the result of the write request. This value is usually called the status code, and a full description may be found in the *UNIVAC 1100 Series Operating System Programmer Reference UP-4144* (current version). The most common status code is: 0, if the write request was completed normally.

3.24.11. FORMATTED INPUT/OUTPUT

The user may specify his own format for input or output by using FORM or IMAGE statements, which will describe in picture form (a mask of alphanumeric characters) how the user expects to input his data or how he expects the output to look (for example, a report with imbedded text). The general form of this statement is

$$n\ \text{PRINT}\ \left\{\begin{matrix}\text{ON e}\\ \text{or}\\ \text{NULL}\end{matrix}\right\}\ \text{IN}\ \left\{\begin{matrix}\text{FORM}\\ \\ \text{IMAGE}\end{matrix}\right\}\ \text{S:V1}$$

where e is an expression specifying a file number (see 3.24.1.). S is a string constant or a string variable and V1 is a list of variables. If 'ON e' is omitted, file zero or the demand terminal is assumed. If the desired function is input rather than output, the word INPUT replaces PRINT and FROM replaces ON if applicable. The word WRITE may be used interchangeable with PRINT.

The string constant or string variable specifies the format of the input or output. Thus, characters within the specified string have special meanings which will be explained following.

3.24.12. PRINT IN IMAGE, NUMERIC

Four types of IMAGE fields are recognized for PRINT IN IMAGE, numeric;

(1) the % (percent sign) field,
(2) the # (number sign) field,
(3) the $ (floating dollar sign) field, and
(4) the * (floating asterisk sign) field.

All other characters (except the decimal point in conjunction with the fields) are treated as text and printed. The latter two ($ and *) must consist of at least four characters to be considered as fields, otherwise they are treated as text.

The decimal point may be included with all the fields with the exception that it may start (be the first character of) only the % field.

The % field is an integer-decimal field and results in integer-decimal output corresponding exactly to the dimensions of the field. The field must be large enough to accept the number and if the number to be output is negative, must account for the sign to be printed. Thus, the field %% can accept for output a positive number of two digits (left of the decimal) or a negative number of one digit. All numbers to be output will be rounded to the requested number of places, right of the decimal. Places provided but not needed will result in blanks (for places left of the decimal) and 0's (right of the decimal). The following are all legal fields for numeric output: %%, %.%, %.%%, %%%%.%%%.The number 1.45 would output, using the four fields, as Δ 1, 1.5, 1.45, ΔΔΔ 1.450, (Δ represents a blank output). Notice in the first case the number is rounded to 1 and the character position not needed resulted in a blank output, in the second case the number is rounded to one decimal place; in the third case, the number is printed as is; in the last case, the positions not needed resulted in leading blanks and trailing zeroes. Notice also, that the number −1.45 could not be accepted for the second and third cases, as not enough positions left of the decimal point are provided to accommodate both the negative sign (which must be printed) and the number 1.

The # (number sign) field is an exponential format output of either of two types:

(1) The first is without a decimal point and must consist of *at least 7* consecutive #'s. The output generated will be in exponential format, with the mantissa portion beginning with a decimal point and having a number of places to the right of the decimal, equal to the number of #'s in the image *minus 6.* The six is accounting for the six character positions needed for exponential output, namely, one for the decimal point, one for the sign (which is a blank for positive numbers), one for the E of the output, one for the sign of the exponent, and two for the magnitude of the exponent. Seven #'s are required to provide at least one digit to the right of the decimal point, more than seven results in more than one digit right of the decimal point on output. Again, the number will be rounded to the required number of placed provided.

Example:

The preceding type of exponential, the image '########' , eight #'s would result in the number 120 output as Δ.12E+03; nine #'s would result in Δ.120E+03 and seven #'s would result in Δ.1E+03.

(2) The second type of exponential output is utilized when it is desired that the mantisssa have one or more (maximum 6) digits to the left of the decimal point and consists of the following, one to six #'s followed by a decimal point followed by *at least 5.* The output will be in exponential form with the number of places left of the decimal equivalent to that in the image field, and the number of places right of the decimal equivalent to the number right of the decimal in the field minus 5. The five are accounting for the sign of the mantissa (blank output if number is positive), the E, the sign of the exponent, and the two digits of the exponent. (The decimal point of the image accounts for the decimal point of the output.)

Example:

The field #.###### would output 120 as Δ1.2E+02 and −126 as −1.3E+02.

The floating $ and floating * fields are similar to the % field except that for the floating $ will print a $ instead of a blank to the left of the first number printed, and the floating * will print *'s in place of all blanks to the left of the first significant digit of the output.

Example:

$$$$ will cause 12 to be output as Δ$12 and **** will cause 12 to be output as **12. The fields must contain at least one space for the character $ (or *) in addition to an extra position if the number output is negative, since at least one of the field characters must be output. Thus, $$$$.$$ can accept 123 for output but not −123 since the one $ that must be printed and the sign which must be printed result in 5 positions needed left of the decimal. The dollar sign and asterisks are never printed right of the decimal point (if one exists). These characters right of the decimal are the same as % signs in handling.

Termination of fields for image output is the detection of a character which is not part of the current field. This would include a decimal point if one has already been detected in the field.

Example:

The image '%%$$$' contains two fields, the first (the % signs) is terminated by the first $ detected. The image '%%.%.%' also contains two fields, the first is %%.% and the second is .%, the second . acts both as terminator of the first field and first character of the second field.

All characters except the %, #, ., $ and * are considered as text as are the $ and * fields where there are less than 4 in the field, and will be printed as text.

Example:

The image '%%Δ ISΔ A,Δ%% ΔISΔ B' will print the values 12 and 23 as 12Δ ISΔ A,Δ23 ΔISΔ B (where Δ signifies a blank).

The decimal point will be considered as text unless it follows one of the field characters or immediately precedes a %.

Another field is possible for image numeric output, the single # or "free form" field. This is discussed in 3.24.14.

3.24.13. PRINT IN IMAGE, STRINGS

Only two types of fields are recognized as string fields for PRINT IN IMAGE (excluding the single #, "free form" field discussed later); they are the % and # fields. These are *without a decimal point* only. The string to be output will be truncated from the right if the field is not large enough and space filled on the right if the field is too large. Neither of these conditions are considered error conditions.

Example:

The field %%% will output the string ABCD as ABC and the string AB as ABΔ (Δ signifies blank).

Any attempt to output a string in a non-string field is an error condition and terminates the program. For the purpose of output of strings, there is no minimum number required of the # field with the knowledge that one # is the "free form" field rather than a one character output. Decimal points are considered as field terminators when string output is in operation.

3.24.14. FREE FORM OUTPUT

A single # (number sign) not followed by a decimal point is considered a free form field and results in both numerics and strings output in standard form, as they would under a simple PRINT V (where V is a numeric or string variable). A space will be printed leading all positive numbers. Strings are printed exactly as they exist.

3.24.15. PRINT IN FORM

An image is to be considered an image of a line. As such, when an image is exhausted of fields, a line of output is generated. The image is then searched from the beginning if more fields are needed.

A form is considered to be a set of fields and as such does not generate a carriage return upon exhausting the form, but merely starts searching from the start of the form for more fields if needed. A carriage return may be generated in form by use of the characters / and \. The former is a field terminator and the latter is not. (See discussion of field termination, form, below). Some examples to illustrate the differences are:

Example:

(1) Print in form '%%%': A,B would put the variables A and B on the same line (with no intervening spaces if the variables filled the field).

(2) Print in image '%%%':A,B would print A on one line, B on the next line since the exhaustion of the image results in line output.

(3) Print in form '%%%/' or '%%% \' would result in the same output as print in image '%%%' since the characters / and \ cause line output.

3.24.16. PRINT IN FORM, NUMERIC

The discussion of PRINT IN IMAGE, numeric applies to PRINT IN FORM with some exceptions:

(1) A form field is terminated only by a terminator character (discussed below) and not by a simple change of type of field.

(2) For the floating $ and * fields, two consecutive characters constitute the floating field, and not a minimum of four as in image.

Additionally, the floating $ and * fields will not print the sign of a negative number. All characters of the form are considered *not* to be text, unless they are enclosed by *two* quote marks on each side of the character(s). This will mark them as text characters to be printed. Fields are terminated only by the appearance of a space, / (carriage return) or the end of the form. To generate a space on output, the utility character B is used. To generate a carriage return without terminating a field, the utility character \ (line field) is used.

PRINT IN FORM has available a set of field characters much more expressive and versatile than the PRINT IN IMAGE statements. At the same time, this increased versatility requires a more thorough understanding of the many rules that must follow and much more care is required to avoid error.

As mentioned, those fields that function in PRINT IN IMAGE are present for PRINT IN FORM. It is most important to understand the major differences between PRINT IN FORM and PRINT IN IMAGE. For PRINT IN FORM:

(1) Fields are terminated *only* by 'field terminators.'

(2) The end of the form does not generate a line of output (carriage return).

(3) Text must be specifically delineated as such by enclosing it in *two* single quote marks on *each* side of the desired text.

First, we should grasp the significance of 'field terminators.' These are, specifically, the blank, the carriage return symbol (/), and the end of the form itself.

Example:

The form A$ = '%%Δ%%' has two distinct fields, each two digits, or, two characters.

Using this form to output the variables A and B where A is 12 and B is 34 would result in the line of output: 1234. Notice that the numbers are not separated by a space since the space in the form serves merely as a field terminator; it does not generate an output space. It must also be understood that the line of output was generated by the end of the variable list and not the end of the form. In fact, the form '%%' would have resulted in an identical output of A and B (here the end of the form serves as the field terminator). In exhaustion of the form, the field termination occurs, and if more fields are needed, the form is searched from beginning for needed fields. This may occur as many times as necessary to output the required number of variables.

After looking at two of the field terminators, we will turn to the third, the carriage return (/) symbol. This symbol will cause a line of output to be generated, at the same time serving as a terminator of a field.

Example:

Print in form '%%/%%': A, B would output the above used values (12 and 34) on two lines since the / symbol terminates the first field (two % signs) and generates line output.

It is possible to have two or more consecutive field terminators (except for the end of the form); however, it serves a useful purpose only for the last mentioned (/). More than one space is superfluous since one is sufficient to terminate a field and does not itself create any output; however, spaces are accepted and the user may use them to make his form more readable to himself or others. Two or more consecutive /'s, however, do serve a useful purpose in that they will generate the desired number of blank lines of output.

Example:

PRINT IN FORM '%%//%%': A, B again using 12 and 34 for A and B, would result in the numbers 12 and 34 being printed with one blank line between. The first field (two % signs) would receive the variable A, the first / would terminate the field and generate the line of output, the second / encountered would generate a blank line of output (the variable A has already been output). The second field would receive the variable B and the line of output generated by the end of the variable list, B being the final variable output.

Text output is possible in form, only when the text is delineated by two quote marks blanking the text. The exception to this is the special character B which denotes an *output blank* and may appear anywhere in the form. Text appearance does not terminate fields, neither do the B's denoting blanks to be printed, thus text and B's may be embedded in fields. The exception is the floating fields (discussed later) in which text and B's may not be embedded. Examples of this will be shown later, after discussion of special form characters.

PRINT IN FORM has, besides the fields present in image, certain characters designated as "special form" or "precise form" characters. For the following, we shall be concerned with numeric output only . String output will be discussed in 3.24.17.

For numeric output there are four special form characters to be noted. They are D, Y, Z and Q. The significance of each is as follows. D is a digit indicator and will result in a digit output. Those field characters not needed for output of the number will be zero filled.

Example:

The field 'DDD' would output the number 12 as 012 and the number 1 as 001, i.e., each character of the field outputs a digit. The character Y is also a digit indicator except that all positions not needed result in a blank output and all zeros in the number result in a corresponding blank output.

For example, the field 'YYY' would output the number 12 as 12 (preceded by a blank) and the number 103 as 1 3 (with a blank between the 1 and the 3). The character Z is similar to D except that leading zeros would be replaced by blanks; for example, the field 'ZZZ' would output 12 as 12 (preceded by a blank) and 103 as 103 (only leading zeros result in blanks). The character Q is a very special character; it performs essentially like the Z except that leading zeros are not printed, and do not print a corresponding blank. Instead, the number to be output in the Q field is left justified, and leading zeros are ignored. As such, the Q field is an 'indeterminate length' field because only as many of the Q's as are needed to output the variable are actually used. It should be mentioned now that, for this reason, the Q field is for output only and cannot be used for an input operation. An example of the Q field, the form 'QQQ' would output 12 as 12 with *no* leading space, and the number 1 as 1 (again with no leading spaces).

It should be mentioned that none of these four special form characters will output the sign of a number. To output the sign of a number requires the use of one of the three sign fields, S, + and −. These three characters will output the sign of the number when used singly (as in the form 'SDD') or when used in the 'floating sign' field (as in the form 'SSS' or '+++'). The single character is referred to as a 'static' sign field, the multiple characters are 'floating.' The floating fields act identically to the floating * and $ fields discussed earlier. The differences in the three fields are: the S outputs the sign of the number (− if negative, + if positive), the − outputs a − sign

if the number is negative and a blank if the number is positive. The + will print a + if the number is positive and a blank if the number is negative. For the floating fields, these characters will float to the immediate left of the first character printed.

Example:

The number 12 would be output using the three forms,'SSSS' , '++++' , and '----' , as +12 (preceded by one blank), +12 (preceded by two blanks) and 12 (preceded by three blanks).

These special form characters may be combined to form fields; for example, 'DYYDD' is an acceptable form. Certain rules are established, and they are as follows:

(1) The floating fields may not be broken by any other characters (except comma and decimal point discussed later).

(2) The floating fields may appear before, but not after D's, Y's, in a field; additionally they cannot be part of the Z or Q fields.

(3) The appearance of any of the floating field characters to the right of a decimal point (if present) results in the same handling as if they were D's.

Example:

The form '+++.++' is identical to '+++.DD ' .

(4) Q's and Z's may appear before but not after D's or Y's.

(5) No more than one floating field character may appear in one field (exceptions noted under exponential output discussion).

(6) No field characters (other than the 'static' sign characters) may appear before the Q fields.

(7) Of the characters Q, Z, and the 'floating' signs, only one type may appear in a field.

The decimal point is available in a similar manner as in the normal fields, for example, the form 'DD.DD' calls for a numeric output with two digits left of the decimal and two to the right. The handling of the number is similar to the normal, but follows the individual characters of the special forms. The previous form 'DD.DD' would output 12.3 as 12.30, 12.345 as 12.35 and .3 as 00.30. Notice that rounding is still in effect and the D's require an actual digit output (leading zeros are required). A special form character, V exists for decimal output. The V is the same as the . (decimal point) in a form except that is suppresses the print of the decimal point. "DDVDD' would output 12.34 as 1234, the decimal point implied between the 2 and 3.

Exponential format may be requested by using the special form characters E and K in conjunction with the other field characters. The E requests exponential output with the E printed. Use of K instead of E merely suppresses the print of the E.

Example:

'DD.DE+DD' would output 127 as 12.7E+01.

The exponential portion of the number, the part to the right of the E, may be considered a field by itself when observing the rules of precise form characters mentioned above.

For example, the form 'D.DESS' is valid, Rule (2) above is not violated, since the exponential portion of the form is considered (for the rule definitions only) as another field.

There are, however, two particular rules concerning exponential output to be considered.

(1) The sign of the exponent may only be printed between the E (or K) and the digits of the exponent.

(2) Floating sign fields for the exponent part of a number must be at least three characters in length. The significance of rule (1) is that the appearance of a sign character (S,+,–) after the digit characters of the exponent is assumed to apply to the mantissa.

Example:

The form 'SD.DESDDBBS' would output –45.6 as –4.6E+01–.

An example of the use of K is the form 'DD.KSDD' would would output 1217 as 12.+02. The decimal characters (. and V) may not be used following the K or E on exponential output. Also, it should be obvious that no more than one K or E and no more than one . or V may be present in the same field.

Two special characters remain to be discussed. The comma and the line feed (\ ) symbol. The comma is used in conjunction with the other special form characters to cause the printing of commas on output. It is conditional in the sense that it will not print the comma (instead printing a blank) until the digits have begun to print.

Example:

The form 'ZZ,ZZZ' would print 12345 as 12,345 but print 345 as 345 (preceded by 3 blanks).

This conditional status holds for all the other field (digit) characters except D's, which will cause the commas to print (since the D field outputs digits for every place). Another way to put this is that no comma present in a field will cause a comma to print before the first digit of output. When commas are part of the Q field, they too will be discarded completely from the left to adjust to the size of the number to be output.

Example:

'QQ,QQQ,QQQ' would output 1234 as 1,234 (with no preceding spaces).

The line feed character (\ ) acts identically to the carriage return symbol (/) with one notable exception. It is *not a field terminator,* and thus may be used to generate split-line output.

Example:

The form 'DDD. \ D' will print the number 123.4 as 123. on one line and 4 on the next line. Multiple \ symbols may be used to generate blank line output similarly to the usage of /.

For print in form, the rules of field capacities found in PRINT IN IMAGE apply; that is, if a number is too large for the field provided, an error message will result, and the field will be replaced with ****, instead of a numeric output. The floating fields require that at least one space be provided for output of the floating character (or blank).

For numeric output, the special form characters may not be concatenated with the normal form characters (in the same field).

Example:

'DD%.DD' is an invalid form field as is 'D.D#####'. However, a multi-field form may have fields of each type in it.

3.24.17. PRINT IN FORM, STRINGS

For PRINT IN FORM, strings, the % field of image output is still valid. However, the # field is accepted only in its single (or free form) connotation.

In addition to the % and single #, there are four special form characters available, the characters D, Y, A, X. These may be concatenated (with the % sign if desired). The differences of each are as follows: D will accept only a digit or blank for output, changing all blanks to zeros. Y will accept only digits or blanks for output, changing all zeros to blanks. A will accept only the 26 characters of the alphabet and blank for output. X (and %) will accept any character for output.

Text and the character B (to generate a blank on output) may be embedded in string fields without termination as may the line feed character (\ ).

Any violation of the previous rules on A, D and Y will result in an error message and the corresponding character replaced with an asterisk.

For example, an attempt to print the string 'ABC' using the form 'ADA' would result in an error message and an output of A*C.

In addition to the free form field denoted by the single #, there is another field, the single R for free form output. This character is valid only for strings. It is similar to the single # except that it generates a carriage return after output of the string. (Acting the same as the field '#/' would.) Both the single # and the single R output strings exactly as they exist, character for character.

Strings are left justified in all string fields. Character positions provided, but not needed are assumed to have blanks for output.

Example:

The form 'AAADD' would output the string 'ABCD' as 'ABCD0'. The line output character R is accepted as a single R; using two or more consecutive R's without field terminators between will still result in each R being treated as a separate field.

3.24.18. INPUT IN FORM AND IMAGE

To understand input using the picture formatting image and form statements, it is best to keep in mind the rules of output. A thorough understanding of output reduces the confusion of input and clarifies the rules to be followed. Generally all input should be assumed to have been previously output, using the same form or image as the input form or image. What a particular form or image produces on outputting a certain number (or string) determines what that same form or image expects on input.

There are some exceptions to this general rule, of course, but they are few and minor.

In the INPUT IN IMAGE statement the fields are identical to those on output. The differences in the handling of the fields are to the users advantage.

For example, the field $$$$ may be used on output for a number no larger than 999 (−99 if negative); however, the $ sign which must be output need not be input, thus the field will input a number up to 9999 (or −999) even though it could not output these numbers. This is true for the floating asterisk fields also; that is, no asterisks need be input. They are ignored if present. The only restriction is that any character other than digits, decimal points, signs, or field characters, are invalid and will result in error.

The main restriction on input lies with the multi-field forms and images. In these, it becomes very important that proper spacing be maintained, remembering always the corresponding output that was generated.

Example:

If the image '%% %%' is used to input A and B and the input line is 1234, the value of A would be 12 and B would be 4 and not 34.

The reason is that the space in the image caused the 3 input to be treated as text (as the corresponding character would be on output). Thus, text (and the B's in the forms) must have corresponding characters on input to keep the input "lined up." The text in the forms and images result in the corresponding characters input being "blanked."; that is, changed to blanks. The form "DDBB DD', used to input the values for A and B (for the

example assume A = 12 and B = 34) would require the input 12??34) where ? is any character). Notice that there are not three characters between the numbers, the space in the form would not generate a character on output; it does not account for one on input.

The restrictiveness of the special form characters is less on input than on output, but still must be taken into account. The special form characters D and Y, although different on output, are identical on input, each will accept blanks or digits only, changing all blanks to zeros. This is true for strings as well as numeric input. The + (static or floating) would output a blank for a negative sign.

Example:

The form field +DD would print the number −12 as Δ 12 (Δ is a blank) and would print 12 as +12, thus on input a number preceded only by blanks would be assumed to be negative. The field +DD would input Δ 12 as −12.

The movement of carriage returns and line feeds on input must be identical to that which would be done on output.

For example, the image '%%%' would output the numbers 123 and 456 on separate lines since the end of the image after printing the first variable (123) would result in a line output and carriage return. Thus, to input the numbers 123 and 456 using this image, the input must be on separate lines. You would type 123, hit return, and type 456 to effect the correct input. The input routine in this particular case, would completely ignore any characters typed after the 123 since the image indicates precisely three characters followed by a carriage return.

The fact that all fields of a form or image must be filled with some input to keep the input aligned with the correct field, is not applicable to the last variable input or the last field prior to a carriage return.

For example, the image '%%%' would cause the input stream of 12 followed by a carriage-return character as 12 since the carriage return results in acceptance of as much input as was given (to a maximum of 3 characters). The form 'DD BDD' could be used to input A as 12 and B as 3 by merely typing 12Δ3 (Δ is a blank). Since the 3 is the last variable being input, (B) the second character of the second field need not be accounted for. Trailing blanks on input are disregarded. However, notice that 1Δ 23 would result in A being assigned the value 1 and B the value 3. (The blank input was changed to zero by the D in the form and the 2 input was blanked by the B in the form.)

Violation of field restrictive characters will result in an error message and a return to request the input be sent again.

For example, an attempt to input 123 into a form 'DDA' as A$ (string input) would result in an error message and a retry on the input. The same would be true if −12 were to be input to the form field 'YYY'. Since Y could not output a sign, it will not input a sign.

An exception to this is the floating $ and * fields in form. They will not output a sign, but will accept one on input.

INPUT IN IMAGE with a single # (number sign) field acts as a combination output and free form input statement. If there is a single # in the image, any characters (even those normally treated as control characters) prior to the # are treated as imbedded text for *output*. These characters are sent to the terminal as output and the terminal "sits" at the next column position. The user then responds with free-form input (string or numeric) which must match the remainder of the image (the single # and any other control or text characters). Everything prior to the first single # is treated as output and the remainder of the image is treated normally.

Example:

```
100 INPUT IN IMAGE 'TYPE YOUR NAME #': A$
TYPE YOUR NAME
```

would be printed at the terminal. The user would respond *on the same line* with a string of characters which would become the value of A$.

3.24.19. FORM REPLICATION

In using Form (print or input), the form string itself may utilize "repeater" characters or "shortened form" characters. These characters allow the user to generate complicated forms with rather simple strings. There are three different methods.

(1) The "numeric repeater", example: '3D/4Y'. Use of a number in front of a character calls for repetition of the character N times (where N is the numeric repeater). The number only applies to one character, which may be any character. The above example then is equivalent to 'DDD/YYYY', the D repeated 3 times, the / only once since no numeric repeater appears in front of it, and the Y repeated 4 times.

(2) The "multiple character repeater", brackets; example '2[DX]3[DY]'. The numeric in front of the brackets call for repetition of what is inside the brackets. The above example is equivalent to 'DXDXDYDYDY'. The usage of brackets then is the same as the simple numeric repeater except that it will allow repetition of character sets rather than single characters only.

(3) The "multiple field repeater", parentheses: example '3(DY)2(X)'. The parentheses are equivalent to brackets in that the contents of the parentheses are repeated N times (N is the numeric multiplier). Where the brackets and parentheses differ is in the concept of "field" as opposed to "character(s)". The contents of the parentheses are considered to denote one (or more) field(s). Since a space serves as a field terminator only and has no other effect on input or output, a blank will be placed following each repetition of the parentheses contents.

It should be understood now that all forms are expanded prior to their inspection for input or output usage. The numeric repeaters, brackets, and parentheses will be removed and the form expanded to its equivalent containing none of these characters. On this "break out", spaces are generated at the completion of parentheses content repetition (each repetition).

Thus, the above example is equivalent to 'DY DY DY X X'. Notice that spaces are generated between the DY's and the X's, to separate the fields. The form '3(D)' is equivalent to 'D D D', again, three separate fields. Parentheses may be nested (up to 4 levels) remembering that each parentheses content is at least one field. Thus, the form '3(DY3(D))' is equivalent to the form 'DY D D D DY D D D '. It should be pointed out that *no nesting of brackets* (type 2 above) is permitted.

Brackets may be present inside parentheses, however;

Example:

'2(D2[XA] Y)'

is equivalent to 'DXAXAY DXAXAY'. No space is generated between the XA pairs or between the A and Y; brackets do not delineate fields, only character sets. The space between the Y and D is generated as the contents of the parentheses completes the first cycle of generation. The space then separates the two fields intended by the original form. Fields may be generated by use of brackets instead of parentheses if field terminators are inserted in the desired locations.

Example:

'2[XA] 2[D] ' is equivalent to '2(XA)2(D)', is equivalent to 'XA XA D D'. The blanks inside the brackets cause the contents of the brackets to generate fields.

In summary the following rules apply to repetition format:

(a) Numeric repeaters are of the form 2D3Y and merely call for the repetition of a single character a certain number of times. 2D3Y≡DDYYY.

(b) Character set repeaters (or multiple character repeaters) are of the form 2[XA] 2[DA] , similar to numeric repeaters except that a character set instead of a single character is multiply-generated.

Example:

2[XA] 2[DA] =XAXADADA.

(c) Field repeaters are of the form 2(XA)2(DA) and generate fields equal to the contents of the parentheses. Blanks are inserted as the form is expanded to its non-repeater form. The inserted blanks will separate the fields in the final form.

Example:

2(XA)2(DA)≡XA XA DA DA.

(d) Negative numbers used as repeaters are not accepted; the minus sign is treated as a part of the form and reproduced as a minus sign.

Example:

'–2A'='–AA'.

(e) Repetition format may be utilized *only on forms*. Use of these methods for *image* input or output is valid.

(f) Parentheses may be nested up to 4 levels.

(g) Brackets may *not* be nested within brackets, but pairs of brackets may be nested inside parentheses. Parentheses may be nested within brackets but will follow the rules for field generation (c above), rather than bracket rules.

3.25. DEBUGGING AIDS

Several statements are provided in UBASIC which will help the user find problems in his program logic.

3.25.1. VAR=ZERO

UBASIC will initialize all algebraic variables to zero and all string variables to blanks (current length of zero). If the statement VAR=ZERO does not appear in a program, all simple variables will be checked to assure that they are assigned a value somewhere in the program by a LET, EXCHANGE, INPUT or FOR statement. If a variable is referenced but not assigned a value, an appropriate diagnostic will appear as follows:

```
STRING VARIABLE [variable list] UNASSIGNED, or
ALGEBRAIC VARIABLE [variable list] UNASSIGNED
```

Example:

Thus,

```
100 A=B
200 END
```

would produce the diagnostic

```
ALGEBRAIC VARIABLE [B] UNASSIGNED
```

while

```
50  VAR=ZERO
100  A=B
200  END
```

would produce no diagnostic.

3.25.2. PAUSE

The user may halt execution of his program at any point to examine variables with the statement

```
n  PAUSE
```

at the specified point.

When the PAUSE takes place the following is printed:

```
PAUSE AT LINE NO: XXX
COMMAND? >
```

The user must then type in one of the following commands:

PRINT v1	– print the values of the listed variables
SET v=e	– set the variable v to a new value as calculated from the expression e
VAR=ZERO	– set all algebraic variables to zero and all string variables to blanks (set current length to zero)
DEBUG ON	– turn the debug mode on
DEBUG OFF	– turn the debug mode off
RESUME	– resume execution
STOP	– terminate the executing program
DUMP	– terminate the executing program with a post-mortem dump

If the user responds with one of the first five of the above commands, UBASIC will solicit additional commands until the user responds with RESUME, STOP, or DUMP.

At any time a UBASIC program is running in one of two modes, DEBUG ON or DEBUG OFF. In DEBUG OFF mode the PAUSE statement will have no effect (i.e., the information previously described will not be typed out). Thus, once a program is debugged, it can be run unhindered by the PAUSE statements, even though they still exist in the program.

DEBUG ON mode may be set by:

(1) calling UBASIC with a B option.
(2) by system command (DEBUG ON).
(3) in response to a command solicitation previously described.

DEBUG OFF mode may be set by:

(1) calling UBASIC without a B option.
(2) by system command (DEBUG OFF).
(3) in response to a command solicitation previously described.

3.25.3. BRK

The user can effect an orderly break in his program execution at any time (even if DEBUG is OFF – see 3.25.2.) by depressing the interrupt (BREAK) key. This will cause the message

```
INTRPT LAST LINE
```

to be typed out, at which time the user depresses the carriage return key. UBASIC will then halt execution and (after printing any buffered lines of output) solicit action by typing

```
YES?
```

The user may indicate (see 4.11.) that he wishes to interrupt execution of his program by responding BRK. This causes action similar to a PAUSE statement with the printout as follows:

```
BREAK AT LINE NO:XXX
COMMAND? >
```

The user may then respond with any of the commands in an identical manner as to PAUSE (see 3.25.2.).

3.25.4. TRACE

The logic flow of a UBASIC program may be determined by a trace of program execution. To accomplish this, two statements should bracket the block of statements to be traced as follows:

```
n1  TRACE  ON
.
.     statements to be traced
.
n2  TRACE  OFF
```

When any of the statements to be traced is first executed, UBASIC will solicit information from the user as to the trace information to be provided for subsequent statement traces. This is done with the messages as follows:

```
TRACE XXX
LINE NUMBER ONLY? YES OR NO >
```

Where XXX is the line number of the statement to be executed.

A YES answer will cause execution to resume. A NO answer will cause the message:

```
ALL VARIABLES? YES OR NO >
```

to be typed. A NO answer will be followed by the question:

```
WHICH VARIABLES? >
```

This should be answered by the names of the variables separated by commas. Having determined that these are legal names or if the ALL VARIABLES question were answered by YES, UBASIC will ask:

```
ONLY WHEN CHANGED? YES OR NO >
```

After a YES or NO answer UBASIC will continue execution.

Prior to the execution of each statement in the specified block, UBASIC will print:

```
TRACE XXX
```

Where XXX is the line number. Then all variables, specified variables or no variables will be printed according to the user's specifications.

During DEBUG OFF mode (see 3.25.2.), TRACE statements will have no effect. If, for any reason, the user wishes to change the trace data specifications (add a variable to the list or change from line numbers only to all variables) he may do so by turning DEBUG OFF (after an interrupt or PAUSE) and then turn DEBUG ON as follows:

```
YES?  BRK
BREAK AT LINE NO: XXX
COMMAND? >DEBUG OFF
COMMAND? >DEBUG ON
COMMAND? >RESUME
```

This would cause the next traced line to go through the 'first time through' trace solicitation procedure, at which time different answers may be given. The answer SAME to the question 'WHICH VARIABLES?> ' will result in the same list as was previously submitted in the current run.

As many blocks as are desired to be traced may be bracketed with TRACE ON, TRACE OFF pairs. If the final TRACE OFF is omitted, then tracing will be effective through the end of the program.

3.26. EXEC

UBASIC allows execution of a statement which will submit control cards to the Executive without leaving UBASIC. It has the form:

```
n EXEC e$
```

where e$ is a string expression containing the control card image to be submitted to the Executive. The following control card images may be submitted by the EXEC statement:

@ADD	@FREE	@START
@ASG	@LOG	@SYM
@BRKPT	@QUAL	@USE
@CAT		

Execution will be stopped if the string expression e$ is not properly formatted. If the Executive rejects the control card, a status word will be returned in CSF and may be checked by that function (see 2.7.1.). These facility status bits are fully explained in *UNIVAC 1100 Series EXEC 8 Hardware/Software Reference UP-7824* (current version).

Example:

```
30 IF FLD(0, 1, CSF)=1 THEN STOP
```

4. SYSTEM COMMANDS

The following commands are system commands and thus are not preceded by line numbers. These system commands are the control language for UBASIC and provide the capabilities for making this a powerful system.

The program file selection mechanism for UBASIC is:

(1) A file selection may be made any time a program name is supplied to the program. The following commands supply a program name to UBASIC:

(a) NEW
(b) OLD
(c) RENAME
(d) UNSAVE

(2) When both a named program file and TPF$ are attached to UBASIC, TPF$ is selected by appending an (*) to the name supplied to UBASIC by the preceding commands.

Example:

(a) OLD:NAMEA

This command brings the program NAMEA into main storage from the program file named on the processor call statement (see Appendix A).

(b) UNSAVE:NAMEB*

This command deletes the program NAMEB from TPF$.

(3) When a named program file is not attached to UBASIC, the file handler mechanism ignores the asterisk (*).

4.1. NEW

NEW: The NEW statement in the UBASIC system is used to begin a new program at any time. NEW erases all previous unsaved programs and asks for a new program name. This new program name may also be specified in the NEW command by following NEW with a colon and the new name as follows:

NEW:TEST

The program name may be up to twelve characters long. If a line termination is supplied in place of the new program name, the name NAME$ will be assumed. Similarly, if a UBASIC run does not begin with a NEW command, one will be assumed and the name NAME$ will again be assumed. The form of this command is as follows:

```
NEW:NAME
```

or

```
NEW:NAME*
```

4.2. SAVE

The SAVE command is used to save UBASIC programs on the Mass Storage devices. The current program at the time of the SAVE is stored.

4.3. OLD

The OLD command specifies that a previously saved program is to be used. The system, following an OLD instruction, erases the current program, requests the name of the saved program, and reads in this program for use. OLD, like NEW, may be followed by the name desired, as follows:

```
OLD:NIM
```

or

```
OLD:NIM*
```

4.4. SCRATCH

In UBASIC the SCRATCH System command is used to erase the current program, leaving only the name. This has the same effect as NEW except that the same program name is retained.

RENAME has the complementing effect. It erases only the program name and asks for a new one. RENAME may also specify the new name with a colon as follows:

```
RENAME:NAME2
```

This is especially useful before saving programs since saving a program named NAME$ is prohibited. The form of this command is:

RENAME:NAME

or

RENAME:NAME*

4.5. REPLACE

The REPLACE command is used to save a program under the same name as a previously saved program. REPLACE is exactly equivalent to an UNSAVE followed by a save command. The effect is to replace a saved program with the current program.

4.6. CATALOG

The CATALOG command will produce a list of active programs in the program file. An * will select TPF$ if a named program file is attached.

4.7. LIST, LISTNH AND LENGTH

The LIST command is used to list the current program or selected portions of it. Before listing, the program is sorted, and the blank or replaced lines are deleted.

The general form of the LIST command is:

LIST line 1 , line 2 , line 3 , line 4 , ...

If no line numbers are specified, the whole program is listed. Otherwise, only those lines numbered between line1, and line2, between line3, and line4,.... are listed. If an odd number of line numbers (including 1) is specified, listing continues from the last line number specified to the end of the program. If the command were LISTNH instead of LIST, the program would be listed without the usual heading.

Before listing, it is sometimes advisable to verify the length of the program. This is accomplished with the command LENGTH, which responds with the number of lines in a program. LENGTH is also useful in determining the size of a program for diagnostic purposes, since it is possible to write a program which exceeds the memory capacity supplied in UBASIC. (See Appendix B. for size restrictions.)

4.8. SEQUENCE

The SEQUENCE command is used to eliminate the necessity of typing line numbers before every statement in a UBASIC program. The command supplies a starting number and an increment.

Example:

```
SEQUENCE    100,10
```

will start numbering at 100 and increment by 10. The SEQUENCE produced line numbers are printed out at the beginning of each line, following which the user types his statement, followed by a line termination (carriage return on teletypewriters). Sequencing continues in this manner until an END statement is read.

4.9. RUN AND RUNNH

The system command RUN is used to start the execution of a UBASIC program, whether this program has been introduced by an OLD command, by typing in the program, or by reading it from some other medium. RUN specifies that the program is complete and that the execution of its contents should be attempted. The UBASIC compiler then translates the UBASIC program into instructions, which can be executed by the UNIVAC 1100 Series Processor. If no fatal errors have been detected, these instructions are executed. When execution has been completed, the elapsed computer time is printed, and the system is ready for the next statement.

The RUNNH command performs the same function except that it inhibits the printing of the time and date heading.

4.10. BYE AND GOODBYE

Either the BYE or the GOODBYE command signifies that the user is finished with UBASIC. Upon receipt of either command, UBASIC will erase the current program and sign off (but without disconnecting the teletypewriter). Control is then returned to the Executive.

4.11. STOP

In UBASIC, the STOP system command is used to stop program execution at any time. If input (see INPUT) is requested, the word STOP without quotes halts the run and prints the elapsed time. Thus, it can be used to stop a loop containing an INPUT instruction. To use the STOP command during pure execution (with no INPUT or PRINT statements involved) requires a more complex procedure.

On a demand terminal, the user must first press the interrupt key (*BREAK*), then the Executive will respond with: INTRPT LAST LINE. The user must then type in: *carriage return* and then STOP *carriage return*. The carriage return, returns control to UBASIC and makes STOP valid. If RUN were typed after carriage return, control would be returned to the program exactly where it left off. If debugging is desired, BRK may be typed (see 3.25.3.).

Note carefully that if a PRINT statement is contained within an execution loop, hundreds of lines of print will have been generated before *break* is typed, and these lines must be printed before STOP can be effective. Thus, if such a print-loop occurs, there is no choice but to type an X instead of carriage return and STOP. The *X will have the effect of a BYE*, except that no sign-off message occurs.

4.12. EDIT

In UBASIC, there are six editing commands.

4.12.1. EDIT DELETE

The EDIT DELETE command is used to delete (erase) selected portions of a program without having to type each line number individually. The EDIT DELETE command is followed by pairs of line numbers. All the lines numbered greater than or equal to the first of the pair and less than or equal to the second will be deleted. If there is an odd number of statement numbers listed, all the line numbers greater than or equal to the last number will be deleted.

Example:

```
EDIT DELETE 10,30,50,120,900
```

will delete lines 10 – 30, 50 – 120, 900 – (END) inclusive.

4.12.2. EDIT EXTRACT

The EDIT EXTRACT command is the complement of the EDIT DELETE command. With EDIT EXTRACT the sections of the program specified are retained and all other lines are deleted.

Thus after the command:

```
EDIT EXTRACT 10, 90, 200
```

the current program will consist only of the lines numbered between 10 and 90, inclusive, or 200 and greater.

4.12.3. EDIT RESEQUENCE

The EDIT RESEQUENCE command is used to renumber the statements in a program, beginning at any line number and continuing by a specified increment. The format of the EDIT RESEQUENCE command is as follows:

```
EDIT RESEQUENCE 100, 51, 10
```

This particular example will replace the old line number 51 with 100, the next with 110 and so on until the last statement. If the old line number is not specified or is zero, resequencing begins at the first line of the program. If a new line number is not specified, numbering begins at 100, and if an increment is not specified, 10 is used.

Thus the command is as follows:

```
EDIT  RESEQUENCE  is  equivalent  to
EDIT  RESEQUENCE  100,  0,  10
```

4.12.4. EDIT MERGE, EDIT WEAVE, EDIT INSERT

EDIT MERGE, WEAVE, and INSERT are used to combine from two to nine saved programs into one current program. The general format for these instructions is as follows:

EDIT type NAME1, NAME2, line2, NAME3, line3,...where type is MERGE, WEAVE, or INSERT, and NAME1 is the name of the saved program to be used as the main program. NAME2, NAME3,... are the other saved programs, which are to be inserted into NAME1 after line2, line3,...respectively. The resulting program is used as the current program.

(a) If the command is EDIT MERGE, the resulting program will be resequenced (as EDIT RESEQUENCE 100, 0, 10) without deleting similarly numbered lines, if they come from different sources. The sense of statement number references is preserved by renumbering these also.

(b) If the command is EDIT WEAVE, the resulting program will not be resequenced, but will be sorted. If a line number is duplicated, only the line from the added program with that number is retained.

(c) If the command is EDIT INSERT, the resulting program is neither sorted nor resequenced, until the next LIST or RUN is performed.

4.13. UNSAVE

The UNSAVE command is used to release saved programs in the users program file, and removes the users program name from the catalogue. The current program can be released by the second form following. The format of this command is as follows:

```
UNSAVE:NAME
```

or

```
UNSAVE
```

4.14. PUNCH

The current program or selected portions of it may be punched onto cards by this command. The general form of this command is the same as for the LIST command.

4.15. DEBUG

Two commands are available to alter the debug mode of operation. They are as follows:

(1) DEBUG ON, and
(2) DEBUG OFF.

They will turn the debug mode on and off, respectively. With the debug mode on, all TRACE and PAUSE statements are activated, while they are ignored when debug mode is off (see 3.25.2.).

APPENDIX A. OPERATING PROCEDURES

In order to use UBASIC, it is first necessary to understand the procedure necessary for running on a UNIVAC 1100 series system. The fundamental procedure from a teletypewriter remote station is as follows:

(1) Push the ORIG button and wait for a dial tone, then dial one of the numbers used for accessing the 1100 series system.

(2) When the connection is made, a high pitched tone will be heard. Following this tone, press the HERE IS key or in some way supply a legal site ID.

(3) If the 1100 series system is accepting teletypewriter communications, the system will respond with:

UNIVAC 1100 TIME/SHARING EXEC

(4) If the message in Step 3 above is printed, the 1100 series system is ready to accept the UNIVAC 1100 series Executive RUN control statement which must be the first line submitted by the user. The RUN control statement contains such information as user's run identification, account number, name or project, time estimate, and possibly other items which are not normally required. (See *UNIVAC 1100 Series Operating System Programmer Reference, UP-4144* (current version).) The following is the general form.

Example:

@RUN IIIIII,XXXXXX,NAME,TIME

where:

@RUN	appears in column position 1,2,3, and 4.
IIIIII	must be preceded by at least one space (blank), and consists of up to six alphanumeric characters. It is used for reference by operators and dispatchers.
XXXXXX	is the user project number assigned by the Computer Center.

NAME is up to a twelve character consistent spelling of an identification of the project.

TIME is the maximum time in minutes the run is expected to take for completion. This time is only with respect to central processor time, and is independent of the actual time of day start and completion times.

The time estimate (TIME), is optional, and if not specified by the user, then a standard value is automatically assigned.

An example of the @RUN statement is as follows:

```
@RUN      123456,000000,USER,2
```

which specified a run with a time estimate of two minutes. When the RUN statement is accepted, the system will respond with the date and time as follows:

```
DATE:  021871          TIME:  120000
```

(5) The next line required is one of the various system control statements recognized by the Executive. Since it is desired to use UBASIC, the next line is as follows:

```
@BASIC      FILENAME.
```

which gives control to the UBASIC system. UBASIC will respond with its version number, the date, and time. The new program name NAME$ is assumed, and UBASIC is ready for the first system command or the first line of a program.

FILENAME. is the name in the Executive filename notation [[Q] *] [F] [([FC])] [/ [RK] [/ [WK]]] [.] of a program file which is either catalogued or already assigned to the run. A user TPF$ is also automatically attached to the processor. Since TPF$ is always assigned, the FILENAME. field on the BASIC processor line is optional. Note that normally an up to 12 character alphanumeric field followed by the period will be sufficient.

Example:

```
@BASIC
@BASIC      FILE123.
```

After the user has terminated a line (system command or statement) the UBASIC processor will process the line appropriately. If an error is detected in a statement or some particular response is necessary to a system command, UBASIC will send a specific message to the user. He may then respond with either a new system command or statement. If no such error or particular response is necessary, UBASIC will respond with the greater than sign (>), which the user should be careful to wait upon prior to typing in his next statements. The > merely indicates that the last line has been accepted, was correct, the function carried out and another line is acceptable. If SEQUENCE has been specified to provide automatic line numbers, then the line number substitutes for the >.

(6) After the user finishes with UBASIC and wishes to sign off, he types the command BYE or GOODBYE. UBASIC will then sign off with the closing data and time. Then to disconnect from the 1100 series system, the user types an Executive control statement as follows:

@FIN carriage return

This closes the run, prints the elapsed computing time, sign on time, sign off time, console messages (if any) and finally the line:

* * * *LINE INACTIVE* * * *

Then the control shift is used to type an EOT (end of transmission). This disconnects the teletypewriter.

NOTE: If at any time during a run, the teletypewriter ceases line feeding, remote communications are discontinued and the teletypewriter should be disconnected.

APPENDIX B. IMPLEMENTATION RESTRICTIONS

In UBASIC the following restrictions apply:

(1) FOR's may only be nested 32 deep.

(2) DEF functions may have no more than 34 arguments.

(3) Recursion of functions may occur to a level of 50 to 100 deep, depending on the type and number of arguments.

The installation may alter restrictions on the following (standard values are in parentheses):

(1) Depth of GOSUB nesting (50).
(2) Largest matrix which may be inverted (50 x 50).
(3) Maximum number of lines in a single program (500).
(4) Size of the data array (200).
(5) Size of storage reserved for string constants (500).
(6) Size of storage reserved for arithmetic constants (200).

Approximately 4000 words of storage are reserved for the total program, the size of which may be estimated as follows:

P/6 + 11*A1+10*S+A2+W

where:

P is the number of characters in the symbolic program.
A1 is the number of elements in string arrays.
S is the number of simple strings references.
A2 is the number of elements in algebraic arrays.
W is the number of machine instructions generated by the UBASIC – approximately five times the number of UBASIC statements in the program.

If the program size is exceeded, UBASIC will automatically request assignment of additional core blocks of 512 words (via MCORE$) informing the user. This process will be repeated as necessary up to an installation set maximum (standard of 2).

APPENDIX C. UBASIC SYNTAX IN BACKUS NORMAL FORM

<null> ::=

<character> ::= any FIELDATA characters

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<integer> ::= <digit>|<digit><integer>

<statement number> ::= <integer>[1]

<statement reference> ::= <statement number>| *+<integer> | *–<integer>[1]

<sign> ::= <null> | + | –

<number> ::= <integer>| .<integer> | <integer>. |<integer>.<integer>|
<sign><number>

<exponent part> ::= <null> | E <sign> <integer> | %[2]

<arithmetic constant> ::= <number> <exponent part>

<character string> ::= <null> | <character> | <character string> <character>

<string constant> ::= '<character string>'[3]

<function name> ::= FN <letter>

<arithmetic array variable> ::= <letter>

<string array variable> ::= <letter>$

---

[1] value of integer ≤ 99999

[2] % is equivalent to E–2

[3] a ' within a string constant must be represented by '' (two single quotes).

<array variable> ::= <arithmetic array variable> | <string array variable>

<simple arithmetic variable> ::= <letter> | <letter> <digit> | <function name>[4]

<simple string variable> ::= <letter>$ | <letter> <digit>$

<simple variable> ::= <simple arithmetic variable> | <simple string variable>

<subscripted arithmetic variable> ::= <arithmetic array variable>(<expression>) |
<arithmetic array variable>(<expression>,<expression>)

<subscripted string variable> ::= <string array variable>(<expression>)

<arithmetic variable> ::= <simple arithmetic variable> |
<subscripted arithmetic variable>

<string variable> ::= <simple string variable> | <subscripted string variable>

<variable> ::= <arithmetic variable> | <string variable>

<system function name> ::= ABS | ATN | COS | COT | DET | EXP | INP | INT | LGT | LOG | MOD | NUM | RND |
SGN | SIN | SQR | TAN | CLK | CNT | COL | COS | COT | CSF | DAT | DET | DIG |
EXP | FRP | LEN | MAX | MIN | MXL | SEP | SER | STD | TIM | TIS | TYP | TWT |
XPT | VAL | VAS | CAT$ | DTS$ | PAD$ | TRM$ | EXT$ | CPY$ | ADD$ | PUT$ |
STR$ | NOT | AND | IOR | XOR | EQV | IMP | LSS | LEQ | GTR | GEQ | EQU | NEQ |
FLD | INS | TAB |

<formal parameter list> ::= <simple variable>[5] |
<simple variable>[5], <formal parameter list>

<parameter list> ::= <expression> | <expression>, <parameter list> |
<string expression> | <string expression>,<parameter list>

<function> ::= <system function name> | <function name>

<function value> ::= <function>(<parameter list>) | <function>[6]

---

[4] a function name is a simple variable only within the body of its definition.

[5] function names may not appear in parameter lists.

[6] the type and number of expressions in the parameter list are determined by the function.

<arithmetic operators> ::= **|*|/|+|—[7]

<term> ::= <arithmetic variable> | <function value> | <arithmetic constant>

<expression> ::= <sign> <term> | <expression> <arithmetic operator> <expression> |
<sign> <expression> | <sign>(<expression>)

<string expression> ::= <string variable> | <string constant>

<let head> ::= <arithmetic variable> | <arithmetic variable>=<let head>

<string let head> ::= <string variable> | <string variable>=<string let head>

<explicit let> ::= LET <let head>=<expression> |
LET <string let head>=<string expression>

<implied let> ::= <let head>=<expression> | <string let head>=<string expression>

<let> ::= <explicit let> | <implied let>

<variable list> ::= < variable> | <variable>,<variable list>

<read> ::= READ <variable list>

<input> ::= INPUT<variable list>

<print item> ::= <null> | <string expression> | <expression> |
<string constant> <expression> | TAB(<expression>)

<print item list> ::= <print item> | <print item>,<print item list> |
<print item>;<print item list>

<print> ::= PRINT <print item list>

<data item> ::= <arithmetic constant> | <string constant> | <character string>[8]

<data list> ::= <data item> | <data item>,<data list>

---

[7]operators are listed in order of precedence; operators with equal precedence in an expression are evaluated from left to right in the absence of parentheses.

[8]this character string may not contain commas.

<data> ::= DATA<data list>

<relational operator> ::= < > | > < | = | < | > : <= | =<|>= |=>

<relation> ::= <expression> <relational operator> <expression>|
<string expression> <relational operator> <string expression>

<go to> ::= GO TO <statement reference>

<transfer> ::= GO TO<statement reference> | THEN <statement reference>

<if> ::= IF<relation> <transfer>

<statement reference list> ::= <statement reference>|
<statement reference>,<statement reference list>

<on> ::= ON<expression> <transfer>,<statement reference list>

<step specification> ::= BY<expression>|STEP<expression>

<for specification> ::= TO<expression>|TO<expression> <step specification>|
<step specification>TO<expression>

<for> ::= FOR<simple arithmetic variable>=<expression> <for specification>

<next> ::= NEXT<simple arithmetic variable>

<dimension item> ::= <arithmetic array variable>(<integer>) |
<string array variable>(<integer>) |
<arithmetic array variable>(<integer>,<integer>)

<dimension list> ::= <dimension item> | <dimension item>,<dimension list>

<dim> ::= DIM<dimension list>

<stop> ::= STOP

<def> ::= DEF<function name>(<formal parameter list>)=<expression>|
DEF<function name>(<formal parameter list>)[9] |
DEF<function name>=<expression>| DEF<function name>[9]

<fnend> ::= FNEND

---

[9]this is used to start a multi-lined defined function which ends with an FNEND; the body of the DEF is between the DEF and the FNEND statements.

<gosub> ::= GOSUB<statement reference>

<return> ::= RETURN

<call> ::= CALL<function name>(<parameter list>) | CALL<function name>[10]

<mat operator> ::= + | – | *

<mat function> ::=TRN(<arithmetic array variable>) | ZER | CON | IDN |
INV(<arithmetic array variable>) | ZER(<expression>) |
ZER(<expression>,<expression>) | CON(<expression>) |
CON(<expression>,<expression>) | IDN(<expression>, <expression>

<mat let> ::= <arithmetic array variable>=<arithmetic array variable> |
<arithmetic array variable>=<mat function> |
<arithmetic array variable>=<arithmetic array variable>
<mat operator> <arithmetic array variable> |
<string array variable>= <string array variable>

<mat assign> ::= MAT<mat let>

<mat print list> ::= <array variable> | <array variable>,<mat print list> |
<array variable>;<mat print list>

<mat print> ::= PRINT<mat print list> | PRINT<mat print list>, |
PRINT<mat print list>;

<mat input> ::= INPUT<array variable> | INPUT<array variable>(<expression>) |
INPUT<arithmetic array variable>(<expression>,<expression>)

<mat read> ::= READ<array variable> | READ<array variable>(<expression>) |
READ<arithmetic array variable>(<expression>,<expression>)

<mat input-output> ::= MAT<mat print> | MAT<mat input> | MAT<mat read>

<change> ::= CHANGE<array variable>TO<string variable> |
CHANGE<string variable>TO<array variable>

<comment> ::= <character string>

<rem> ::= REM<comment> | <statement>#<comment >

<restore> ::= REST |REST* | REST$ | RESTORE | RESTORE* | RESTORE$

---

[10]the type and number of expressions in the parameter list are determined by the function.

<end> ::= END

<assign statement> ::= <let>|<mat assign>|<change>

<program control statement> ::= <go to>|<if>|<on>|<for>|<next>|<stop>|
<call>|<gosub>|<return>

<input-output statement> ::= <read>|<print>|<input>|<data>|<restore>|
<mat input-output>

<directive statement> ::= <dim>|<end>|<def>|<fnend>|<rem>

<statement> ::= <assign statement>|<program control statement>|
<input-output statement>|<directive statement>

<BASIC statement> ::= <statement number> <statement>|
<statement number> <statement>#<comment>

APPENDIX D. EXAMPLES OF THE USE OF UBASIC

EXAMPLE 1. BALLISTICS:

This example, BALLISTICS, demonstrates some of the algebraic capabilities of UBASIC. The problem is to find the correct elevation for a gun, given its muzzle velocity and the range to the target. The notes below are keyed to the listing as follows:

NOTES:

Line 1 – *The Basic processor is called.*

Line 2 – *The problem name "BALLISTICS" is specified.*

Line 3 – *This line defines a function the roots of which yield the desired answer.*

Line 4 – *The answer in radians is returned by the function FNS.*

Line 5 – *This line begins the definition of FNS which finds a root of the function FNA, given an interval in which there is a change of sign in FNA.*

Line 6 – *Execution of the program is attempted.*

Line 7 – *The first set of data yields the expected answer of about 30 degrees.*

Line 8 – *The second set of data returns an answer of about 45 degrees, the maximum.*

Line 9 – *The third set of data poses an impossible situation, resulting in the message designed for such a contingency.*

Line 10 – *Satisfied that the program works, the programmer signs off.*

Line 11 – *The @FIN card terminates this particular run. The program listing and run output appear below.*

```
@BASIC                                                          (Line 1)
BASIC VX.X 15:44:27  18 FEB 71
NEW:BALLISTICS                                                  (Line 2)
 READY
5 DEF FNA(A)=-9.8*X/(V*COS(A))+2*SIN(A)*V                       (Line 3)
20 PRINT 'WHAT IS THE DISTANCE(M) AND MUZZLE VELOCITY(M/S)';
30 INPUT X,V                    @GET THE STARTING INFO
75 T=FNS(0,3.14159265/4.0)      @DEFINE THE INTERVAL 0-45D      (Line 4)
```

```
160 PRINT 'THE ELEVATION IS'T*180/3.1419265;'DEGREES.'
170 PRINT X/(COS(T)*V);  'SECONDS ARE REQUIRED IN TRAVEL'
180 GO TO 20                    @START A NEW PROBLEM
185 PRINT 'NO SOLUTION IS POSSIBLE'
190 GO TO 20
1000 DEF FNS(E1,E2)             @FUNCTION TO FIND ROOTS OF FNA          (Line 5)
1005 FOR I=1 TO 20              @GO THROUGH 20 ITERATIONS
1010 FNS=(E1+E2)/2              @USING A HALF-INTERVAL METHOD
1020 IF FNA(FNS)*FNA(E1)<=0 THEN 1060
1030 IF FNA(FNS)*FNA(E2)> 0 THEN 1200 @NO SOLUTION POSSIBLE
1040 E1=FNS                     @SET A NEW INTERVAL
1050 GO TO 1070
1060 E2=FNS
1070 NEXT I
1080 GO TO 1300
1200 PRINT 'NO SOLUTION IS POSSIBLE'
1250 FNS=0
1300 FNEND
9999 END
RUN                                                                     (Line 6)


BALLISTICS 15:44:27  18 FEB 71

WHAT IS THE DISTANCE(M) AND MUZZLE VELOCITY(M/S)?
88167,1000
THE ELEVATION IS 29.8829 DEGREES.                                       (Line 7)
 101.687 SECONDS ARE REQUIRED IN TRAVEL
WHAT IS THE DISTANCE(M) AND MUZZLE VELOCITY(M/S)?
102040,1000
THE ELEVATION IS 44.0552 DEGREES.                                       (Line 8)
 141.984 SECONDS ARE REQUIRED IN TRAVEL
```

```
WHAT IS THE DISTANCE(M) AND MUZZLE VELOCITY(M/S)?
100,1                                                    (Line 9)
NO SOLUTION IS POSSIBLE
THE ELEVATION IS 0 DEGREES.
 100 SECONDS ARE REQUIRED IN TRAVEL
WHAT IS THE DISTANCE(M) AND MUZZLE VELOCITY(M/S)?.
BYE                                                      (Line 10)

BASIC OFF AT 15:44:27  18 FEB 71

@FIN                                                     (Line 11)
```

EXAMPLE 2. ACKERMANNS:

This example illustrates the recursive definition of Ackermanns Function A(M,N). The function is defined as follows:

$$A(0,N) = N+1$$
$$A(M,0) = A(M-1,1)$$
$$A(M,N) = A(M-1,A(M,N-1))$$

The program is coded to compute and print I,J, and A(I,J) for I = I1,I1+I3, I1+2*I3,...,I2; and J = J1,J1+J3, J1+2*J3,...,J2. The values taken for I1,I2,I3,J1,J2, and J3 appear on the DATA statement (Line 80). It is likely that available core space will be exhausted before all specified values are used.

The listing of the program appears immediately as follows:

```
@RUN        123456,ØØØØØØ,ACKERMANN
@BASIC
NEW:ACKERMANNS
    1  DEF FNA(M,N)
    2  IF  M=0 THEN 6
    3  IF  N=0 THEN 8
    4  FNA=FNA(M-1,FNA(M,N-1))
    5 GO TO 9
    6 FNA=N+1
    7 GO TO *+2
    8 FNA=FNA(M-1,1)
    9 FNEND
10 READ I1,I2,I3,J1,J2,J3
20 FOR I=I1 TO I2 STEP I3
```

```
30 FOR J=J1 TO J2 STEP J3
40 PRINT I,J,FNA(I,J)
50 NEXT J
60 NEXT I
70 STOP
80 DATA 0,4,1,0,6,1
90 END
@FIN
```

EXAMPLE 3. SORT:

This example reads numbers via the MAT INPUT statement, sorts the numbers and prints them via the MAT PRINT statement. Also illustrated is the NUM function which is automatically assigned a value equal to the number of data items read by the MAT INPUT statement. Note that, & is used on the first sample data line to continue the data to the next line. The program and sample run are listed as follows:

```
10 DIM A(100)
20 MAT INPUT A
30 PRINT 'THERE WERE 'NUM;' ITEMS READ.'
40 PRINT 'SORTED ARRAY:'
45 F=1
50 FOR I=1 TO NUM-1
60 IF A(I)<=A(I+1) THEN 90
70 T=A(I)
75 A(I)=A(I+1)
80 A(I+1)=T
85 F=0
90 NEXT I
100 IF F=0 THEN 45
120 MAT PRINT A
130 GO TO 20 #REPEAT FOR ANY NUMBER OF ARRAYS
999 END
RUNNH
? -1,.2345,0,1E22,5E-9,-4.23E-3,&
? 21.3,45,999,.25,.5
THERE WERE 11 ITEMS READ.
```

```
SORTED ARRAY:
-1
-4.23 E-3
0
5.0 E-9
.2345
.25
.5
21.3
45
999
1.0 E+22

? STOP
```

EXAMPLE 4. ARITHMETIC:

This example consists of a simple computer aided instruction program for teaching young students the four basic arithmetic operations. The program allows the student to rate himself as good, fair, or poor, and generates problems of difficulty as specified by the student. The program may be easily changed to make problems more (or less) difficult, by modifying the constant in line 250.

As may be noted from the listing, this program is quite short relative to what it accomplishes. This leads one to speculate that the UBASIC language may be good for programming simple CAI courses. The most notable feature is that of printing strings and numbers in natural fashion, without regard to formatting.

A listing of the program and part of the run are listed as follows:

```
LIST

ARITHMETIC  20:57:44  18 FEB 71

100 DIM S$(4),T$(6),Q$(3),A(3)
110 MAT READ S$,T$,Q$
115 RANDOMIZE
120 PRINT
130 PRINT
140 PRINT 'NEXT STUDENT, PLEASE.'
150 PRINT 'WHAT IS YOUR NAME:'
160 INPUT N$
170 PRINT 'HI, ';N$;',';
```

```
180 PRINT 'HOW IS YOUR ARITHMETIC:  GOOD, FAIR OR POOR';
190 INPUT R$
200 FOR Q=1 TO 3
210 IF R$=Q$(Q) GO TO 250
220 NEXT Q
230 PRINT 'I SAID: ';
240 GO TO 180
250 LET Z = 7*Q
260 LET S=INT((2*INT(Q/2)+2)*RND+1)
270 LET W=0
280 LET A(1)=INT(Z*RND/(INT(S/3)+1))+1
290 LET A(2)=INT(Z*RND/(INT(S/3)+1))
300 LET A(3)=A(1)+A(2)-(A(1)-A(1)*A(2)+A(2))*INT(S/3)
310 PRINT A(2*INT(S/2)-S+3);S$(S);A(1);' = ';
320 INPUT C
330 IF C<>A(2+S-2*INT(S/2))GO TO 360
340 LET Z=1.05*Z
350 GO TO 260
360 LET W=W+1
370 PRINT T$(2*W-1);N$;T$(2*W)
380 ON W GO TO 310,310,120
500 DATA '+','-','X','/','NO',',',TRY AGAIN.'
510 DATA 'THATS TWICE NOW, ',',TRY ONCE MORE.',' '
520 DATA ', YOU HAD BETTER BRUSH UP ON YOUR ARITHMETIC!'
530 DATA 'POOR', 'FAIR', 'GOOD'
540 END

RUN

ARITHMETIC 21:00:37  18 FEB 71

NEXT STUDENT, PLEASE.
WHAT IS YOUR NAME?  MARSIEDOTES
HI, MARSIEDOTES, HOW IS YOUR ARITHMETIC:  GOOD, FAIR OR POOR?  FAIR
   17 - 13 = ? 4
   13 - 12 = ? 1
```

```
   3 / 3 - ? 1
   1 x 7 = ? 7
   0 x 4 = ? STOP
   PROGRAM STOPPED.
   TIME:    .055

RUN

ARITHMETIC  22:47:48  18 FEB 71

NEXT STUDENT, PLEASE.
WHAT IS YOUR NAME?  HERB
HI, HERB, HOW IS YOUR ARITHMETIC:  GOOD, FAIR OR POOR?  GOOD
   23 - 13 = ? 10
   0 x 10 = ? 0
   8 x 8 = ? 64
   19 + 12 = ? 31
   0 + 12 = ? 12
   0 x 8 = ? 0
   10 + 24 = ? 34
   156/13 - ? 11
NO, HERB, TRY AGAIN.
   156 / 13 = ? 13
THAT'S TWICE NOW, HERB, TRY ONCE MORE.
   156 / 13 = ? 22
HERB, YOU HAD BETTER BRUSH UP ON YOUR ARITHMETIC!

NEXT STUDENT, PLEASE.
```

EXAMPLE 5. MAGIC SQUARE:

This example program computes and prints magic squares of odd order. The squares are called magic because no two numbers in the array are equal and the sum of any row, column, or diagonal is constant for any particular square.

The function to compute the square is defined by statements 80 through 998. As written, the program requires the size of the matrix to be less than 12 so that the square can be printed correctly on teletypewriters.

A good example of the TAB function appears in statement 1400; because, for readability, the numbers of the array should be right justified and evenly spaced over the row.

The row/column/diagonal sum is computed by statement 900.

The program and a sample run are listed as follows:

```
@RUN ONE,ØØØØØØ,MAGIC
DATE:  021871   TIME:  133715
@BASIC
BASIC VX.X  13:37:42   18 FEB 71
NEW;SQUARE
 READY
10       REM MAGIC SQUARE COMPUTATION.
20       DIM L(13,13)
30       GOSUB 1000
40       GO TO 30
80       DEF FNM(N,KO,L) @******START OF FUNCTION DEFINITION******
    82   REM GIVEN N ODD AND>2----KO AS START INTEGER---COMPUTE
    84   REM MAGIC SQUARE IN L(I,J)---USES N KO L S J K I M
    86   REM VALUE OF FUNCTION IS ROW/COLUMN/DIAGONAL SUM
    88   LET K=KO
    90   LET J=INT((N+1)/2)
    92   LET I=J+1
   100   M=1
```

```
  200  L(I,J)=K
  210  LET K=K+1
  220  LET M=M+1
  230  IF K=N*N+KO THEN 900
  240  IF M<=N THEN 300
  250  LET I=I+2
  260  IF I<= N THEN 100
  270  I=I-N
  280  GO TO 100
  300  I=I+1
  310  LET J=J+1
  312  IF I<= N THEN 300
  314  LET I=I-N
  320  IF J<= N THEN 200
  330  LET J=J-N
  340  GO TO 200
  900 FNM=((N*N+2*KO-1)*N)/2
988     FNEND @******END OF FUNCTION DEFINITION******
1000    PRINT
1010    PRINT  'TYPE ODD NUMBER<12 AND START NUMBER<800'
1020    PRINT  'AN EXAMPLE WOULD BE 3,1 TO GET 3x3 MAGIC SQUARE'
1100    INPUT N,KO
1110    IF N<12 THEN 1300
1200    PRINT  'TRY AGAIN'
1210    GO TO 1100
1300    IF KO>=800 THEN 1200
1310    LET S=FNM(N,KO,L)
1340    PRINT   'N=' , N,'---SUM=',S
1350    PRINT
1360    FOR I=1 TO N
1370    PRINT
1380    PRINT
1390    FOR J=1 TO N
1400    PRINT  TAB(5*J-2.001-LGT(L(I,J)));L(I,J);
1410    NEXT J
```

```
1415     PRINT
1420     NEXT I
1430     PRINT
1440     RETURN
1450     END

RUN
SQUARE     13:43:52  18 FEB 71
TYPE ODD NUMBER<12 AND START NUMBER<800
AN EXAMPLE WOULD BE 3,1 TO GET 3x3 MAGIC SQUARE
? 5,1
N=             5          ---SUM=         65
11     24       7     20       3
 4     12      25      8      16
17      5      13     21       9
10     18       1     14      22
23      6      19      2      15
TYPE ODD NUMBER<12 AND START NUMBER<800
AN EXAMPLE WOULD BE 3,1 TO GET 3x3 MAGIC SQUARE
? STOP
  PROGRAM STOPPED.
TIME:    .070
@FIN
```