Xerox BASIC

Sigma 5-9 Computers

Language and Operations
Reference Manual

90 15 46

XEROX

Xerox BASIC

Sigma 5-9 Computers

Language and Operations

Reference Manual

90 15 46G

August 1974

File No.: 1X23
XK14, Rev. 0
Printed in U.S.A.

NOTICE

This publication is a revision of the Xerox BASIC/LN,OPS Reference Manual for Sigma 5-9 Computers, Publication Number 90 15 46F (dated July 1973). A change in text from that of the previous manual is indicated by a vertical bar in the margin of the page.

RELATED PUBLICATIONS

Title	Publication No.
Xerox Sigma 5 Computer/Reference Manual	90 09 59
Xerox Sigma 6 Computer/Reference Manual	90 17 13
Xerox Sigma 7 Computer/Reference Manual	90 09 50
Xerox Sigma 8 Computer/Reference Manual	90 17 49
Xerox Sigma 9 Computer/Reference Manual	90 17 33
Xerox Batch Processing Monitor (BPM)/BP, RT Reference Manual	90 09 54
Xerox Batch Processing Monitor (BPM)/OPS Reference Manual	90 11 98
Xerox Batch Time-Sharing Monitor (BTM)/TS Reference Manual	90 15 77
Xerox Control Program-Five (CP-V)/TS Reference Manual	90 09 07
Xerox Control Program-Five (CP-V)/OPS Reference Manual	90 16 75

Manual Content Codes: BP - batch processing, LN - language, OPS - operations, RP - remote processing, RT - real-time, SM - system management, TS - time-sharing, UT - utilities.

CONTENTS

1. BEGINNING BASIC

INTRODUCTION

To use a computer, the user must learn a language the computer understands. Xerox Sigma 5-9 computers understand several languages. Most of these are meant for some special purpose such as the solution of scientific, engineering, or business problems. BASIC is intended as an all-purpose language. Although BASIC is often called a "beginner's language", the computational power of a given BASIC program depends a great deal on the experience of the user. An experienced BASIC user should have no difficulty in creating very powerful programs.

Because of its similarity to ordinary English, BASIC is a good language for users who are not professional programmers and who may have no particular interest in the internal workings of the computer. Many BASIC programmers never see the computer they are programming, but communicate by means of a teletypewriter terminal at a remote location. To use a terminal, the user must dial the telephone number of a Sigma 5-9 time-sharing computer and wait for a log-on request to be printed on the teletypewriter. The user must then type his account number, identifier, and account password (if any), followed by a carriage return. An example is shown below.

```
XEROX CP-V AT YOUR SERVICE
ON AT 14:32 JUN 28,'73
LOGON PLEASE: 99,PAT

14:34 06/28/73 99 43-25  [1]

!BASIC
VER.C01
 >
```

As shown above, the computer types a page heading and then an exclamation mark to indicate that it is ready for an executive-level command. The user types the word BASIC to indicate that he wants to use the BASIC subsystem. BASIC then responds with a > to indicate that it is ready to accept input from the user terminal.

SYMBOLIC NAMES

BASIC recognizes symbolic names representing mathematical variables. Such names may consist of a single letter of the alphabet or a letter followed by a single digit from 0 through 9:

```
X
Y
B4
```

The following are not valid names in BASIC:

```
XX
Y23
4D
```

ADDITION AND SUBTRACTION

Suppose you want to add a series of numbers such as 27.3, 14.1, 6.0, 3.5, and 36.25. One way of doing this is by placing BASIC in the "desk calculator" mode by typing a PRINT statement expressing the desired addition. BASIC will respond by computing the indicated sum and printing the total when a carriage return is typed:

```
>PRINT 27.3+14.1+6+3.5+36.25
 87.1500
>
```

Since BASIC statements cannot be continued from one line to the next, this method will work only if all of the numbers to be added can be typed on a single line of 132 characters. The consequences of this restriction can be avoided by letting part of the sum be represented by a symbolic variable such as the letter P:

```
>LET P=27.3+14.1+6
>PRINT P+3.5+36.25
 87.1500
>
```

The symbolic variable P could have been redefined to represent the final sum before typing the PRINT statement

```
>LET P=27.3+14.1+6
>P=P+3.5+36.25
>PRINT P
 87.1500
>
```

The statement

```
>P=P+3.5+36.25
```

is not a mathematical equation in the usual sense. A LET statement in BASIC is actually an "assignment" statement specifying that the current value of the symbol to the left of the "equals" sign is to be replaced by the value of the expression to the right of the equals sign. Note that the word LET in an assignment statement is optional.

Quantities can be subtracted by using a minus sign rather than a plus sign

```
>PRINT 10-13
-3
>
```

As in addition, symbols may be used to represent values in an expression involving subtraction:

```
>LET A=10, B=13
>PRINT A-B
-3
>
```

Note that more than one value assignment may be made in a single LET statement, as shown above, if a comma is used to separate each such assignment. If the above assignment had been written as

```
    >LET B=A+3, A=10
```

the value of B would be unpredictable, because BASIC executes LET statements from left to right, and A is not assigned the value 10 until the second assignment of the LET statement is performed.

MULTIPLICATION AND DIVISION

Multiplication and division can be done in much the same way as addition and subtraction. The asterisk is used to indicate multiplication, and the slash is used to indicate division. Thus, the product of 2 and 4 could be obtained as shown below.

```
    >PRINT 2*4
     8
    >
```

Parentheses can be used to group two or more quantities:

```
    >PRINT 3*(4+5)
     27
    >
```

Without the parentheses, the above PRINT statement would have produced the value 17 rather than 27, because BASIC would then assume that the value 5 was to be added to the product of 3 and 4. It would not be possible to avoid the use of parentheses by a rearrangement of the above expression to put the addition to the left of the multiplication, as in:

```
    >PRINT 4+5*3
```

This statement would produce the value 19 rather than the desired 27, since BASIC performs any indicated multiplication or division before doing addition or subtraction unless the order of precedence is indicated explicitly by means of parentheses.

Nested parentheses are evaluated from the innermost to the outermost:

```
    >PRINT 2*(3+4*(5+6))/7
     13.4286
    >
```

In the above example the "innermost" subexpression is 5+6. This sum is evaluated first and the result is multiplied by 4 and then added to 3 before the multiplication by 2 is performed. The final operation before printing is the division by 7. Note that the result is rounded to 6 significant digits.

EXPONENTIATION

Exponentiation is indicated by use of the up-arrow operator (↑) or the double asterisk (**):

```
    >PRINT 10**2
     100
    >
```

Within the same level of parenthesization, exponentiation takes precedence over any other indicated operation. That is, it is performed before multiplication, division, addition, or subtraction unless this would conflict with the grouping indicated by parentheses:

```
    >PRINT 2**2*3**(4-2)
     36
    >
```

In the above example, the first operation performed by BASIC is the raising of 2 to the second power. The sum of 4 and -2 is then computed and the quantity 3 is raised to this power. The final operation before printing is the multiplication of 4 (the square of 2) by 9 (the square of 3).

INDEXED REPETITION

Many applications of BASIC require a series of operations to be performed more than once. To make this as easy as possible, two special statements are provided: FOR and NEXT. The FOR statement specifies the conditions under which the repetition is to be done and the NEXT statement indicates the end of the series of BASIC statements that is to be repeated. FOR and NEXT must be used in the "compile and execute" mode rather than the desk calculator mode. For this reason, each line of the program must begin with a unique number ranging from 1 to 99999. Line numbers need not be contiguous, but lines are executed in ascending order. Many BASIC programmers prefer to begin each program with line 100 and make each line number a multiple of 10, allowing room for changes and additions to the program at a later time.

In the program shown below, FOR and NEXT statements are used to cause BASIC to print three lines. Note that RUN is typed following the program rather than preceding it.

```
    >10 FOR I=1 TO 3
    >20 PRINT I
    >30 NEXT I
    >RUN
    10:12    JUN 07    RUN2BAA...
     1
     2
     3

           30 HALT
    >
```

In the above example, the indexed variable "I" in the FOR statement is assigned an initial value of 1 (by the number 1 following the equals sign). Instead of the letter I, any letter of the alphabet could have been used, but the letters I through N have become traditional in FOR statements. When the NEXT statement is executed following the printing of the first line, the value of I is incremented by 1 automatically and the loop is executed again. Note that the FOR statement initiates execution of the loop but is not a part of it. When the NEXT statement is executed following the printing of the second line, I is again incremented by 1 and the loop is executed for the third time. When the NEXT statement is executed following the printing of the third line I is not incremented, since the limiting value of 3 has been reached, and the loop is not executed again. Since 4 is greater than 3, the limiting value following the word TO, the loop is not executed again. The message 30 HALT indicates that line 30 was the last line executed in the program.

Often in programming FOR and NEXT loops, one may take advantage of the fact that the indexed variable changes in value as the loop is repeatedly executed. This is illustrated by the following example.

```
>10 FOR I=1 TO 4
>20 PRINT I**3
>30 NEXT I
>RUN
10:13    JUN 07   RUN2BAA...
 1
 8
 27
 64

          30 HALT
>
```

In the above example, the cube of the indexed variable is printed each time the loop is executed.

The values of symbolic variables used within a loop can be changed during execution of the loop, as illustrated by the following program which prints the first 8 terms of a Fibonacci series. Note that the statements within the loop are typed indented, to make the extent of the loop more readily apparent. This optional practice is especially recommended for nested loops.

```
>10 LET J=0, K=1
>20 FOR I=K TO 8
>30   PRINT J
>40   M=J, J=K, K=K+M
>50 NEXT I
>RUN
10:14    JUN 07   RUN2BAA...
 0
 1
 1
 2
 3
 5
 8
 13

          50 HALT
>
```

Although the value of K varies as the above program is executed, this does not affect I, since the initial value of I is determined only when the FOR statement is executed and the loop is first entered. Note the use of an optional form of the LET statement, without the word LET, in line 40 of the program.

You may use a LET statement to alter the value of the indexed variable within a loop, in addition to the increment added automatically whenever the NEXT statement is executed.

```
>10 FOR I=1 TO 10
>20    LET I=2*I
>30    PRINT I
>40 NEXT I
>RUN
10:19    JUN 07   RUN2BAA...
 2
 6
 14

          40 HALT
>
```

In the above program, the initial value of I is 1, as specified by the FOR statement. The LET statement doubles this value and the NEXT statement adds 1 to it. Thus, I has the value 3 at the beginning of the second execution of the loop and 7 at the start of the third execution of the loop. When the NEXT statement is executed for the third time, I is not incremented because the limit of 10 has been exceeded, and loop execution stops.

PRINT FORMATTING

The statement

```
>PRINT
```

causes BASIC to print a single blank line. The statement

```
>PRINT X
```

causes the value of X to be printed, beginning in column 2. Column 1 is reserved for a possible minus sign. The statement

```
>PRINT X,Y
```

causes BASIC to print the value of X, beginning in column 2, and the value of Y on the same line, beginning in column 16.

If closer spacing is wanted, a semicolon can be used in place of a comma. The statement

```
>PRINT X;Y
```

causes BASIC to print the value of X, beginning in column 2, followed by the value of Y with 3 or 4 columns separating the two, so that the value of Y will begin in an even-numbered column.

If a value is negative, a minus sign precedes it.

```
>PRINT X;Y;Z;N4
 5  -10     300    20.5000
>
```

A combination of formats can be used.

```
>PRINT X,Y;Z
 5                -10     300
>
```

If the user does not want the first value to print in column 2 he can use a comma or semicolon following the word PRINT.

```
>PRINT ,X;Y
                   5  -10
>
```

The statement

```
>PRINT 'THESE WORDS'
```

could be used to print THESE WORDS beginning in column 1.

TABBING

The TAB function is used in PRINT statements to advance the output device to a specified column. For example, the statement shown below causes the teletypewriter to advance to column 12 and print the word HERE.

```
>PRINT TAB(12)"HERE"
           HERE
>
```

Note that HERE is a literal text string, identified as such by enclosure in quotes. Either single or double quotes (i.e., 'THIS' or "THIS") may be used to enclose a literal text string.

A symbolic or literal value may be used in the same PRINT statement as a literal text string (either with or without a TAB expression). However, since an expression must not follow another expression immediately, a symbolic or literal value must not be used next to a TAB expression. To avoid this difficulty, one can use an empty text string (e.g., "") to separate two expressions in a PRINT statement.

```
>PRINT TAB(6)""X" APPLES"
      5 APPLES
>
```

Note that column 6 will be blank only if X is positive.

If a comma or semicolon is used to separate a TAB expression from a following expression as in the statement

```
>100 PRINT TAB (5),X
```

BASIC will perform the TAB function but will assume that the punctuation is only being used as a separator.

A TAB expression may contain symbolic as well as literal values, allowing great flexibility in line format. This capability is very useful in programming for graphic output.

For example, the following program produces a graphic plot of a damped sine wave:

```
>100 X=X+.7,K=EXP(-X/15)
>110 PRINT TAB(15+15*K*SIN(X))"*"
>120 IF X<15 THEN 100
>RUN
10:43   JUN 07   RUN2BAA...

                            *
                          *
                   *
          *
     *
   *
        *
               *
                     *
                       *
                      *
                 *
            *
        *
        *
          *
               *
                   *
                    *
                   *
                *

      120 HALT
>
```

In this example, the EXP intrinsic function returns the value of e (the quantity 2.7183...) raised to the power of the argument. The IF statement causes BASIC to return to statement 100 unless the current value of X is equal to or greater than 15.

DATA INPUT FROM A TERMINAL

The INPUT statement is used to solicit input via the user terminal. A question mark is printed by BASIC to prompt the user.

```
>100 PRINT 'ENTER LENGTH AND WIDTH'
>110 INPUT L,W
>120 PRINT 'AREA='L*W
>RUN
10:46   JUN 07   RUN2BAA...
ENTER LENGTH AND WIDTH
?5, 10
AREA= 50

      120 HALT
>
```

In the above example BASIC prints a request to type values for length and width. The values 5 and 10 are typed following the prompt character. Note that the comma after the value 5 is optional. A single blank would be sufficient to separate the two values.

ERROR MESSAGES

BASIC messages to the user are explained in Chapter 6. Most of these messages inform the user of a syntax error in a program line, a logical error in program structure, or a pragmatic error in program execution. Syntax errors are detected when a line is typed, logical errors are detected at compile time, and pragmatic errors are detected at run time.

If the user types

```
    >100 X=2Y
```

BASIC prints the message

```
    100 BAD FORMAT
```

on the following line. The user can correct the line by retyping

```
    >100 X=2*Y
```

The program shown below contains a FOR statement without a corresponding NEXT statement.

```
    >100 FOR I=1 TO 5
    >110 PRINT ,'X='I;'Y='I**3
    >120 END
    >RUN
    10:49    JUN 07   RUN2BAA...
     MISSING NEXTSTMT
    >
```

When the above program is compiled, BASIC prints the message

```
    MISSING NEXTSTMT
```

The user can correct the program by typing

```
    >115 NEXT I
```

and then recompiling by typing another RUN command.

In executing the following program, a divisor becomes zero, causing BASIC to print the message shown below.

```
    >100 FOR X=1 TO 2
    >110 FOR Y=1 TO 4-X
    >120 Z=1/(X↑2+3*X*Y-X-Y↑2)
    >130 PRINT X,Y,Z
    >140 NEXT Y
    >150 NEXT X
    >RUN
    10:52    JUN 07   RUN2BAA...
     1               1               .500000
     1               2               .500000

          120 DIV BY ZERO
    >
```

The user can correct this by typing

```
    >115 IF X↑2+3*X*Y-X-Y↑2<>0 THEN 120
    >116 Z="  INF."
    >117 GOTO 130
    >
```

Note that line 116 above assigns a six-character literal text string to the name Z. Z then denotes an alphanumeric constant, or "aconst", as discussed in Chapter 2.

PROGRAM MODIFICATION

A line in a program can be changed by retyping the entire line. A new line can be added to a program by typing it, giving it any unused line number within the desired area of the existing program.

```
    >100 R=12, Y=-12.5
    >110 Y=Y+1, X=SQR(R↑2-Y↑2)
    >120 PRINT TAB(36-X)'*'TAB(36+X)'*'
    >130 IF Y<11.5 THEN 110
    >
```

The user could add a line between 110 and 120, using any line number from 111 through 119.

```
    >115 X=1.7*X
    >
```

If the user wanted to combine lines 110 and 115, in the above example, he could do so by retyping line 110 and deleting line 115 by typing the line number followed by a carriage return.

```
    >110 Y=Y+1, X=1.7*SQR(R↑2-Y↑2)
    >115
    >
```

Other methods of program modification are discussed in Chapter 4.

SAVING A PROGRAM

To save a program for future use, the SAVE ON command can be used:

```
    >SAVE ON FILENAME
    >
```

In the above example, the current contents of the text editing area are saved in a file named FILENAME if a file does not already exist with that name. If a file with that name does exist, the save does not occur and an error message is produced.

If the user wants to unconditionally save a program for future use, he can use the SAVE OVER command:

```
    >100 PRINT "HELLO, THIS IS 'FILENAME'."
    >SAVE OVER FILENAME
    >
```

In this example, the current contents of the text editing area are unconditionally saved in a file named FILENAME. If a file with that name already exists, it is replaced with the new file information.

File names may consist of up to 11 characters. They must not begin with a single quote (') or double quote (") character and must not contain blanks, unless the entire file name is delimited by quotes. Examples of valid file names are

```
>SAVE OVER "FILE NAME"
>SAVE ON ELEPHANTINE
>SAVE OVER ?@"/#↑!
>SAVE ON 2+2=4
>SAVE OVER THIS(2)
>
```

LOADING AND RUNNING A SAVED PROGRAM

Before loading a program, it is good practice to use the CLEAR command to clear the text editing area.

If CLEAR is not used the loaded program will be combined with whatever happens to be in the text editing area. After a program has been loaded it can be executed by giving a RUN command

```
>CLEAR
>LOAD FILENAME
>RUN
11:14   JUN 07   FILENAME...
HELLO, THIS IS 'FILENAME'.

      100 HALT
>
```

2. ELEMENTARY FEATURES OF BASIC

In the following chapters, certain conventions have been adopted for defining the BASIC commands. Capital letters indicate command words that are required in the literal form shown. Lowercase letters are figurative representations of constants, step numbers, etc. Command parameters enclosed by braces ({ }) indicate a required choice. Parameters enclosed by brackets ([]) are optional. Ellipsis marks (...) signify optional repetitions of the preceding bracketed parameter. BASIC recognizes the period as a decimal point, not as a terminator.

ELEMENTS OF A BASIC PROGRAM

There are a number of elements common to most BASIC programs. These are: line numbers, simple constants and variables, arithmetic operators, expressions, and intrinsic functions. In addition to these, BASIC programs involving text manipulation often use alphanumeric constants as well as string literals, variables, and expressions. String variables and expressions are not permitted in BTM BASIC.

LINE NUMBERS

Every line in a BASIC program must begin with a unique integer. Line numbers may range from one through 99999 but need not be contiguous, allowing for insertions. Lines are executed in ascending sequence, except where the sequence of execution is modified by branching or looping. Leading zeros are permitted in line numbers but are not required.

NUMBER RANGES

Because BASIC converts all input values to an internal double precision floating-point binary format, the appearance of input values and output values may differ due to rounding during input conversion. An example of this rounding is shown below.

```
>X=111111111111111
>PRINT X
 1.11111E+14
>Y=1/1000
>PRINT Y
 1.00000E-03
>Z=3E4
>PRINT Z
 30000
```

INPUT

BASIC handles input numbers within a range of 5.398*10↑-79 through 7.237*10↑75, and zero. Up to 16 significant decimal digits can be input.

OUTPUT

Output numbers are printed in fields of varying widths accross the page according to the following rules.

1. Numbers are left-justified in their fields.

2. Positive numbers are preceded by a blank, negative numbers by a minus sign.

3. If the number is a whole number (integer) whose magnitude is less than 1,000,000,000 (10^9), it is printed in from 1 through 9 positions after a blank or a minus.

4. If the number is nonintegral or its magnitude is greater than or equal to 10^9 (for example, -10.5, .5, 123.45, or 10^{12}), its most significant part will be rounded to 6 or 16 digits according to the PRC function and will be treated as follows:

   a. If, after rounding, the absolute value of the number is greater than or equal to 0.1 but less than 10^6 or 10^{16}, the number is printed, in 8 or 18 print positions, in fixed-point notation; that is, its form will be a blank or minus, a maximum of 6 or 16 integer digits, followed by the decimal point and a maximum of 6 or 16 digits.

   b. If, after rounding, the absolute value of the number is less than 0.1 or greater than 10^6 or 10^{16}, the number is printed in floating-point (scientific) notation; that is, its form will be a blank or minus, the most significant integer digit, a decimal point, 5 or 15 decimal digits, followed by the letter E, a plus or minus, and a 2-digit exponent.

PRECISION CONTROL

In the editing mode, output precision can be controlled by means of the ENTER BASIC command (see Chapter 4).

Output precision can also be controlled by use of the PRC intrinsic function in a PRINT statement. Used by itself or embedded in a series of other PRINT elements, PRC(1) sets the output precision to 16 significant figures. This precision remains in effect until reset to the default value of six by a PRC(0) or an ENTER BASIC command. An example is shown below.

```
>LET I=1/30
>PRINT I
 3.33333E-02
>PRINT PRC(1),I
 3.333333333333333E-02
>PRINT I
 3.333333333333333E-02
>PRINT PRC(0);I
 3.33333E-02
>
```

SIMPLE CONSTANTS

A simple constant (that is, a nonvarying quantity) is composed of digits that stand alone, have an embedded decimal point, or are preceded or followed by a decimal point. For example, 2, 7.8, .5, and 12 are simple constants in BASIC.

Simple constants may be modified by floating-point notation, as in 2.5E-15, where the E denotes that the number that precedes it is to be multiplied by 10 to the plus or minus power following the E. Accordingly, the number 2.5E-15 is really the number .0000000000000025. The plus sign is optional for positive powers.

SIMPLE VARIABLES

A simple variable is denoted either by a single letter or by a letter and a digit from 0 through 9. This convention allows the programmer a total of 286 simple variables. For example, A and W3 are simple variables. Note that if the letter-and-digit combination is used, the letter must precede the number.

ARITHMETIC OPERATORS

BASIC uses common mathematical symbols to denote arithmetic operations. These arithmetic operators are shown in Table 1 below. Note that either the up-arrow (or circumflex) or double asterisk is allowed as an exponentiation operator. The use of the double asterisk is convenient for batch operation, since the up-arrow requires an overpunch if cards are used as the input medium.

Table 1. Order of Arithmetic Operations

Order	Symbol	Explanation
1	↑ or **	Exponentiate
2	* and /	Multiply and Divide
3	+ and -	Add and Subtract

The table also shows the order of precedence of the various operations. When no operation takes precedence over another, the computer will perform operations from left to right. The order of operations may be altered by use of parentheses. Use of parentheses is advised if the sequence of operations seems questionable.

Note that an operator of order 1 or 2 may be followed by an operator of order 3, but that no other cases of consecutive operators are permitted.

INTRINSIC FUNCTIONS

BASIC provides a total of 34 intrinsic functions, 15 of which may not be used in BTM BASIC. The functions are listed in Appendix C. When a function is used in a statement, the three-letter function name must be followed by an expression or value enclosed in parentheses. This expression or value is called the "argument" of the function. The value of the argument is either used directly in the function calculation, or signals the computer to perform the calculation in a predetermined manner. The purpose of most of the functions is obvious and familiar. The INT function is often used to acquire the integer part of a calculated number. For example, INT(A), where A is computed to be 2.675, would produce the number 2. The INT function may also be used to obtain three significant digits (with rounding) as in the following example:

```
>50 LET S=INT((A*100)+.5)/100
>
```

When statement 50 is executed, S is assigned the value 2.68.

ARITHMETIC EXPRESSIONS

The term "expression", often abbreviated "expr", represents a simple constant, simple or subscripted variable (see Chapter 3), or function reference that may stand alone or may be used in any combination when separated by the symbols for addition, subtraction, multiplication, division, and exponentiation. The components may also be enclosed by parentheses. The symbols + and - may also be the initial character of an expression and may immediately follow a left parenthesis. Some typical expressions are

```
A+1
(B-X)/D
```

and

```
(2↑X)+SIN(Y)
```

STRING LITERAL

A string literal in a BASIC program is any sequence of text characters, including blanks, enclosed by single or double quotes. If a string literal is enclosed by single quotes, a single quote must not appear in the string. A parallel rule applies to the use of double quotes. Examples of string literals are shown below.

```
>100 PRINT 'THIS IS A STRING LITERAL'
>110 PRINT "THIS IS 'ANOTHER ONE'"
>120 PRINT 'AND THIS IS "A THIRD"'
```

ALPHANUMERIC CONSTANTS

Besides the simple constants previously mentioned, BASIC allows symbolic names such as X, R, or K2 to be assigned string literals of up to six characters. Such symbols are then called "alphanumeric constants" or "aconsts". For example,

```
>100 A='ACONST'
>110 Z='SPACE3'
>120 X6=' X RAY'
>
```

An "aconst value" can be assigned via a LET, INPUT, READ, GET, MAT READ, MAT INPUT, or MAT GET statement and can be tested, for equality or inequality only, via an IF statement.

```
>100 IF A='ALPHA' THEN 200
>110 IF A<>'OMEGA' THEN 300
>
```

An aconst value may occur in unquoted form as an element of a DATA statement or in the response to an INPUT request. If an element begins with a digit, plus or minus sign, or decimal point it is assumed to be a number. If it starts with any other character, or is enclosed in quotes, it is assumed to be an aconst value.

Example:

```
>100 DATA 5, '5', .5, 'FIVE'
>
```

The first and third elements in the above example are treated as numbers. The second and last are treated as aconst values.

Similar rules apply to elements entered in response to an INPUT request, except that in CP-V BASIC if the input is assigned to string variables (see below) all elements are interpreted as strings.

STRING SCALARS (CP-V Only)

String scalars are denoted by a letter and dollar sign. A string scalar consists of up to 132 characters comprising a single string. A string vector is a one-dimensional array, each element of which is a single string. A string matrix is a two-dimensional array of such elements. String vectors and matrixes are discussed in Chapter 3. To avoid conflict, the same letter must not be used to designate a string scalar and a string (or numeric) array.

Examples of string scalars are

```
>100 A$='A STRING'
>110 B$='ANOTHER STRING'
>
```

String scalars can be compared for relative magnitude as well as equality.

```
>100 A$='THIS'
>110 B$='THAT'
>120 IF A$=B$ THEN 200
>130 IF A$>B$ THEN 300
>
```

In this example, the comparison of A$ and B$ fails on the third character. Thus, since the character "A" is lower in the EBCDIC code than "I" (see Appendix H) a branch to line 300 is taken.

ASSIGNMENT STATEMENTS

This section discusses the assignment of simple variables, alphanumeric constants, and string scalars. The assignment of values to vectors and matrixes is explained in Chapter 3.

LET The LET statement replaces the current value of the variable(s) on the left of the equals sign with that of the expression on the right of the equals sign.

LET statements have the general form

```
[line][LET]variable[,variable]... = expression
    [, variable[,variable]... = expression]
```

where

variable is either a simple variable, an alphanumeric constant, or a string scalar.

expression is an arithmetic expression (see "Arithmetic Expressions", above) if the variable is a simple variable. For alphanumeric constants, expressions are either string literals of up to six characters or else string expressions assigning strings of up to six characters (see "Character String Manipulation", Chapter 3).

For string scalars, expressions are either string expressions or string literals of up to 72 characters or the limit set by a SET S command (see "SET", Chapter 4). Strings of excess length are truncated to the maximum permitted.

Arithmetic operations can be performed by use of the LET statement.

```
>100 A=A+1
>110 B=A↑2
>
```

Lines 100 and 110 above could be combined into a single statement.

```
>100 A=A+1, B=A↑2
>
```

Such serial assignments are executed from left to right. Thus, if A is initially 2 then B will be assigned a value of 9. Parallel assignments can also be made.

```
>110 A,B,C=D*E
>110 F,G=F+1
>
```

Note that line 110 above is equivalent to

```
>110 F=F+1, G=F
>
```

Other examples of LET statements are shown below.

```
>100 P,A1,Q3=4*ATN(1), A=1, B=5
>110 C='STRING'
>120 D$='A LONG STRING'
>130 E$=C, F=D$
>
```

In executing line 130 above, aconst F is assigned the string A LONG because of the aconst length limitation.

BRANCHING

Normally BASIC executes program lines in ascending order, beginning with the lowest numbered line. The statements discussed below cause BASIC to alter the normal order of execution, either conditionally or unconditionally. (See also "Branching to a Subroutine", in Chapter 3.)

IF . . . THEN The IF ... THEN statement provides a conditional branching capability. If the test condition specified in the IF ... THEN statement is true, then the next line executed is that specified in the IF ... THEN statement. Otherwise, the statement following the IF ... THEN statement in the normal sequence is executed.

The form of the IF ... THEN statement is

$$[\text{line}]\ \text{IF expr operator expr}\begin{Bmatrix}\text{THEN}\\ \text{GOTO}\end{Bmatrix}\text{line}$$

The condition to be tested is specified between the IF and the THEN (or GOTO) of the statement. The line to which BASIC is to branch on a true test is specified after THEN (or GOTO). An expression may be a simple constant, variable, alphanumeric constant, literal string or string scalar, or a compound arithmetic expression. The operators that may be specified are given in Table 2.

Table 2. Condition Operators

Operator	Explanation
=	Equal to
>< or <>	Not equal to
<	Less than
>	Greater than
<= or =<	Less than or equal to
>= or =>	Greater than or equal to

Examples of IF ... THEN statements are given below:

```
>100 IF X<2 THEN 120
>110 IF Y='EUREKA' THEN 130
>120 IF A$>Z$ THEN 140
>130 IF SIN(X+J)<=COS(X*K) GOTO 100
>140 IF B=C THEN 110
>
```

ON . . . GOTO If many different branches are to be taken according to the value of some expression, the use of a separate IF ... THEN statement (see above) for each branch becomes unwieldy. To overcome this inconvenience, BASIC provides the ON ... GOTO and GOTO ... ON statements.

ON ... GOTO takes the form

$$[\text{line}]\text{ON expr}\begin{Bmatrix}\text{GOTO}\\ \text{THEN}\end{Bmatrix}\text{line}[,\text{line}]\ldots$$

where

expr	is any arithmetic expression.
line[, line]...	is a list of program line numbers.

When the statement is executed, the expression is evaluated and, if necessary, truncated to an integer. If the resulting value is 1, a branch is made to the first line specified in the list. If the value is 2, a branch is made to the second line specified, and so on. If the value is less than 1 or greater than the total number of lines specified in the list, no branch is taken and the next statement in the normal sequence is executed.

ON ... GOTO may not be used as a direct statement in BTM BASIC.

Example:

```
>100 ON SGN(X)+2 GOTO 150,250,200
>
```

In the above example, if X is negative a branch to line 150 is taken, if X is 0 a branch to line 250 is taken, and if X is positive a branch to line 200 is taken.

GOTO . . . ON This statement is identical in operation to ON ... GOTO (see above) and may not be used as a direct statement in BTM BASIC. It has the form

[line]GOTO line[, line]... ON expression

Examples:

```
>100 GOTO 140,160,180 ON Y
>110 GOTO 200,250,300 ON Z+3
>
```

GOTO The GOTO statement can be used to alter the normal sequence of program execution unconditionally. The GOTO statement has the form

[line]GOTO line

GOTO is generally used in conjunction with a conditional branch such as IF ... THEN (see above). An example is shown below.

```
>100 IF TIM(1)<>12 THEN 100
>110 PRINT 'LUNCHTIME'
>120 IF TIM(1)=12 THEN 120
>130 GOTO 100
>
```

DATA OUTPUT

PRINT The PRINT statement tells the computer to print out the current value of a variable, the results of a calculation, a message, or any combination of these items. The output is produced on a line printer, typewriter, card punch, magnetic tape, Teletype, or other device. The PRINT statement has the general form

```
[line] {PRINT} expression(s), or text string(s) with—┐
       {  ;  }                                       │
  └—commas or semicolons ────────────────────────────┘
```

where the word PRINT is usually followed by the name of the item that is to be printed. Notice that a semicolon can be used in place of the word PRINT; this abbreviation is especially useful in the desk-calculator mode. Two sample print statements are

```
>60 PRINT X1,X2
>70 ; 'NO REAL ROOTS'
>
```

Line 60 will print out the calculated values of the variables X1 and X2. Line 70 will cause the message enclosed by single quotation marks to be printed out. Note that a string of text must be enclosed by either single or double quotation marks. Blanks may be used within the string and will be reproduced in the output as presented. More than one text string may be present in a PRINT statement. Each separate string, however, must be enclosed by quotation marks.

A PRINT statement may contain a reference to an intrinsic function. For example

```
>1220 PRINT SQR(X)
```

will calculate and print the square root of the variable X, while

```
>1230 PRINT X;SQR(X)
>
```

will print the current value of the variable, followed by its square root.

PRINT can also contain variables, providing that the variables have been defined in statements preceding the PRINT. The following statement is an example of this use of PRINT.

```
>1250 PRINT B*B-4*A*C
```

Similarly, the statement

```
>1260 PRINT (7/8)↑14
```

will give the value of the fraction 7/8 raised to the 14th power.

The word PRINT, used alone in a statement, causes the printer to advance the paper by one line. An example is shown below.

```
>450 PRINT
```

PRINT FORMATS

Punctuation marks in the PRINT statement (commas and semicolons) define the desired appearance, or format, of the printed output. The punctuation marks tell the print device at which position to start printing. BASIC has two types of output format, regular and packed. Regular format is specified by using commas to separate elements in the PRINT statement; packed format is specified by using semicolons.

Regular Format. When the regular format is specified by using commas to separate the elements in the PRINT statement, the print line is thought of as consisting of a series of 14-character fields. Each comma causes a shift to the next field. For example,

```
>4040 PRINT A,B,C
```

will cause the values of A, B, and C to be printed at 14-character intervals, as in

```
 1             5             4
>
```

When the regular format option is specified, at least two blanks follow the last printed character. In some cases, this spacing may cause an extra field shift.

Packed Format. Packed format, which is specified by using semicolons to separate the elements in the PRINT statement, causes the printed output to be compressed on the page by reducing the spacing between fields. Each semicolon causes a field shift that is either two or three positions in length, so that the shift reaches an even-numbered position. For example, statement 4040 above could be written as

```
>4040 PRINT A;B;C
```

with the resultant output of

```
 1   5   4
>
```

Additional Format Considerations. It is important to note the difference between the TAB function and the output format characters. TAB causes output to be printed at a specified position, and is most useful in providing columnar output. The output format characters (, and ;) cause the output to be printed at intervals that depend on the number of preceding (printed) characters.

If a PRINT statement terminates with a TAB to a column to the left of the current print position, e.g., TAB(0), the line is not printed until a subsequent PRINT is executed. This allows effective continuation of PRINT statements, except in the direct mode.

An expression may not follow another expression in a PRINT statement, but a text string literal may be used anywhere. Thus, a null string (two quotes) may be used to separate two expressions. For example,

```
>5 PRINT A(1)""B(2)
```

If a PRINT statement ends with a punctuation mark, the appropriate field shift takes place and subsequent printing starts at that point on the same print line. Otherwise, subsequent printing starts on a new line.[†] All printing is left-justified in its field. If a field shift places a field in a position to extend beyond the last allowed print position, a new line is generated and the field is printed on the new line.

This procedure is modified when printing text. If a text string overflows the last position, the string is truncated at that point and the remainder is printed on the next line.

Text strings may extend beyond any number of field boundaries. If neither a comma nor a semicolon appear on either side of a text string in a PRINT statement, no spacing will occur before or after the string in the printed output.

Format characters may be used alone in PRINT statements, or they may be used in any number and combination to cause appropriate field shifts.

PRINTUSING and :(Image) As an alternative to use of the PRINT statement (see above), BASIC provides another method of specifying the format of printed output. This method makes use of a PRINTUSING statement and an associated Image statement. The PRINTUSING statement contains parameters to be inserted into the print positions specified by the referenced Image statement.

PRINTUSING takes the general form

```
[line] {PRINTUSING}  line[, expression(s) or ──┐
       {;USING    }                            │
┌──────────────────────────────────────────────┘
└── text string(s)]
```

where the line number that follows the command word designates the Image statement into which this PRINTUSING's parameters will be embedded. Notice that a semicolon can be substituted for the word PRINT. Two sample PRINTUSING statements are shown below, one with the word PRINTUSING and the other with the abbreviated form ;USING.

```
>50 PRINTUSING 75,X,SQR(X),'SQ. ROOT'
>50 ;USING 75,X,SQR(X),'SQ. ROOT'
```

[†]In CP-V BASIC, a line ending with a punctuation mark is not printed until execution of the next PRINT statement that does not end with a punctuation mark.

Both of these statements perform the same action, of course. The parameters of line 50, that is, the current value of X, the square root of X, and the text string 'SQ. ROOT', will be embedded, from left to right, in the designated fields of line 75 (a field is a group of character positions that is treated as a distinct unit). Note that commas are used to separate parameters in PRINTUSING.

The Image statement (identified as such by a colon after the line number) complements the PRINTUSING statement in that Image statements depict the final printed appearance of PRINTUSING parameters. An Image statement has the form.

line :[#s and/or characters to 132 max.]

where the characters that follow the required colon are governed by the following rules.

1. Each digit position is designated by a # symbol. Also, text strings to be derived from the PRINTUSING statement are indicated by # symbols. For example, the statements

   ```
   >50 PRINTUSING 75,X,SQR(X),'SQ. ROOT'
   >75 :IF X=#, # IS ITS ########
   >
   ```

   will generate the following output (assuming X is currently 4):

   ```
   IF X=4, 2 IS ITS SQ. ROOT
   >
   ```

2. If a field is preceded by a plus sign, positive values are preceded by a plus sign and negative values by a minus sign. On the other hand, if a field is preceded by a minus sign, positive values are preceded by a blank, negative values by a minus sign. For example, the statements

   ```
   >5 :-##, -##, +##, +##, ##
   >14 PRINTUSING 5,-19,+20,-21,20,99
   ```

   will generate

   ```
   -19,  20, -21, +20, 99
   >
   ```

   If a field is preceded neither by a plus sign nor by a minus sign and the associated value is negative, a minus will be printed in the first position and any remaining positions of the field will contain asterisks.

3. The decimal point is denoted by a . symbol. For example, the statements

   ```
   >3 :#.# AND -.## ALSO +###.
   >23 PRINTUSING 3, 1.2, -1/4, 100.435
   ```

   will generate

   ```
   1.2 AND -.25 ALSO +100.
   >
   ```

4. If a field contains a decimal point, the user may also append four trailing exclamation points to signify floating-point notation. (If more or fewer than four exclamation points are shown, they will be printed literally in the output.) BASIC determines the need for floating-point notation according to the rules given for the PRINT statement. The four !'s provide for a letter E, a plus or minus sign, and a two-digit exponent. Note that a decimal point may be placed anywhere in the field, but that, on printing, it will follow the first digit. The position of the floating-point notation remains unchanged. For example, the statements

```
>98 PRINTUSING 99, 1/30, 2/30
>99 :VALUES ARE #.##!!!! AND +###.!!!!
```

will generate

```
VALUES ARE 3.33E-02 AND +6.67E-02
>
```

5. Except for the #, period, and ! symbols, characters that follow the colon will be printed exactly as shown, with spacing as provided by blanks in the Image statement.

6. Text strings may be inserted in fields containing decimal points or specifying floating-point notation. In addition, if the field is preceded by an algebraic sign, its position will be preempted by the text string. For example, the statements

```
>11 :THE VALUE IS -#.####
>47 PRINTUSING 11,'TOO BIG'
>
```

will generate

```
THE VALUE IS TOO BIG
>
```

If a text string is larger than its corresponding field, it will be truncated on the right.

In addition to the above rules, printing is subject to the following conventions.

1. If the field to the left of a decimal point is not large enough to contain a numeric value, asterisks are inserted in the printed output as a warning to the programmer. For example, the statements

```
>50 PRINTUSING 75, X, SQR(X), 'SQ. ROOT'
>75 :IF X=#, # IS ITS ########
>
```

will generate, assuming an X value of 25,

```
IF X=*, 5 IS ITS SQ. ROOT
>
```

2. If an Image field is larger than necessary, the printed output will show blanks preceding expression values and following text string values up to the required number of positions in the field.

3. If a PRINTUSING statement specifies more values than there are fields in the complementary Image statement, the Image statement is repeatedly used until all PRINTUSING values are printed. For example, the statements

```
>8 :N=#####
>90 PRINTUSING 8, 1, 4, 90, 81777
```

will generate

```
N=    1
N=    4
N=   90
N=81777
>
```

4. If a PRINTUSING statement specifies fewer values than there are fields in the complementary Image statement, the printout will be terminated at the first unused field of the Image statement. For example, the statements

```
>9 :######## CASES ##.# RESULTS
>103 PRINTUSING 9, 'NO MORE'
```

will generate

```
NO MORE  CASES
>
```

Whenever a PRINTUSING statement is executed, printing starts at the left of a new line. Values are rounded approximately prior to printout. Although the programmer may specify numeric fields greater than 16 characters in length, only 16 significant digits are output (with trailing zeros to fill out the field) for the fractional part of values.

Note that PRINTUSING is not affected by the WIDTH command (see Chapter 4) and will accept up to 132 character positions.

PAGE[t] The PAGE statement can be used to advance the paper to the top of the next page. Figure 1 shows how PAGE and PRINT statements might be used in CP-V to produce a page of tabular data.

Note that before BASIC is called, a PLATEN command is used to set the page length to 16 lines rather than the standard 54 lines per page. A PLATEN ,0 command given prior to calling BASIC would have caused both PAGE statements to be ignored. (For a more detailed discussion of the PLATEN command, see the Xerox CP-V/TS User's Guide, 90 16 92).

[t]CP-V and BPM only.

```
!PLATEN ,16

!BASIC
>100 PAGE
>110 PRINT ,'X', 'X SQUARED', 'X CUBED'
>120 PRINT
>130 X=X+1
>140 PRINT ,X, X↑2, X↑3
>150 IF X<10 THEN 130
>160 PAGE
>RUN
13:32   JUN 08  RUNEBAA...




13:32 06/08/72 356101 24-26      [3]




                X              X SQUARED      X CUBED

                 1              1              1
                 2              4              8
                 3              9              27
                 4              16             64
                 5              25             125
                 6              36             216
                 7              49             343
                 8              64             512
                 9              81             729
                 10             100            1000




13:33 06/08/72 356101 24-26      [4]




     160 HALT
>
```

Figure 1. Use of PAGE Statements in CP-V

DATA INPUT

DATA and READ The data values used in the execution of a program may be contained in a DATA statement. They are called into use at appropriate times by the READ statement. READ and DATA are used in combination with each other.

DATA statements form a chained list of constants that the READ statement accesses from left to right, top to bottom. DATA takes the form

line DATA [constant] [,[constant]]...

Simple constants may be preceded by a plus or minus sign. DATA statements may also contain alphanumeric constants or text strings. An empty field after DATA, as in

```
>1250 DATA
```

or an empty field between commas or after a terminating comma, as in

```
>1260 DATA 1,2,3,,5
>1270 DATA 6,7,8
```

implies a value of zero.

DATA statements may appear anywhere in a BASIC program, and do not have to be consecutive. However, it is good practice to group the DATA statements at the end of the program, thereby making it possible to add as many statements as are needed to contain the data values without disrupting the order of the preceding statements.

READ assigns (in consecutive order) the values in the DATA statement(s) to the variables listed in the READ statement. The form of the READ statement is

[line] READ variable[,variable]...

There is no comma following the final variable in the list.

Example:

```
>555 READ B,C,D
```

If a READ statement requests data after the list of constants in the DATA statement has been exhausted, execution of the program ceases and a message is output to the programmer advising him of the out-of-data condition.

The list of variables following a READ statement may include either of two special entities. A single asterisk means to take an error exit if the current DATA statement list has not been completely read. A double asterisk means to skip any unread elements in the current DATA statement.

Examples:

```
>500 READ X,Y,**
>510 READ A,B,C,*,D,E,F
```

Suppose the program includes these DATA statements:

```
>1000 DATA 1,2,31.5
>1010 DATA 3,4,5,6
>1020 DATA 7,8,9
```

When line 500 is executed, 1 is read into X, 2 is read into Y, and the 31.5 is skipped. When 510 is executed, 3 is read into A, 4 into B, and 5 into C. The single asterisk is encountered with 6 left in statement 1010 and the error message EXTRA INPUT results.

INPUT The INPUT statement requests data from a source that is external to the program, that is, teletype unit, card reader, console, or other input device. (INPUT differs from READ in that when using READ, the DATA statement and its data values are contained within the program itself.) Data may be stored in an external medium for two reasons: either the data is unknown when the program is written but will be supplied when the program is run, or the amount of data is too large for inclusion in the body of the program. The INPUT statement takes the form

[line] INPUT variable[,variable]...

There is no comma following the final variable in the list. When the INPUT statement is executed, data values are read into the computer from the external storage medium and are assigned, one at a time, to the variables designated in the INPUT statement. It should be emphasized that data is stored as it is received, and that the variables are satisfied (that is, associated with the data) in the order in which they are specified. Some sample INPUT statements are shown below.

```
>100 INPUT X
>110 INPUT A,B,Z,Y,R3
>120 INPUT B(1,N), C(N), N
>130 INPUT N, B(1,N), C(N)
```

In the above example, every time statement 100 is executed, the computer will supply a data value to the variable X. Statement 110 will supply data values to A, B, Z, Y, and R3, in that order, from the list of data supplied by the programmer. Statements 120 and 130 will very probably not be equivalent, even though the same variables are specified in both. They will not be equivalent even if the data values are supplied in the same order as the variables were given, unless the value of N is not changed by execution of either of the INPUT statements.

When data input is required, the user is signaled by a "?" character.

The data values that satisfy the variables in INPUT are contained in a list of data separated by commas or blanks. If the list begins with a comma (or in the case of commas with no intervening nonblank characters), the computer understands that a zero value precedes the comma. For example, the computer interprets

```
,5,3 4
```

as meaning the values 0,5,3, and 4. Similarly, if the list ends with a comma, as in

```
1 2 3,
```

the computer will assign the variables in the INPUT statement the data values 1,2,3, and 0. Finally, the list

```
5 ,1 ,,5 1 2
```

will be interpreted as data values 5,1,0,5,1, and 2.

After the entire list of variables in an INPUT statement is satisfied, control passes to the next program statement. If at the time, the current line of input values has not been exhausted, the remaining values will be accessed by the next INPUT statement executed.

The list of variables in an INPUT statement may include the special entities, asterisk and double asterisk, used to act on unused fields in lines entered for input. The double asterisk means skip any unused fields. The single asterisk means take an error exit if unused fields remain in the line of input.

Examples:

```
>200 INPUT A,B,**,C,D
```

means input to A and B, skip anything left in the current input line, and input to C and D from the next line.

```
>210 INPUT A,B,*
```

means input to A and B. Error exit if the input line is not exhausted.

If the input lines shown above are entered in response to statement 100, etc., and N = 2 prior to executing statement 100, the result is as follows:

```
X=0
A=5 B=3 Z=4 Y=1 R3=2
B(1,2)=3 C(2)=0 N=5
N=1 B(1,1)=0 C(1)=5
```

The values 1 and 2 are pending for any subsequent INPUT statement.

Note: In BTM on-line BASIC, with INPUT via the users console, if a portion of an input line is unused at the end of execution, it will be interpreted as new source input and, because of its form, will cause an error message (e.g., BAD FORMAT).

In the batch mode, with input data via the card reader, if there are unused data cards at the end of program execution, a Monitor error message will result and any remaining activities prior to the next job card will be aborted.

LOOPING

FOR and NEXT BASIC provides the programmer with still another method for specifying data values for variables. This method defines a loop using FOR and NEXT statements. A loop is a portion of a program written in such a way that it will execute repeatedly until some test condition is met. A FOR and NEXT loop causes execution of a set of steps for successive values of a variable until a limiting value would be exceeded. Such values are specified by establishing an initial value for a variable together with a limit value, and an increment or decrement that is used to modify the variable each time the loop is executed. When the limit is exceeded, an exit condition built into the loop allows the computer to proceed to the following body of the program. FOR and NEXT loops, therefore, have three main components.

1. An initial value expression for the variable used by the formula.

2. A limit value expression beyond which the variable may not be incremented (or decremented).

3. An optional increment or decrement expression value to be added to (or subtracted from) the value of the variable for each pass through the loop (except the last).

The FOR statement defines loop parameters. It gives the initial value of the variable, the expression for the limit value that the variable may not exceed and that cause the loop to terminate, and (optionally) the increment or decrement expression. If the step increment or decrement is not given in the FOR statement, it is assumed to be +1. The FOR statement takes the form

```
line   FOR   simple variable = expression TO
       expression [STEP expression]
```

The expression preceding TO specifies the initial value of the variable, the expression following TO gives the limiting value, and the expression following STEP gives the increment or decrement. The computer evaluates the initial value expression only once, when the FOR statement is executed. The other two expressions are also evaluated when FOR is executed, but, additionally, are reevaluated every time the NEXT statement is executed. A sample FOR statement is shown below in the discussion of NEXT.

The NEXT statement returns program execution to the beginning of a FOR and NEXT loop after the indexed simple variable has been incremented. NEXT has the form

```
line   NEXT   simple variable
```

Note that the simple variable in the NEXT statement must be specified exactly as it appeared in the FOR statement.

It is possible to branch into or out of a FOR-NEXT loop at any point.

Example:

```
>100 N=5
>110 FOR I=1 TO N
>120 IF I >2 THEN 140
>130 NEXT I
>140 PRINT I
>RUN
14:46     JUL 28   RUNDCAA...
 3

        140 HALT
>
```

In the above example, a branch to line 140 is taken when the value of I reaches 3, rather than waiting for the loop to terminate at I = 5.

The easiest way to understand a FOR and NEXT loop is to follow one through its entire sequence of operations, as in the following statements.

```
>50 FOR X=2 TO 11 STEP 3
>60 PRINT X, 2↑X
>70 NEXT X
```

Statement 50 sets the initial value of X to 2 and specifies that X thereafter will be incremented by 3 each time the loop is performed until X has the limiting value 11.

Statement 60 causes the computer to print out the current value of the variable X and the result of 2^X. Statement 70 causes the computer to return to statement 50, where it picks up the next value of X, that is, +5. The computer then prints 5 and 32 and again goes to NEXT which returns it to FOR. When X attains the limit value of 11, statement 60 will be executed and control will pass to 70. The computer will again try to increment X by 3, but as the upper limit of variable X will have been reached, the computer will "fall through" statement 70 and control will pass to the next statement. At this point, X will have the value 11, the last value that does not exceed the terminal value.

Fractional values may be used in FOR-NEXT loops. When this is done, there is the chance that an expected iteration may not occur because of rounding, as in the following statements:

```
10   FOR I=.1 TO .4 STEP .1
.
.
.
50   NEXT I
```

This loop will be executed only for I = .1, .2, and .3 because the rounded value of I is slightly over .4 on the last try. To get four iterations in this example, use

```
10   FOR I=.1 TO .41 STEP .1
```

Loops may be contained within other loops (nested), but the loops may not "cross". This exclusion is illustrated in Figure 2.

BASIC allows loop nesting to 26 levels, that is, the BASIC program may contain no more than 26 FOR statements whose corresponding NEXT statements have not yet been encountered in compilation.

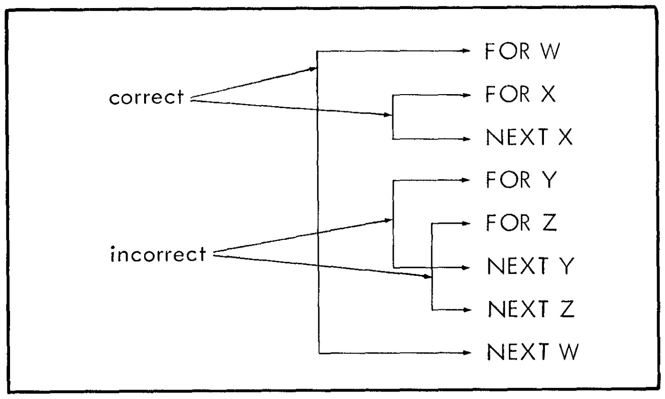


Figure 2. Nested Loops

MISCELLANEOUS STATEMENTS

REM or * The REM (Remark) statement allows the programmer to interject commentary anywhere in the program without affecting its execution. Remarks may be used to identify the complete program, or, more important, the function or purpose of various sections of the program. The Remarks statement takes the form

$$[\text{line}]\left\{\begin{matrix}\text{REM}\\ *\end{matrix}\right\}[\text{commentary}]$$

Notice that a Remarks statement is indicated by an asterisk (*) or the word REM. The commentary portion of the statement may include any characters up to the end of the line If commentary is omitted, REM or * will produce a dummy line in the program.

Examples:

```
>100 REM THIS IS A REMARK
>110 *THIS IS ALSO A REMARK
```

Branching to a Remarks statement is allowed and is recommended when branching to a closed subroutine. Such use of a Remarks statement serves to identify the subroutine. It also allows statements to be inserted at the beginning of the subroutine, if unused line numbers exist between the Remarks statement and the first executable statement of the subroutine.

PAUSE, STOP, or END PAUSE, STOP, or END can be used to halt program execution at any point. The line number of the halt is printed when program termination occurs. Chapter 4 outlines methods for resuming operation of a halted program (see "PROCEED").

PAUSE, STOP, and END have the form

[line] PAUSE

[line] STOP

[line] END

Any number of these may be used in a program, or none at all. If none is used, the program will normally halt after the highest numbered line has been executed. If a branch into an infinite loop occurs, as shown in the example below, the BREAK key can be used to halt execution.

```
>100 INPUT A
>110 PRINT A
>120 GOTO 100
>
```

3. ADVANCED FEATURES OF BASIC

For simplicity, explanations in previous chapters have covered only the essential features of BASIC program elements. This chapter contains additional information on these elements and explains advanced features of BASIC.

MULTIPLE STATEMENTS PER SOURCE LINE

BASIC programs are organized as numbered statements. Many sequences of statements are always executed in sequence. That is, statements in the middle of the sequence are never reached by a GOTO, GOSUB, or RETURN. Sequences of statements of this type may be written in a packed form (more than one per line).

The form is

[line] statement\statement

or

[line] statement & statement

Examples:

```
>10 INPUT :1,A \ PRINT A
>20 A=0 \*THIS IS A REMARK
>30 B=B+1 & GOTO 50
```

Restrictions:

DATA and Image statements must be numbered and be the only statement on the line.

The statements END, GOSUB, GOTO (without ON), PAUSE, REM, RETURN, and STOP may occur only as the last element in a line.

The multistatement capability is advantageous not only in saving memory and file space for the user, but also in allowing grouping of related short statements and insertion of remarks on the same line as a statement (as in line 20 above).

OTHER ELEMENTS OF A BASIC PROGRAM

The additional program elements presented below give the user greater flexibility in using the statements explained in Chapter 2, and also augment the capabilities of the new statements described in this chapter.

SUBSCRIPTED VARIABLES

In addition to simple variables, BASIC also provides for subscripted variables. A subscripted variable denotes an element of an array, that is, a list or table of data. The individual values within the array are called array elements. We refer to an array element by specifying the name of the array (always a single letter) and the position of the element in the array. For example, the fourth element in the array named L is denoted by L(4). The value inside the parentheses is called the subscript, and is represented by an expression that can be reduced by the computer to a single integer value. (Subscript expressions are evaluated to integer value after adding 2^{-12}.) Subscripts range from 1 through the maximum allowed dimensioned value.

Arrays can have either one or two dimensions. A one-dimension array is called a vector and is characterized by a single subscript. The subscript denotes the position of the desired array element in the list of data. Sample vector array elements are A(1) and B(J + 3).

When an array has two dimensions, it is called a matrix. Data in a matrix is thought of as being arranged in rows and columns. Each element in a matrix is identified by two subscripts separated from each other by a comma. The first subscript specifies the row number and the second specifies the column number. For example, C(K, L) and D(M+2, N+3) denote matrix array elements.

As a further example of matrix notation, consider the following table, which lists expenses for a four-day car trip.

	Column	1	2	3	4
Row	Date / Item	June 5	June 6	June 7	June 8
1	Gas, oil	21.29	20.84	19.42	6.08
2	Tolls	1.32	.86	.40	.07
3	Food	11.18	12.83	14.39	5.06
4	Lodging	10.05	12.78	10.35	.00
5	Misc.	1.35	.44	.90	.10

If we consider the table to be a matrix called E, the amount ($10.05) spent for lodging on June 5 would be represented by E(4, 1), and the amount ($5.06) spent for food on June 8 would be represented by E(3, 4).

DIMENSIONING

A dimension is the largest value that a subscript may attain for a given subscripted variable (array). This limit tells the computer how many storage units of the computer's memory to allocate for the array. Dimensions are specified explicitly in the DIM statement, but the user may make array references without corresponding DIM statements. In such cases, implicit dimensions are used. Implicit dimensions are: 10 storage units for a vector and 100 storage units for a matrix (that is, a 10 by 10 matrix). If the program uses MAT statements (explained later), the dimensions of all arrays referred to in these statements must be explicitly defined in DIM statements.

DIM There are three reasons for explicitly specifying the dimensions of an array.

1. The user may wish to allocate more space for his array than allowed by implicit dimensions. Thus, DIM A(18) would reserve 18 storage units for the vector A.

2. The user may wish to restrict the reserved storage space for each array to its exact dimensions, thereby conserving space. For example, DIM B(3,4) reserves 12 storage units for matrix B, thereby leaving for other use the remaining 88 units that would have been allocated by implicit dimensions.

3. The user may wish to use a given array in a MAT statement.

The DIM statement takes the form

$$[\text{line}]\text{DIM}\left\{\begin{matrix}\text{letter\$}^{\dagger}\\ \text{letter}\end{matrix}\right\}(\text{dimx}[,\text{dimx}])\left[,\left\{\begin{matrix}\text{letter\$}^{\dagger}\\ \text{letter}\end{matrix}\right\}(\text{dimx}[,\text{dimx}])\right]\ldots$$

where

letter or letter\$ is the name of the array being dimensioned.

dimx is a dimension expression that denotes the maximum number of row or column elements in the array.

Dimension expressions may not contain user-defined functions, array references, letter digit variables, or letters that have not been set (see "SET" in Chapter 4), and are evaluated during compilation (not during execution) by truncating to an integer value after adding 2^{-12}. If dimensions for more than one array are specified in a DIM statement, they are separated by commas. A given array may be dimensioned only once in a BASIC program via a DIM statement. DIM statements may appear anywhere in a BASIC program. A sample DIM statement is given below.

```
>10 DIM M(3,3), V(128)
```

A string scalar may not appear in a dimension statement. To avoid conflict, the same letter may not be used for both a string scalar and a string array or numeric array.

[†]CP-V only.

VECTORS

Numeric Vectors. A numeric vector is a one-dimensional array containing numeric or aconst data elements. The name of a numeric vector consists of a single alphabetic character. An element is referenced by a subscript expression denoting the relative position of the desired element (see "Subscripted Variables", above).

```
>100 A(1)=3.14, B(A(1))='LARGER'
>110 A(2)=SIN(A(1)-1), B(2)=1
>115 IF A(1)<A(B(A(1)-B(2))+1) THEN 150
>120 PRINT B(A(B(2)))
```

String Vectors.[†] A string vector is a one-dimensional array containing text string elements. The name of a string vector consists of an alphabetic character followed by a dollar sign. An element is referenced by a subscript expression denoting the relative position of the desired element. The subscript expression may be followed by a substring expression specifying the beginning and length of the desired substring (see "Character String Manipulation", below).

```
>100 DIM A$(3)
>110 A$(1)='ABCDEFGH'
>120 A$(2)=A$(1:2)
>130 A$(3)=A$(2:3,4)
```

In the above example, vector element A\$(2) is assigned the string BCDEFGH and A\$(3) is assigned DEFG. Note that all string arrays must be dimensioned via a DIM statement.

MATRIXES

Numeric Matrixes. A numeric matrix is a two-dimensional array containing numeric or aconst data elements. The name of a numeric matrix consists of a single alphabetic character. An element is referenced by a pair of subscript expressions, separated by a comma, denoting the row and column of the desired element (see "Subscripted Variables", above):

```
>100 A(1,1)=1, A(1,2)=2
>110 A(1,3)='THREE', A(2,7)=5
>115 IF A(1,1)>A(1,2) THEN 150
>120 PRINT A(1,A(2,7)-A(1,2))
```

String Matrixes.[†] A string matrix is a two-dimensional array containing text string data elements. The name of a string matrix consists of an alphabetic character followed by a dollar sign. An element is referenced by a pair of subscript expressions, separated by a comma, denoting the row and column of the desired element. The subscript pair may be followed by a substring expression specifying the beginning and length of the desired substring (see "Character String Manipulation", below):

```
>100 DIM A$(2,2)
>110 A$(1,1)='ZEITGEIST'
>120 A$(1,2)=A$(1,1:3,6)
>130 A$(2,1:1,2)=A$(1,2:3)
```

In the above example, A$(1,2) is assigned the string ITGEIS and A$(2,1) is assigned GE. Note that all string arrays must be dimensioned via a DIM statement.

CHARACTER STRING MANIPULATION

In Batch/BTM BASIC, strings are limited to alphanumeric constants up to six characters long and to text strings in PRINT statements. CP-V BASIC permits strings up to 132 characters long and provides capability for

1. Referencing string variables.
2. Using string expressions.
3. Assigning a character string variable.
4. Assigning length or numeric value of a string variable to a simple or subscripted variable.
5. Converting a numeric value to string format.
6. Concatenating strings.
7. Comparing strings.
8. Using strings in input/output statements.
9. Generating alphanumeric constants from strings for file identification.
10. Establishing maximum string length and storage requirements (see "SET" in Chapter 4).

REFERENCING STRING VARIABLES

Strings are identified by a letter and dollar sign followed by a further identification of the type of string specified: string scalar, string array, string array element, or substring. Examples of each of these are given at the end of this discussion. Strings may also be combined in expressions for the purpose of string concatenation.

String scalars have the form

letter$ or $letter[†]

A string scalar may not appear in a dimension statement. To avoid conflict, the same letter may not be used for both a string scalar and a string array or numeric array.

[†]Allowed for compatibility with A00 BASIC.

String array elements are subscripted variables. They have the form

letter$ (expr[,expr])

where the optional expression denotes a matrix element. A string with only one expression is a vector element.

String arrays may be explicitly dimensioned. The form for dimensioning a string array is

DIM letter$ (dim[,dimx])

Substrings are marked by a colon preceding an expression.

letter$ (:expr 1[,expr 2])

where

expr 1 indicates the position of the first character of the substring.

expr 2 indicates the length of the substring in number of characters. If expression 2 is omitted, the substring includes all characters from the indexed character to the end.

If a string is an element of a vector or a matrix, then the form of the substring is

letter$ (expr 1[,expr 2] : [expr 3][,expr 4])

where

expr 1 and expr 2 are the indexes of a string array element.

expr 3 is the index, or string position, or the first character of the substring.

expr 4 is the length, or number of characters in the substring. Again, if expression 4 is omitted the string consists of all characters from the indexed character to the end of the string.

String examples are

`P$`	String scalar.
`H$(1)`	String vector element.
`B$(2,3)`	String matrix element.
`A$(:4)`	Substring consists of all characters from the fourth to the last character of A$.
`A$(:4,1)`	Substring consists of fourth character of A$.
`B$(2,3:5,2)`	Substring consists of the fifth and sixth characters of string matrix element B$(2,3).

STRING EXPRESSIONS

String expressions may be used in CP-V BASIC as arguments for string functions; PUT, PRINT, and PRINTUSING statements; string concatenations; string comparisons; and as file identifiers in OPEN and CHAIN statements. They must be explicitly stated in the PUT, PRINT, and PRINTUSING statements, but may be in either implicit or explicit format in all other cases.

The implicit string expression (strexp) has the form

$$\left\{\begin{array}{l}\text{string}\\ \text{tstring}\\ \text{var}'\\ \text{STR(expr[,rstring])}\end{array}\right\}\left[+\left\{\begin{array}{l}\text{string}\\ \text{tstring}\\ \text{var}'\\ \text{STR(expr[,rstring])}\end{array}\right\}\ldots\right]$$

where var' is a variable containing an alphanumeric constant.

Implicit string expressions are always to the right of the relation operator in string assignment statements and comparisons.

Explicit string expressions (xstrexp) are required to avoid ambiguity on whether or not string processing is called for. The form of this expression is

$$\left\{\begin{array}{l}\text{\$(strexp)}\\ \text{string + strexp}\end{array}\right\}$$

where the dollar sign resolves the ambiguity that arises if the first character is a letter character (as in STR or in a variable). An implicit expression may be used only in a statement where the syntax is unambiguous in indicating string processing.

Examples:

`LET Z$ = B(2,3) + A$(:5)`	implicit string expression
`PRINT $(B(2,3) + A$(:5))`	explicit string expression
`PRINT B(2,3) + A$(:5)`	ILLEGAL ! ambiguous syntax
`PRINT "ABC" + A$`	ILLEGAL

ASSIGNING CHARACTER STRINGS TO STRING VARIABLES

Simple variables provide storage for just one doubleword; therefore, a simple variable is limited to representing an alphanumeric constant (maximum of six characters).

In CP-V BASIC, character strings more than six characters long must be assigned to string variables (letter$). Strings up to six characters are considered alphanumeric constants and may be assigned to simple or subscripted variables.

STRING LENGTH AND VALUE ASSIGNMENTS

For these assignments, CP-V BASIC provides two intrinsic functions: LEN (for length) and VAL (for value). LEN and VAL may only be used in assignment statements. The assignments are made to simple or subscripted (not string) variables and have the form

[line] LET var[,var]... =sfunct(strexp)

where sfunct is LEN or VAL. Both assignments can be made in one statement, separated by a comma as in the example

```
>25 K1=LEN(W$(2,3)), K2=VAL(W$(2,3))
```

in which the length of the matrix string element WS(2, 3) is assigned to the simple variable K1, and its numerical value to K2. The arguments for both functions must be string expressions. If the character string specified for VAL does not represent a correctly formatted decimal constant, an error message is generated and execution terminates.

CONVERSION TO A STRING

The output conversion routine automatically converts an expression to string format, but in manipulating text it may be desirable to have the same conversion performed internally, for example to store an evaluated expression in a file or embedded as a substring within a text string. The string-conversion routine is available in CP-V BASIC for this purpose. It has the form

[line] LET string = STR (expr[,rstring])

where STR is the string-conversion function.

The replaceable-string (rstring) argument is optionally used to indicate the image of the desired format. If the rstring option is not used, format is that for print output conversion.

Like the output conversion, string-conversion is governed by the setting of the precision flag. The string will have a leading blank if it is nonnegative, but will not contain trailing blanks. The minimum length for string is two bytes; maximum length is 22 bytes for long precision and 12 bytes for short precision.

Examples of STR (conversion-to-string) statements are

```
>10 A$=STR(3.5,#.#)
>20 H$(1:9)=STR(SQR(X))
>30 LET W$(2,3)=STR(A1+B1*COS(X))
```

STRING ASSIGNMENT AND CONCATENATION

Another string, an alphanumeric constant, a string converted expression (see above), or a concatenation of any or all of these may be assigned to a string. The form of the string-assignment and concatenation statement is

$$\text{[line] [LET]string} = \left\{\begin{array}{l}\text{strexp}\\ \text{xstrexp}\end{array}\right\}\left[+\left\{\begin{array}{l}\text{strexp}\\ \text{xstrexp}\end{array}\right\}\right]\ldots$$

where strexp is an implicit string expression and xstrexp an explicit string expression (see "String Expressions", above).

Examples:

```
>100 DIM A$(2)
>110 A$(1)='ONE', A$(2)='TWO'
>120 B$=A$(1)+' AND '+A$(2)
>130 B$=B$+'.'
```

The left string is given a value and a length consistent with the items to the right of the equals sign. If the right contains only one term, the statement performs string assignment. If the right side contains two or more terms, concatenation occurs in the order given. If the maximum string length is exceeded, the string is truncated.

If assignment is to a substring whose current length is less than n-1, where n is the first character of the target substring, then the gap to character n-1 is filled with blanks. If target-substring length is specified and the number of characters transferred is less than this specified length, then the gap from the last character transferred to the specified length is also filled with blanks. Characters in excess of specified length are not transferred.

STRING COMPARISON

Strings are compared for identity or "magnitude" in IF... THEN or GOTO statements. The form of the statement is

[line] IF string oper $\left\{\begin{matrix}\text{strexp}\\ \text{xstrexp}\end{matrix}\right\}$ $\left\{\begin{matrix}\text{THEN}\\ \text{GOTO}\end{matrix}\right\}$ line

where oper is a condition operator (strexp and xstrexp were explained under "String Expressions"). Examples are

```
>10 IF W$(:1)='1W' GOTO 99
>20 IF W$<>STR(X1*Y1+3) THEN 40
>30 IF R$>STR(0) GOTO 85
>40 IF A$<"DOG"+B$(1:9) THEN 120
```

Strings are compared from left to right as character pairs. In comparing characters, the EBCDIC collating sequence is followed (see Appendix H). A blank is the lowest character, followed by nonalphanumeric characters. Alphabetic letters are next, in the order A B C ... YZ. Digits are the highest elements in the collating sequence. If one string is shorter than another, the shorter string is "extended" with blanks for comparison. Note that a string may be compared to an aconst:

```
IF A$=Z THEN 100
```

The string comparison capability of CP-V BASIC makes it possible to sort BCD files alphabetically:

```
>100 INPUT=$ & ENDFILE :1, 140
>110 R=0 & OPEN 'FILE' TO :1,INPUT UPDATE
>115 INPUT :1,C$ \*INITIALIZE KEY (1)TO 1
>120 INPUT :1;KEY(1),A$,B$
>125 IF A$>B$ THEN 130 & GOTO 120
>130 ;:1,A$ & ;:1;KEY(1)-1,B$
>135 R=1 & GOTO 120
>140 IF R=1 THEN 110
>
```

The example shown above reads a file named FILE and sorts the records into ascending alphabetic order. Note that this program would have to be modified considerably to sort large files quickly, and will not work unless all records have keys that are consecutive integers (see "Keyed and Sequential Access", below).

STRING INPUT/OUTPUT

Explicit string expressions and text strings may be used in input/output statements in the form used for expressions and alphanumeric constants in Batch/BTM BASIC, subject to the general rules governing strings. That is, a run-time error results if a text string (more than six characters) is provided as input to a nonstring variable; or nonstring input (neither a text string nor an alphanumeric constant) is provided to a string. The statements used are INPUT, PUT, READ, GET, DATA, PRINT, MAT PUT, MAT GET, MAT READ, PRINTUSING, and MAT INPUT.

Examples:

```
>140 MAT GET :3, A$(2,3), B$
>500 DATA 'ONCE UPON', "A TIME"
>885 PUT V$, W$(1), X$(1,A)+'?'
>910 READ H$(1), H$(2), H$(3)
>700 GET :3;K,W$(A1,A2), X$
>750 PRINT $("IT'S "+STR(A)+B$)
>760 PRINT USING 100, H$(1), C$
>800 PRINT :4,A$(1,1),A$(1,2)
>400 MAT PUT A$, B$, C$
>410 MAT GET A$(2,3), V$
>420 MAT READ F$(4), G$
>430 MAT INPUT W$, X$
>440 MAT GET A$, B
>445 PRINT :1,B$(2,3:4,5);A↑2
```

Line 440 requires that the data file have the correct number of string array elements to fill A$, immediately followed by numeric data to fill numeric array B.

The examples above show cases of string input/output only. The forms are similar to those described earlier for nonstring input/output. In CP-V BASIC, statements may mix (with appropriate caution) string and nonstring items in the same statement, as shown in Appendix A.

STRING INPUT MODE CONTROL

Normally, when a string is reached in the list for an INPUT statement, the next data field in the record is accessed. Blanks and commas are treated as field separators unless they occur within quoted fields. An alternate form is provided in which an entire input line, or record, is treated as a single field.

The form for switching string INPUT mode is

[line] INPUT = $\left\{\begin{matrix}\text{\$}\\ \text{any other character}\end{matrix}\right\}$

INPUT = $ switches to full record input mode. Each input referenced to a string accesses a full record and treats it as a single string (as though it were enclosed in quotes). If a record has been partially input (for numeric assignment) and a reference to a string follows, the remainder of the record is treated as a single field.

INPUT = X (any character but $) switches back to normal input mode, which is the default.

String input mode is changed only by these explicit statements and remains as set through successive operations within BASIC until explicitly reset.

Strings including characters whose EBCDIC value is less than 64 (blank) can be input, but trailing characters of 64 or less are ignored and a null (EBCDIC zero) is interpreted as an end-of-string. Caution is advised in using control characters (see Appendix H) in input strings.

GENERATION OF ACONSTS FROM STRINGS

A string or string expression may be assigned to a simple or subscripted variable, but only six characters will be transferred and the rest truncated. If the string contains fewer than six characters, trailing nulls are generated to satisfy the aconst format.

Examples:

```
>A1=P$
>A2=B$(:4)
>A3=$('NO.  '+C$(A4))
```

This provides an indirect means to assign strings as external names or file identifiers by first assigning strings to simple variables.

STRING EXPRESSIONS AS FILE IDENTIFIERS

In CP-V BASIC, string expressions may be used to designate the name, password, and account for file identification in OPEN and CHAIN statements. The string expressions must not result in text strings exceeding 11 characters for name or 8 characters for account and password.

Examples:

```
>120 OPEN 'FILE'+A$(:I,1) TO :1,INPUT
>340 CHAIN B$(N):'ABC';'SECRET'
```

In line 120 above, if A$ = '1234567' and I = 3 then 'FILE3' is opened.

USER-DEFINED FUNCTIONS

DEF If the programmer wants to make use of a function that is not included in the set of BASIC intrinsic functions, or if he intends to make repeated use of an involved expression, he may define the function in a DEF statement and make reference to it according to a name he designates. The form of the DEF statement is

line DEF FN letter(simple variable[,simple variable]...) = expression

where

letter provides a unique name for the function.

simple variable is a dummy argument appearing in parentheses to the left of the equals sign. These only serve to identify which of the simple variables in the expression to the right of the equal sign are arguments. There must be at least one such argument, although it is not necessary that any or all of the arguments appear in the expression. Each time the function is evaluated, current argument values will be substituted for these terms in the expression. There is no comma following the final simple variable in the list.

The following examples illustrate typical DEF formats:

```
>65 DEF FNA(X)=X+B*X
>100 DEF FNB(X)=X*SIN(FNA(X+C))
>120 DEF FNX(X0,X1,X2)=X0*X1*X2/K
>550 X=FNX(1,2.3)+FNB(Y+3.14)
```

Line 550 is an example of how the functions defined in lines 100 and 120 might be used later in the program. The variable X to the left of the equals sign is a different entity from the dummy variables X in the DEF statements.

DEF statements may appear anywhere in the BASIC program, including those cases in which the function is referenced prior to its definition.

BASIC checks DEF statements for identical simple variables in the list of dummy arguments, undefined functions, multidefined functions, and consistency between the number of arguments supplied by the programmer when the function is called (referred to) and the number of arguments in the DEF statement. However, it is the responsibility of the programmer to avoid circular definitions in and among the DEF statements. Improper uses of DEF are shown below.

Case 1. Circular definition within statement:

```
>1200 DEF FNA(X)=X+FNA(X)
```

Case 2. Circular definition among statements:

```
>1400 DEF FNA(X)=X+FNB(X)
>1450 DEF FNB(X)=X*FNC(X)
>1500 DEF FNC(X)=FNA(X)/X
```

REREADING DATA

RESTORE The RESTORE statement alters the normal sequence of DATA statement accession. DATA statements are normally accessed as the preceding DATA statement is exhausted. For example, of the following set of DATA statements,

```
>100 DATA 1,2,3,4
>110 DATA 5,6,7,8
>120 DATA 9,10,11,12
```

statement 110 will be accessed only after data value 4 in statement 100 has been assigned to a variable, and statement 120 will be accessed after data value 8 in the preceding statement is assigned. RESTORE allows the programmer to alter this sequence by directing the computer (via a line number) to a specified DATA statement from which data accession will proceed in the normal manner.

The RESTORE statement is frequently used for accessing data that will be used several times in the program, and eliminates the need for writing duplicate DATA statements when the same data is to be accessed more than once. The form of the RESTORE statement is

[line] RESTORE [line]

where the second "line" must be the line number of a valid DATA statement in the program. Some sample RESTORE statements are given below.

```
>740 RESTORE 125
>900 RESTORE
```

If the line number is omitted in the RESTORE statement (as in line 900 above), the computer will return to the first DATA statement in the program.

BRANCHING TO A SUBROUTINE

GOSUB and RETURN The GOSUB and RETURN statements provide subroutine capability in BASIC. A subroutine is a section of the main program that completes a specific task. GOSUB, in the main body of the program, directs the computer (via a line number) to the first statement of the subroutine. After the subroutine has been executed, RETURN directs the computer to the statement following GOSUB, where the main program continues. The form of GOSUB and RETURN are

[line] GOSUB line

[line] RETURN

where RETURN is the last executed statement of the subroutine.

Some sample GOSUB and RETURN statements are shown below.

```
>10 GOSUB 500
 .
 .
 .
>525 RETURN
```

The RETURN statement does not contain the line number of the statement following GOSUB. BASIC remembers its place in the program.

An attempt to execute a RETURN statement before a GOSUB statement is executed causes output of an appropriate error message. Execution of too many GOSUBs before a RETURN also causes an error message to be printed. The program may execute up to 20 GOSUB statements before a RETURN is needed.

CHARACTER CONVERSION

CHANGE The CHANGE command (CP-V only) can be used to convert string characters to equivalent EBCDIC values and vice versa. To convert a string to EBCDIC, the command has the following form:

$$[\text{line}]\ \text{CHANGE} \begin{Bmatrix}\text{string}\\ \text{xstrexp}\end{Bmatrix} \text{TO letter}$$

Examples:

```
>10 CHANGE A$ TO B
>50 CHANGE $('246'+C$) TO D
```

The string characters are converted to EBCDIC values stored in the vector specified by the letter. The letter must represent a vector dimensioned by a DIM statement. The current dimension of the vector is set to the number of string characters converted.

Assuming that A$ = 'A1' when line 10 above is executed, the decimal equivalent of 'A' (i.e., 193) is stored in B(1) and the equivalent of '1' (i.e., 241) is stored in B(2). Decimal equivalents of all EBCDIC characters are listed in Appendix H.

To convert a vector to a string, the following form is used:

[line] CHANGE letter TO string

The elements of the specified vector are converted to characters and placed in the specified string or substring. The current dimension of the vector is used. If a value has a fractional part, it is truncated.

Example:

```
>15 CHANGE X TO Y$
```

Assuming X has the elements 193, 241, 90, and 7, the characters 'A', '1', '!', and 'bell' will be stored in Y$.

FILE MANIPULATION

Files are made up of records, each of which may contain one or more data elements. A file may be organized as a consecutive sequence of records or as a set of records arranged according to sort keys. The data in files may exist in a "print" form (BCD) or in the form used within the computer (binary). BASIC allows operations on BCD and binary files with sequential or keyed access. Files are also used to store and fetch BASIC programs. The BASIC statements OPEN, CLOSE, GET, PUT, ENDFILE, and special forms of PRINT and INPUT provide file manipulating capability.

FILE NOMENCLATURE

Files may have names of up to 11 characters. In BTM/BPM BASIC only two forms are allowed. Names may be enclosed in quotes (single or double) or may be aconsts (limited to six characters) stored in simple variables. In CP-V BASIC a name may also be any string expression (see Chapter 5) of up to 11 characters. It is advisable to restrict the characters to letters and digits if the files are to be accessed by processors other than BASIC.

The name may be optionally followed by a password of up to eight characters and/or an account identifier of up to eight characters. An account should not be specified for output operations, since output will not be permitted on other users' accounts.

The general form for file identification in an OPEN statement is

name [;password][:account]

Examples:

```
"DATAFILE"
A1;'SECRET':'12345678'
```

In the second example, the simple variable A1 must contain a name as an aconst.

Passwords are used for file security, and the account is used to input data created in other users' accounts. A password is preceded by a semicolon and an account number is preceded by a colon. If both account and password are specified, the order of their appearance is optional.

The term 'fileid' will be used for name [;password][:account] in describing forms of the OPEN and CHAIN statements.

I/O STREAM NUMBERS

Files are "opened" to specified input/output "streams". An I/O stream is a means of transferring data between a BASIC program and the computer's file system. In some ways an I/O stream is like a bank teller's window that may be "closed" when not in use and, when "open", may be assigned a specific function such as paying or receiving. BTM on-line BASIC currently permits three I/O streams. BPM (Batch) and CP-V BASIC permit four streams. Only one file can be opened on a given stream number at one time, but the same stream number may be used to open another file later, closing the currently open file. The stream number is specified in the OPEN statement and may be any expression which evaluates to a legitimate stream number. Fractional values are truncated to integers, and expressions which do not result in integer values 1 to 4 should generally be avoided.

KEYED AND SEQUENTIAL ACCESS

BASIC allows file access in either sequential or keyed form. All files created by BASIC are actually keyed but they may be sequentially written and read without explicit references to keys. Sequential files created without keys may be read sequentially.

A "key" is a means of selecting a specific record in a file. All records in BASIC files are arranged in ascending order according to the numeric value of the record keys. If a file is created without explicit keys, the first record written is given a key of 1, the second record will have a key of 2, and so on. A key is not part of the contents of a record but is used to identify the record, in much the same way that a license plate identifies an automobile.

The keys used in BASIC are numbers in the range 0.001 to 9999.999. Sequential files are created with keys 1.000, 2.000, ... (these keys are compatible with the keys, or "sequence numbers", used in the Xerox Edit processor).

The key value for a given I/O operation can be set by explicit reference to the key, using any arithmetic expression. The value is multiplied by 1000, rounded to an integer, and the result used as a three-byte binary key. If a subsequent operation on the same file does not reference an explicit key, the key value is incremented by one for each record accessed.

If an output statement with an explicit key creates more than one record, the subsequent records have keys incremented by one per record. If an input statement with an explicit key requires more than one record of data, records are read sequentially starting at the record with the referenced key.

The user can determine the largest key value in any file via a few BASIC statements. For example, suppose the user wants to extend a file named FILE1 and wants to know the last key value of the file before doing so. He can use the following sequence of statements to do this:

```
>OPEN 'FILE1' TO :1,INPUT
>INPUT :1;9999,A

               KEY NOT FOUND
>;KEY(1)
 4
>CLOSE :1
>
```

The example indicates that a 4 is the largest key value in the file named FILE1 (see the third-from-last line). To find the largest key value in any other file, simply substitute the appropriate file name in the first line of the above example.

UNKEYED I/O IN THE UPDATE MODE

When a file is opened in the update mode (see "Binary File Update" and "BCD File Update" below) a BASIC program may input data from the file or may replace existing data records with new information. The most straightforward way of updating an old record is to specify the key of the record, in a PRINT or PUT statement (PRINT for BCD files and PUT for binary files). It is possible to update records without specifying a key, however file positioning is not separately maintained for input and output operations. For example, if a PUT is followed by a GET and no key is specified in the GET statement, any data left over from a previous GET is input first (i.e., residual data from the last record read). Then the record accessed is the next one, in ascending key sequence, after the last record PUT. An unkeyed PUT following a GET replaces the last record accessed.

An unkeyed PRINT following an INPUT replaces the last record accessed. An unkeyed INPUT following a PRINT accesses the next record after exhausting all data in a previously INPUT record.

In general, it is advisable to specify keys in all UPDATE operations.

OPEN The OPEN statement performs the following file management functions:

1. Designates that the named file is to be opened for BCD or binary input, output, or update.

2. Assigns the file to an I/O stream number.

3. If a file is to be opened for output (PUT or PRINT is specified) and an old file of the same name exists, the old file is deleted when the new file is opened. If a file is to be opened for input and the same file is currently open for output, the file is closed for output before being opened for input. Re-opening a file for input repositions the file to the first record. A file may be open for input on more than one I/O stream at the same time.

4. Indicates whether an existing file may be written over if a file of the same name is to be open for output.

5. Positions the opened file at its starting point. (A file opened for output is initialized as an empty file.)

6. Declares a file as a TFILE if the OPEN statement so indicates. TFILES are released at the end of a terminal session or, in batch operations, at the end of the JOB. The TFILE directive is ignored if the file has a password.

BINARY INPUT

The OPEN statement for binary input has the form

[line]OPEN fileid[,]TO :stream, GET[[,]TFILE]

Example:

```
>120 OPEN 'DATA' TO :3, GET
```

This opens the file on I/O stream 3 and does not declare the file temporary.

DEFAULT FORM FOR BINARY INPUT

An abbreviated form may be used for binary input.

[line]OPEN fileid[,]I[any characters]

This is equivalent to

[line]OPEN fileid TO :1, GET

BCD INPUT

The OPEN statement for BCD input has the form

[line]OPEN fileid[,]TO :stream, INPUT[[,] TFILE]

Example:

```
>125 OPEN 'IT' TO :1, INPUT
```

This opens the file named IT to I/O stream 1 for BCD input.

BINARY OUTPUT

The OPEN statement for binary output has the form

[line]OPEN fileid[,]TO :stream, PUT, $\left\{\begin{matrix}\text{ON}\\\text{OVER}\end{matrix}\right\}$[[,]TFILE]

Example:

```
>130 OPEN 'OUTF' TO :A(I),PUT,OVER,TFILE
```

If A(I) = 4, this opens "OUTF" to stream 4 for binary output (PUT). OVER indicates that an old file named OUTF is to be written over if present. TFILE indicates this is a temporary file, to be released at end of job.

DEFAULT FORM FOR BINARY OUTPUT

An abbreviated form may be used for binary output.

[line]OPEN fileid[,] O [any characters]

This is equivalent to

[line]OPEN fileid TO :2,PUT,OVER,TFILE

BCD OUTPUT

The OPEN statement for BCD output has the form

$$[\text{line}]\text{OPEN fileid}[,]\text{TO :stream, PRINT}[,]\begin{Bmatrix}\text{ON}\\ \text{OVER}\end{Bmatrix}[[,]\text{TFILE}]$$

Example:

```
>140 OPEN 'BCDOUTFILE' TO :4, PRINT ON
```

The 'ON' directive means if an old file exists with name BCDOUTFILE, it is not to be overwritten.

BINARY FILE UPDATE

To update a binary file, use the form

[line]OPEN fileid[,]TO :stream, GET[,]UPDATE [[,]TFILE]

This opens an existing binary file in the update mode, allowing input (GET) and output (PUT) on the file.

Example:

```
>145 OPEN 'BIN' TO :2, GET UPDATE
```

This opens file BIN to I/O stream 2 for input or output.

BCD FILE UPDATE

To update a BCD file, use the form

[line]OPEN fileid[,]TO :stream, INPUT[,]UPDATE [[,]TFILE]

This opens an existing BCD file in the update mode, allowing input (INPUT) and output (PRINT) on the file.

Example:

```
>150 OPEN 'SESAME' TO :3, INPUT UPDATE
```

This opens file SESAME to I/O stream 3 for input or output.

ENDFILE The ENDFILE statement allows the user to branch to a designated line number in his program when an out-of-data condition occurs or a specified key is not found. The form of the ENDFILE statement is

$$[\text{line}]\text{ENDFILE :stream,}\begin{Bmatrix}\text{E}\\ \text{line number}\end{Bmatrix}$$

The "stream" may be any expression. If the expression evaluates to a legitimate stream number, endfile control will be applied to any GET or INPUT via that stream.

If the expression evaluates to zero, endfile control applies for READ and for INPUT from the card reader in the off-line (batch) mode. However, I/O stream zero can be specified to provide endfile branching when an end-of-file (i.e., an ⓒF) is INPUT from the user terminal.

The "stream" expression is followed by a "line number" (not an expression) or the letter "E". E indicates reset to normal error exit. A "line number" indicates the location (in the user's program) to transfer to on the out-of-data condition.

Example:

```
>100 OPEN "FILE" TO :1, INPUT
>110 ENDFILE :1, 150
>120 INPUT :1, A$
>130 PRINT A$
>140 GOTO 120
>150 CLOSE :1
```

Note that modification and recompilation of a program that has been interrupted by use of the BREAK key will reset endfile control to an error exit. To restore endfile control after a program interruption, type the desired ENDFILE statement (or statements) as a direct statement just before resuming program execution unless the desired ENDFILE statement will be re-executed within the program.

CLOSE The CLOSE statement closes the file on the indicated I/O stream.

[line]CLOSE :stream

This closes the open file, if any, on the indicated stream number.

Example

```
>200 CLOSE :N
```

If N = 3, stream 3 is closed.

The following forms may also be used.

[line]CLOSE I[any characters](equivalent to CLOSE :1)
[line]CLOSE O[any characters](equivalent to CLOSE :2)

Files may also be implicitly closed by an OPEN statement and are closed on leaving BASIC. Files may be left open through execution of several programs.

GET The GET statement retrieves binary data from files created by PUT statements. Data is assigned to specified variables as it is received from the file via the indicated I/O stream. Access may be sequential or keyed. GET has the form

[line]GET[:stream[;key],] variable[,variable]...

A default form can be used

GET variable...

This is equivalent to

GET :1, variable...

The variables may be simple or subscripted. There is no comma following the final variable in the list. As in READ and INPUT, the variable list may include either of the special entities * or **. The single asterisk causes an error exit if the current record has not been exhausted of data. The double asterisk causes any unused data in the current record to be discarded.

Attempts to GET on an I/O stream which is not open for binary input or to use an illegal I/O stream number will terminate the run with an appropriate error message.

The specification of a nonexistent key or an attempt to read beyond end-of-file gives an OUT OF DATA error exit and message. This exit may be modified by the user via ENDFILE (see above).

A GET statement may require reading more than one record to satisfy the variable list. If the GET statement has the key value n, the records read are accessed sequentially starting with n.

If a keyed GET is followed by a nonkeyed GET, records are accessed sequentially starting with the keyed record.

Example:

```
>100 OPEN "PUTFILE" TO :3, GET
>110 ENDFILE :3, 150
>120 GET :3, A1, B4, C7
>130 PRINT A1*(B4-C7)
>140 GOTO 120
>150 CLOSE :3
```

The form of the PUT statement is

$$[\text{line}]\,\text{PUT}\left[:\text{stream}[;\text{key}],\right]\begin{Bmatrix}\text{expr}\\ \text{aconst}\end{Bmatrix}\left[,\begin{Bmatrix}\text{expr}\\ \text{aconst}\end{Bmatrix}\right]\ldots$$

This statement writes data into a file in internal (binary) format. The expression following the colon indicates the I/O stream (thus, because of an OPEN statement, the file to be written on). The optional expression preceded by a semicolon designates the key. If a key is designated, the first items written will be to a record with that key value. A PUT statement may generate more than one record. If so, and if the statement contains the explicit key n, the records generated will have keys of value n, n + 1,...

A default form can be used

$$[\text{line}]\,\text{PUT}\begin{Bmatrix}\text{expr}\\ \text{aconst}\end{Bmatrix}\left[,\begin{Bmatrix}\text{expr}\\ \text{aconst}\end{Bmatrix}\right]\ldots$$

This is equivalent to

$$[\text{line}]\,\text{PUT}\ :2,\begin{Bmatrix}\text{expr}\\ \text{aconst}\end{Bmatrix}\left[,\begin{Bmatrix}\text{expr}\\ \text{aconst}\end{Bmatrix}\right]\ldots$$

The data values to be entered into the file may take the form of an expression or an alphanumeric constant. There is no comma following the final expression or aconst in the list. Some sample PUT statements are shown below.

```
>880 PUT "ERICEL",55,72
>881 PUT :1,TIM(X),DAY(X),YER(X)
>882 PUT FNH(A1)+P*Q
```

The FNH in statement 882 is an example of a user-defined function. These are explained under "User-Defined Functions", earlier in this chapter.

PUT statements can be used to write on files opened for PUT or opened for GET, UPDATE. In the case of updates, records can be inserted or replaced using the key option. In newly created files, the key option can also be used to write records in a nonsequential order and to replace previously written records by repeating a PUT with the same key.

The form of PUT and GET records is described in Appendix E. These records generally include 14 data values, but short records may be created by use of keyed PUT statements or as the last record written before closing the file or using the flushing technique (see "I/O Flushing" below).

Example of keyed PUT statement:

```
>300 PUT :4;121,A,B,C,D,E,F,
     G,H,K,J,K,L,M,N,O,P
```

This statement will cause BASIC to write a record via I/O stream number 4 with the key 121 and the values contained in the simple variables A to N. The values in O and P will be the first two values written on record 122. If the next PUT statement does not include a key, writing will continue on record 122. (This example was chosen to indicate the caution that should be used in mixing keyed and nonkeyed output statements.)

If a PUT statement does not fill the current output record, that record is not normally output until it is later filled, a keyed PUT is executed, or a CLOSE is executed. Short records may be forced out by using the special expression "**" (see "I/O Flushing", below).

A PUT statement may result in an error message if the selected I/O stream is not open in the proper mode, an illegal stream number is selected, or an out-of-range key is selected.

INPUT The action of an INPUT statement for file input is analogous to that of a normal INPUT statement (see "INPUT", Chapter 2) except the BCD input is routed from an open file through a specified I/O stream rather than from the terminal (on-line) or card reader (off-line). Sequential or keyed access is permitted. The form of a file INPUT statement is

[line] INPUT [:stream[;key],] input list

Example:

```
>250 INPUT :3,A(1),A(2),**
```

The "input list" is the same as for normal INPUT statements. One line constitutes one record and a single INPUT statement may access several records sequentially. If the INPUT statement specifies a key, reading starts at the beginning of the keyed record.

Attempts to input a specific keyed record that does not exist, (if no ENDIFLE is in effect) or to use an I/O stream not open for BCD input or an illegal I/O stream will result in an error exit and message.

An attempt to INPUT beyond the end-of-file gives an OUT OF DATA error exit. This exit may be modified by the user by means of an ENDFILE statement (see above).

BASIC will read records consisting of one or more blanks, and in the keyed mode it will read keyed null records.

PRINT The action of the PRINT statement for file output is analogous to that of a normal PRINT statement (see "PRINT", Chapter 2) except the BCD output is routed to an open file via a specified I/O stream. Sequential or keyed output is permitted. The form of the file PRINT statement is

$$[\text{line}]\left\{\begin{matrix}\text{PRINT}\\ ;\end{matrix}\right\}[\text{:stream}[\text{;key}]\text{,}]\ \text{print list}$$

Notice that a semicolon may be substituted for the word PRINT. The first expression in the PRINT statement is the optional I/O "stream" number. The optional second expression is the "key" value. The "print list" allows any arguments acceptable in a normal PRINT statement.

A record is generated for each line of print (governed by WIDTH, see Chapter 4), thus one statement may generate more than one record. If the print list ends with punctuation, a partial record is formed. The record is output if a full line is formed, a PRINT on the same stream ends without punctuation, or any PRINT is executed on a different I/O stream (or to the terminal). In general, it is bad practice to end a file PRINT statement with punctuation (comma or semicolon) or a TAB(0).

If a PRINT statement generates more than one record, the key is incremented by 1 for each record. This should be particularly noted if files are generated with later updates in mind or when in the update mode. If it is likely that multirecord PRINT statements will be used, the file should be created as a keyed file with key increments large enough to allow insertions and replacements without inadvertent overwrites.

Examples:

```
>400 PRINT :4;I,1,2,3,4,5,6,7,8
>410 PRINT :4;I+1,9,10,11,12
```

Statement 400 generates two records with keys I and I+1. Then statement 410 generates a new record with key I+1.

If the file is created with a key interval of 10 records, it might be generated or updated as follows.

```
>400 PRINT :4;10*I,1,2,3,4,5,6,7,8
>410 PRINT :4;10*(I+1),9,10,11,12
```

In this case, statement 400 generates records with keys 10I and 10I+1. Statement 410 generates a record with key 10I+10.

PRINTUSING The action of the PRINTUSING statement for file output is analogous to that of a normal PRINTUSING statement (see "PRINTUSING and :(Image)", Chapter 2) except the BCD output is routed to an open file via a specified I/O stream. Sequential or keyed output is permitted. The form of the file PRINTUSING statement is

$$[\text{line}]\left\{\begin{matrix}\text{PRINT}\\ ;\end{matrix}\right\}[\text{:stream}[\text{;key}]\text{,}]\text{USING line}\left[\text{,}\left\{\begin{matrix}\text{expression}\\ \text{text string}\end{matrix}\right\}\right]\ldots$$

Notice that a semicolon may be substituted for the word PRINT.

Examples:

```
>100 PRINT :1;K, USING 200, X, Y4
>110 ;:2,USING 250,"Z=",Z
```

I/O RESIDUE

In executing GET, INPUT, or READ statements the current record may or may not be exhausted of data, depending on the amount of data contained in that record. Normally, any residual data remaining in a record is retained for use by a subsequent GET, INPUT, or READ. The use of a single asterisk in a GET, INPUT, or READ causes BASIC to take an error exit if residue occurs. A double asterisk causes any residue to be discarded.

In the example shown in Figure 3 the user loads, lists, and runs a program named LISTFILE. The data used by LISTFILE is contained in a file named FILE, the contents of which are displayed in the example through use of the Edit subsystem. See the Xerox Edit (for CP-V)/Reference Manual (90 16 33) for an explanation of how Edit is used to create, display, and alter files. When executed, the program LISTFILE opens FILE to stream 1 and prompts the user to type an employee number. It then inputs a header record to B$ and inputs the first field of each subsequent record to E$(1) until it finds one beginning with 76540. The INPUT :1, ** statement in line 70 causes unwanted residue to be discarded. Line 100 inputs the remaining data from the selected record.

```
!EDIT
EDIT HERE
*EDIT FILE
*TS 1-4
'EMPL NO.       NAME          SOC SEC NO.    ADDRESS        PHONE'
10712 JACK 468-54-234 123GRT 876-0987
76540 MIKE 654-87-932 123KIU 654-6543
87654 TED 432-65-876 987PIP 876-6543
*END

!BASIC
>LOAD LISTFILE
>LIST 10-110
10 DIM E$(5)
20 OPEN 'FILE' TO :1, INPUT
30 INPUT N$
40 INPUT :1, B$
50 INPUT :1, E$(1)
60 IF E$(1)=N$ THEN 90
70 INPUT :1, **
80 GOTO 50
90 ;B$
100 INPUT :1,E$(2),E$(3),E$(4),E$(5)
110 ;E$(1),E$(2),E$(3),E$(4),E$(5)
>RUN
09:22    JUN 09   LISTFILE...
?76540
EMPL NO.        NAME          SOC SEC NO.    ADDRESS        PHONE
76540           MIKE          654-87-932     123KIU         654-6543

     110 HALT
>
```

Figure 3. INPUT Residue Example (CP-V Only)

I/O FLUSHING

If a PUT statement does not fill the current output record, that record is not normally output until it is filled by a subsequent PUT, a keyed PUT is executed, or a CLOSE is executed.

The writing of short records can be forced by use of the double asterisk (see Figure 8, Appendix E). This capability is useful primarily in the update mode.

Examples:

```
>320 PUT :4;N,A(1),A(2),A(3),**
```

This ensures that a three-element record with a key value of 4 is immediately output.

```
>440 PUT :1,**,A,B,C,**,D,E,F,**
```

This ensures that two short records are output, with three elements each. In this case, if a partial record was pending prior to executing the statement, it is output first as a short record.

LOWER CASE TEXT

CP-V BASIC allows the use of lower case characters in text strings. This capability is often useful in manipulating BCD files that are to be output on a 2741 Selectric typewriter terminal or other device capable of printing both upper case and lower case letters. A teletypewriter terminal can generate lower case characters for input to the computer by use of the ⓔ) control code, and can return to upper case by use of the ⓔ (control code, but prints all alphabetic characters as upper case.

The program shown below will read a file composed of upper case text sentences and convert it to lower case except for the first letter of each sentence. It assumes that all sentences end with a period.

```
>100 U$='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>110 L$='abcdefghijklmnopqrstuvwxyz'
>120 INPUT=$ & ENDFILE :1, 250
>130 ;'FILE NAME'TAB(0) & INPUT F$
>140 OPEN F$ TO :1, INPUT UPDATE & P=1
>150 INPUT :1,R$ & L=LEN(R$)
>160 FOR I=1 TO L
>170 IF R$(:I,1)=' ' THEN 240
>180 IF P=0 THEN 190 & P=0 & GOTO 240
>190 FOR J=1 TO 26
>200 IF R$(:I,1)<>U$(:J,1) THEN 220
>210 R$(:I,1)=L$(:J,1) & GOTO 240
>220 NEXT J & IF R$(:I,1)<>'.' THEN 240
>230 P=1
>240 NEXT I & ;:1,R$ & GOTO 150
>250 CLOSE :1
```

RUNNING CONSECUTIVE PROGRAMS

CHAIN CHAIN directs the computer to acquire and run another program of a series of programs without future action by the programmer. The format of the CHAIN statement is

```
                      ┌;password         ┐
                      │;password:acct    │
[line] CHAIN xname    │:acct             │
                      └:acct ;password   ┘
```

where name (and, optionally, password and acct) is the identity of a program as defined in the discussion of the OPEN statement.

When executed, CHAIN produces the following results:

1. The current program is discarded, but the values of its simple variables are retained.
2. The named program is obtained, compiled, and executed if possible.
3. Output buffers are flushed.

Note that only the simple variables computed by the program are retained when CHAIN is executed; all array values and dimension information are lost. A sample CHAIN statement is shown below.

```
>950 CHAIN "PART2" :A;P
```

In BTM BASIC if 'A' chains to 'B', after RUN is issued and execution is complete, the repeat of the run starts with 'A'. In CP-V BASIC the second run executes 'B'.

CHAIN LINK The CHAIN LINK statement (CP-V only) differs from the CHAIN statement in that it preserves array and string values. The CHAIN statement retains only the simple variables and discards the rest of the program.

The form of the statement is

```
                           ┌;password         ┐
                           │;password:acct    │
[line] CHAIN LINK xname    │:acct             │
                           └:acct ;password   ┘
```

An example is

```
>950 CHAIN LINK "PART3" :A;P
```

where

"PART3"	is the name of the chained program.
:A	is account A.
;P	is the password.

A and P contain alphanumeric constants.

Although CHAIN LINK preserves array context, array dimensioning is not preserved. Therefore any array used in a program that is chained to must be redimensioned in that program, via a DIM statement, if explicitly dimensioned in the first program.

When CHAIN LINK is used and a program is called from the files, CP-V BASIC beings execution with the time and date and the name of the program followed by three periods. Using the above example, the printout at the time statement 950 is executed would be

```
09:36 MAR 07 PART3...
```

If a program restarts itself by a CHAIN LINK, the program name is not repeatedly identified.

MATRIX OPERATIONS

Matrix operations in BASIC are controlled through use of a special set of MAT statements. In addition to the usual set of allowed matrix manipulations, BASIC provides options for input of matrixes via console or file, copying or matrixes, and the solution of simultaneous equations. Some of the matrix operations apply to vectors as well as to matrixes. At times, vectors are treated as either row or column matrixes. The only matrix operations that can be used with string arrays are MAT GET, MAT INPUT, MAT PUT, MAT READ, and MAT SIZE.

MAT statements may be specified to use variable dimensions, as long as these are within the dimension limits specified in DIM statements. In some cases, dimensions will vary because of the operations that are performed on them (for example, multiplication of nonsquare matrixes). Thus, there is current dimensioning (the result of the latest matrix operations, or as specified by the user in MAT SIZE statements), and absolute dimensioning (given in DIM statements). All matrix I/O is done in row major sequence.

Every array that is named in a MAT statement must be dimensioned in a DIM statement. This assures that absolute dimensions exist, sets initial current dimensions, and differentiates between vectors and matrixes. (Note that use of a letter to designate an array does not preclude the use of the same letter to designate a simple variable.) Current dimensioning may not exceed absolute dimensioning. Matrix operations and their corresponding MAT statements are presented below.

MAT GET The MAT GET statement reads array values from the currently open input file. It complements the MAT PUT statement, but can also read data prepared by ordinary PUT statements. MAT GET takes the form

    [line] MAT GET[:stream[;key],]adescr[,adescr]...

The default form

    MAT GET adescr[,adescr]...

is equivalent to

    MAT GET :1,adescr[,adescr]...

where adescr is an array descriptor of the following format:

    aname[(dimx[,dimx])]

In the array descriptor, the terms aname and dimx present the single letter array designator and dimension (subscript) expression, respectively, of the array. As shown in the general form above, therefore, an adescr term may indicate a single letter array designator, a designator followed by one subscript (a vector), or a designator followed by two subscripts (a matrix). The dimx expressions in an adescr term constitute variable dimensions; they provide a simple method for varying current dimensions during execution (not during compilation).

One example of a MAT GET statement is

```
>1008 MAT GET A(3,3), B(4,8)
```

MAT PUT The MAT PUT statement enters arrays into the currently open output file. It takes the form

    [line] MAT PUT[:stream[;key],]adescr[,adescr]...

The default form

    MAT PUT adescr[,adescr]...

is equivalent to

    MAT PUT :2,adescr[,adescr]...

There is no comma following the final array name in the list. In the sample statement

```
>1002 MAT PUT A,B
```

array A will be completely output before array B is output. Current dimensions determine how much data is output from a given array. Current dimensions may be set as part of the adescr, which is described above for MAT GET. Matrixes are entered into output files in row major sequence; that is, with the last subscript varying most rapidly.

Note that one MAT PUT may create several records. The key for the first record may be selected, but later records will have keys incremented by 1 per record. See "I/O Flushing", above, for treatment of partial records.

MAT INPUT The MAT INPUT statement is the array counterpart of the variable-oriented INPUT statement described earlier in this chapter. Format and a sample statement are shown below.

    [line MAT INPUT[:stream[;key],]adescr[,adescr]...

If ":stream" is included, input is from an open BCD input file assigned to that stream.

Example:

```
>1007 MAT INPUT A(3,4), B
```

When statement 1007 is executed, 12 values must be supplied for the 3 x 4 matrix A. These values are then followed by input for array B. Note that the number of values input to array B must match the current dimensions for B.

The rule for the use of commas and empty fields in the data list read by MAT INPUT is the same as described for the INPUT statement.

MAT PRINT The MAT PRINT statement prints arrays in regular or packed format. The form of the MAT PRINT statement is

```
[line] MAT PRINT[:stream[;key],]aname[{,}  ─┐
                                      {;}   │
└─ aname]...[,]
            [;]
```

where aname is the letter designation of an array that has been dimensioned in a DIM statement. Some sample MAT PRINT statements are shown below.

```
>1000 MAT PRINT A, B; C
>1010 MAT PRINT D;
```

There are two types of print formats, regular, or packed. An aname parameter followed by a semicolon causes the named array to be printed in packed format. Otherwise, regular format is used. Statement 1000 will cause array A to be completely printed before any element of array B is printed. Array A will be printed in regular format, array B in packed format, and array C in regular format. Current dimensions are used to determine how much data is printed from an array. Statement 1001 will cause array D to be printed out in packed format.

Vectors are printed in row fashion. Each row of a matrix is printed as one or more consecutive print-rows with a blank line between successive matrix-rows. Column 1 of a matrix always occurs in the leftmost print field.

Notes:
1. If MAT PRINT :stream is used, each line of output creates one record. In general, more than one record will be created. A record containing one blank byte is created after the last MAT PRINT record is written.
2. Not applicable to text arrays.

MAT READ The MAT READ statement is similar to the READ statement described in Chapter 2, except that it acquires whole arrays of data, rather than just single data items. MAT READ has the form

[line] MAT READ adescr[,adescr]...

where adescr is an array descriptor of the same format previously explained under "GET", namely

aname [(dimx[,dimx])]

If dimension expressions are included in the array descriptor, they specify current array dimensions. If they are omitted, current dimensioning results from previous conditions. Some samples are shown below.

```
>1005 MAT READ A(K/L+M,7)
>1006 MAT READ B, C(3,4)
>1007 MAT READ D
```

MAT SIZE The MAT SIZE statement redefines current dimensions of the named array. MAT SIZE has the form

[line] MAT SIZE aname (dimx[,dimx]) [,aname(dimx[,dimx])]...

Examples:

```
>1003 MAT SIZE A(X+Y,Z), B(3)
>1004 MAT SIZE D(4,5)
```

where

aname	is the single letter designator of an array.
dimx	is any expression representing a legal subscript.

ASSIGNMENT FUNCTIONS

The matrix assignment functions resemble the LET statement in form and function. An array name (aname) always appears immediately to the left of the equals sign and the named array is assigned values according to the specifications to the right of the equals sign.

ZERO

This statement zeros those elements of the named array that fall within the range of current dimensions. Its format is

[line] MAT aname = ZER[(dimx[,dimx])]

The optional dimx terms have the same meaning described previously under MAT GET; that is, they are dimension expressions specifying new current dimensions (subscripts) of the array, and as such constitute variable dimensions. For example, assuming the array B(5,5), that is, a 5 x 5 matrix named B, the following statement,

```
>1010 MAT B=ZER(3,2)
```

will zero elements (1,1),(1,2),(2,1),(2,2),(3,1), and (3,2), leaving the remainder of the matrix unchanged. Other examples are

```
>1020 MAT C=ZER(10)
>1030 MAT D=ZER
```

CONSTANT

This statement is analogous to that discussed above, except that it sets matrix elements to 1 (instead of 0). Its format is given below.

[line] MAT aname = CON[(dimx[,dimx])]

Example:

```
>2250 MAT L=CON(3,5)
```

IDENTITY MATRIX

This statement forms an identity matrix. Since the array must be a square matrix, new current dimensioning may have to be provided. Two forms are shown below.

[line] MAT ananme = IDN[(dimx)]

[line] MAT aname = IDN[(dimx), any characters to end of line]

The first form, which gives the current dimension for the row value only, is sufficient to define the square matrix.

second form is provided for those users who wish to clearly indicate that the array is a square matrix. However, it may stand alone. Appropriate samples are shown.

```
>1011 MAT Z=IDN
>1012 MAT B=IDN(4)
>1013 MAT C=IDN(3,3)
>1014 MAT D=IDN(X+Y)
```

The result of this operation is a matrix in which all elements of the principal diagonal are set to 1 and all other elements are set to 0.

Example:

```
>DIM A (3,3)
>MAT A=IDN
>MAT PRINT A;

 1   0   0

 0   1   0

 0   0   1
>
```

COPY

The copy statement copies arrays, and sets current dimensioning of the array copied <u>into</u> to that of the array copied <u>from.</u> The form of the copy statement is

[line] MAT aname = aname

In the sample copy statement below,

```
1015 MAT A = B
```

matrix B is copied into matrix A. Assume that B is a 4 x 4 matrix with current dimensioning (3,3). Only elements (1,1), (1,2), (1,3) ... (3,3) from B are copied into A. Further, any remaining elements in array A are not changed, and the current dimensioning of A becomes (3,3).

SCALAR MULTIPLICATION

The scalar multiplication operation multiplies an array by a scalar quantity. The form of the statement is

[line] MAT aname = (expr)* aname

where expr is an expression representing the scalar multiplier, and the parentheses are required. In sample statement 1016, below,

```
>1016 MAT A=(A)*A
>1017 MAT B=(SIN(X+H))*C
```

the parenthesized A is interpreted by BASIC as a simple variable, not as an array name.

Scalar multiplication can be used to set all elements of an array to any constant value by using a MAT CON statement followed by a scalar multiplication. An example is

```
>10 MAT A=CON
>20 MAT A=(10)*A
```

This example sets all elements of array A to the value 10.

ADDITION AND SUBTRACTION OF ARRAYS

Array addition and subtraction are performed through use of the statement shown below.

[line] MAT aname = aname$\begin{Bmatrix} + \\ - \end{Bmatrix}$aname

This statement adds or subtracts the corresponding elements of the two arrays named on the right of the equals sign and stores the results in the array named on the left. A sample is given for reference.

```
>1018 MAT Z=B+C
```

The ambiguity introduced by allowing addition or subtraction of two vectors with storage in a matrix, or copying, transposition, and scalar multiplication of a vector into a matrix is resolved by considering the vectors as row vectors. Current dimensions of both arrays named to the right of the equals sign must be equal for addition and subtraction. The current dimensions of the array named on the left side of the equals sign are set equivalently.

TRANSPOSITION

It is not necessary to transpose a vector array; the result is an exact copy of the argument vector. Matrixes are transposed by use of the statement shown below.

[line] MAT aname = TRN(aname)

Sample statements are shown below

```
>1019 MAT A=TRN(A)
>1020 MAT B=TRN(C)
```

Current dimensioning of the matrix named on the left side of the equals sign is set consistent with current dimensioning of the matrix named on the right.

MULTIPLICATION

In the multiplication operation, vectors are taken to be row or column matrixes as appropriate. If a vector is multiplied by a vector, the scalar (dot) product results. The form of the multiplication statement is

[line] MAT aname = aname*aname

The following is a sample multiplication statement.

```
>1021 MAT Z=B*C
```

<u>Notes:</u>
1. Current dimensioning must be consistent with the usual rules of matrix multiplication.
2. The same array name may not appear on both sides of the equals sign.

INVERSION

The inverse of a square matrix is specified as shown below.

[line] MAT aname = INV(aname[,simple variable])

where the inclusion of the simple variable (in which to store the computed determinant of the argument matrix) is a user option. Some sample statements are

```
>1022 MAT A=INV(H)
>1023 MAT B=INV(I,D)
```

In calculating the inverse of the square (by current dimensioning) argument matrix, the target matrix is initially set to an identity matrix. Then the target is converted by those elementary row operations that reduce the argument matrix to the identity matrix. Upon completion of the conversion, the target matrix is approximately the inverse of the argument matrix. The values of the argument matrix are destroyed; both matrixes have current dimensioning originally applicable to the argument matrix.

Notes: 1. The argument matrix must be square (according to its current dimensions).

2. Results are approximate, not exact.

3. At the user's option, the computed determinant of the argument is stored in a simple variable.

4. The contents of the argument matrix are destroyed, but current dimensions remain.

5. The same array name may not appear on both sides of the equals sign.

SIMULTANEOUS EQUATION SOLUTION

Solution of simultaneous equations is accomplished via the statement shown below.

[line] MAT aname = SIM(aname[,simple variable])

where the simple variable modification is a user option.

Some sample statements are shown below.

```
>1024 MAT M=SIM(E)
>1025 MAT S=SIM(H,D2)
```

The target array contains one or more sets of linear equation constant column vectors. The dimensions of this array must be compatible with the square argument matrix. For example, if the argument matrix has current dimension of (n,n), the target array must be either an n-dimension vector (one solution, or else an n x m matrix (m solutions). The argument matrix contains the coefficient matrix. The solution of the simultaneous equations is arrived at by converting the target array by those elementary row operations that reduce the argument matrix to the identity matrix. Upon completion of the conversion, the values of the argument matrix are destroyed, but current dimensions for both the target and the argument arrays are unchanged. The target array contains the appropriate values that are computed by taking

$(\text{argument})^{-1} \times (\text{target})$

This result is equivalent to solving one or m sets of simultaneous linear equations having the same coefficient matrix, that is, the argument matrix.

Notes: 1. The argument matrix must be square (according to its current dimensions).

2. Results are approximate, not exact.

3. At the user's option, the computed determinant of the argument is stored in a simple variable.

4. The contents of the argument matrix are destroyed, but current dimensions remain.

5. The same array name may not appear on both sides of the equals sign.

ACCURACY OF INVERSION AND SIMULTANEOUS EQUATION SOLUTION

The results of matrix inversion will vary in accuracy because of precision losses during the conversion process. If, during conversion, a pivotal element is smaller in magnitude than 10^{-13}, it is considered to be zero and the matrix is considered singular. If all elements of a matrix are of small magnitude (e.g., 10^{-6} or less), it should be scaled upward so the greatest magnitude of any element is near unity. If a matrix consists of elements of large magnitude, it should be scaled down again to near unity for the maximum element.

When a determinant calculation is requested in using the inversion or simultaneous equation functions, the following special situations may occur:

1. If the determinant value calculation results in a magnitude greater than 7.234×10^{75}, the value of the simple variable will be the alphanumeric value OVERFL. This does not affect the calculation of the inverse or simultaneous equation solution.

2. If the matrix is singular, the simple variable is given a value of zero; the values of the argument and target arrays are destroyed.

4. BASIC COMMANDS

The BASIC commands described in this chapter are instructions to the compiler. They are never used in numbered statements.

ACCOUNT and PASSWORD

The command

ACC[OUNT] [name]

establishes the name of an account that is to be accessed by a LOAD, WEAVE, or CATALOG command (see below). The name may have from one to eight characters and may contain blanks or other nonprinting characters if the name is enclosed in single or double quotes. If the name begins with the letter O and the command word is abbreviated to ACC, the use of quotes is mandatory.

The account name is reset to the log-in account by execution of a CLEAR, CLOSE, FILE, SAVE, or RENUMBER command or by an ACCOUNT command in which no name is specified. Execution of an OPEN, CHAIN, or CHAIN LINK[t] statement causes the account name to be set to that specified in the statement or, by default, to the log-in account.

The command

PAS[SWORD] [name]

establishes a password to be used in executing FILE, SAVE, LOAD, WEAVE, and RENUMBER commands (see below). Restrictions on the name are the same as for ACCOUNT, except that if the name begins with the letter S and the command word is abbreviated to PAS the use of quotes is mandatory. As with ACCOUNT, the password is subject to change by the execution of a CLEAR, CLOSE, FILE, SAVE, or RENUMBER command, or an OPEN, CHAIN, or CHAIN LINK statement. If no password is specified in a PASSWORD command, a null password is assumed.

In the examples shown below, ACCOUNT and PASSWORD set the file attributes, and ACC and PAS reset them.

```
ACCOUNT   LBRY1
ACC

PASSWORD  '3731'
PAS
```

Note: If a file is declared temporary (e.g., through use of the TFILE option in an OPEN statement) it is released at the end of the job or terminal session unless it is created with a password.

[t]CP-V only.

CATALOG[t]

Typing the command

CAT[ALOG]

causes BASIC to print, via the M:SL DCB, the names and attributes of all files in the user's account or the account specified by an ACCOUNT command (see above).

```
>ACCOUNT 6101
>CATALOG

ACCOUNT - 6101

 O GRAN   HR:00 MO/DY/YR    FILE NAME

 K    3   09:00 06/19/72   BITS1
 C    2   09:00 06/19/72   BO
 K   43   09:00 06/19/72   BOMP
 K    2   09:00 06/19/72   BOMPALL
 R   26   09:00 06/20/72   BOMPBASE
>
```

The listed attributes are file organization (keyed, consecutive, or random), number of granules, time and date of creation, and file name.

LOAD and WEAVE

To load a file containing numbered BASIC statements, the programmer types

LOA[D] [xname]

If xname is not specified, the current runfile name is used. Current PASSWORD and ACCOUNT information are used. If no file exists, the following message is output:

```
UNABLE TO OPEN
```

If the file exists, records are read and handled analogously to line insertion via the terminal. Error messages are generated if syntax errors are encountered. Direct statements and commands, which have no associated line numbers, may not be read using LOAD.

If the xname begins with a "D", the LOAD command must not be abbreviated to LOA unless the xname is in quotes.

If the LOAD command is typed while BASIC is in compilation or run mode (see the STATUS command, below), the current program is deleted prior to the load. If BASIC is in edit mode, LOAD "weaves" the file statements into the existing program by step number. If a statement in the loaded file has the same number as one in the existing program, the file statement replaces the existing statement.

A direct CHAIN or CHAIN LINK[†] statement is one means of replacing a current program with a new program and retaining data context instead of weaving.

The WEAVE[†] command is equivalent to a LOAD executed in edit mode. WEAVE has been added to provide weaving action on a program load regardless of current operating mode. The syntax is

WEA[VE][xname]

LIST

To list a line or a series of lines, the user types

$$\text{LIS[T]}\begin{bmatrix}\text{line}\\ \text{line}_1 - \text{line}_2\end{bmatrix}\left[,\begin{Bmatrix}\text{line}\\ \text{line}_3 - \text{line}_4\end{Bmatrix}\right]\ldots$$

which will cause listing of the appropriate lines from the program. If the user types

```
>LIST
```

all lines will be listed. If none exist, the NO PROGRAM message will be output.

DELETE and EXTRACT

To delete a line, the programmer types the line number followed by a New Line, and the line with that number will be deleted. If the line does not exist in the program, the line number will be output along with the following error message:

```
NO PROGRAM
```

To delete specific lines or series of lines, the user may type

$$\text{DEL[ETE]}\begin{Bmatrix}\text{line}\\ \text{line}_1 - \text{line}_2\end{Bmatrix}\left[,\begin{Bmatrix}\text{line}\\ \text{line}_3 - \text{line}_4\end{Bmatrix}\right]\ldots$$

after the prompt, as in

```
>DEL 72
```

or

```
>DEL 85 - 97, 112 - 124
```

If the line or series of lines is not found, the NO PROGRAM message will be output preceded by the specified line or line series numbers. If an illegal line number is used, the LINE # ERROR message is printed, preceded by the first six characters of the illegal line number. The NO PROGRAM message does not inhibit further processing of the DEL line, whereas LINE # ERROR does inhibit further processing.

[†]CP-V only.

To delete an entire program, it is faster to use CLEAR rather than DELETE. Further, the use of CLEAR recovers all the space in the text editing area.

The DELETE command may also be used to delete a named file in the user's account. The syntax is as follows:

DEL[ETE] xname

If a password is currently in effect, it is applied. For example, to delete a file named FILE1 with password SECRET, the user would execute two commands.

```
>PASSWORD 'SECRET'
>DELETE 'FILE1'
```

In this case the user should also type

```
>PASSWORD
```

to remove the password SECRET prior to other file operations.

Note: In BASIC, file names may be created with embedded blanks or commas. Such names are not readily accessible to other subsystems such as EDIT. The file deletion capability within BASIC has been added primarily to allow the user to delete files which are inaccessible in EDIT, FERRET (BTM), or PCL (CP-V).

The programmer may delete a complete program except for certain areas designated by their line numbers or line number groups. This is done by typing

$$\text{EXT[RACT]}\begin{bmatrix}\text{line}\\ \text{line}_1 - \text{line}_2\end{bmatrix}\left[,\begin{Bmatrix}\text{line}\\ \text{line}_3 - \text{line}_4\end{Bmatrix}\right]\ldots$$

If no line numbers are specified, no operation takes place and a prompt is issued to the programmer.

RENUMBER

The programmer may specify that his program statements be renumbered, starting with a certain line number and renumbering in specified increments. The command to be typed is

REN[UMBER] [line$_1$[,line$_2$[,incr]]]

where

line$_1$	is the lowest new line number.
line$_2$	is the point in the program at which to start renumbering.
incr	is the increment by which the new line numbers are to be spaced.

For example, the sample statement

```
    >REN 135, 170, 5
```

will cause the computer to change statement number 170 to number 135, the next statement number to be changed to 140, etc., until the end of the program is reached. Default values for the parameters in the RENUMBER statement are 100, 1, and 10, in that order. When a replacement is made for a step number, replacement is also made for any occurrence of the same step number within any statement in the program. Illegal RENUMBER syntax causes output of the message

```
ILLEGAL
```

Also, if the renumbering process generates a line of more than 132 characters, the following message is output:

```
LINE TOO LONG
```

If RENUMBER increments a line number past the upper limit of 99999, the following message is output:

```
LINE # ERROR
```

If the renumbering process cannot be completed because of errors, the original program will remain unaltered.

A RENUMBER operation causes the renumbered program to be saved as a temporary file with the current runfile name. ACCOUNT is reset and if the runfile name is the default name, PASSWORD is reset. When the operation is completed, this file is loaded to replace the existing version of the program.

NAME and FILE

BASIC establishes a default name for a temporary file that may be used as required for program storage. This file is referred to as the runfile. If the user types

NAM[E][xname]

the name of the runfile is changed to the xname given. If no xname is given, the default runfile name is restored. NAME Tfiles the given xname. The xname may be an alphanumeric constant enclosed in single or double quotes or an alphanumeric constant, with no embedded blanks, not enclosed in quotes. Xname is restricted to 11 characters at most, not counting enclosing quotes.

If the programmer types

FIL[E] [PAC[K]][†]

[†]The PACK option applies to CP-V BASIC only.

the source program currently in the text editing area is copied with the current runfile name, which has been Tfiled. This file will be temporary unless there is a current password, and the current runfile name is not the default name.[†]

The PACK option serves to recover the space that was occupied by discarded text. If the option is exercised, program text is read back into core memory in edited format, that is, without the empty spaces due to deletions.

FILE does not modify program context. FILE PACK sets BASIC to the editing mode and inhibits returning to the execution without recompiling object code. Array and string context is preserved, as is DATA pointing control and GOSUB-RETURN status.

SAVE ON and SAVE OVER

The user may save selected program statements on permanent files. The command is

$$\text{SAVE[E O]}\begin{Bmatrix}\text{N}\\ \text{VER}\end{Bmatrix}\text{xname}\begin{bmatrix}\text{line}\\ \text{line}_1\text{ - line}_2\end{bmatrix}\left[,\begin{Bmatrix}\text{line}\\ \text{line}_3\text{ - line}_4\end{Bmatrix}\right]\ldots$$

If the SAVE ON form is used, a check will be made that no file exists with the same xname before opening for output. If none exists, an output file is created. The current PASSWORD information is used and ACCOUNT is reset.[†]

SAVE OVER creates the file unconditionally, with current password information. ACCOUNT is reset. The file is <u>permanent</u> unless two conditions are met: the file name is TFILEd sometime during the terminal session (or batch job), and the file has no password.

Samples of SAVE ON and SAVE OVER are given below.

```
    >SAVE ON 'PERM2' 1-55, 80-105
    >SAVN 'A2' 1,5,10,15,30-40
    >SAVE OVER PERM
    >SAVVER 'END' 50-59, 120
```

The third example above causes all lines of the current program to be saved in a permanent file (named PERM). This is the most commonly used form of the command.

CLEAR

To erase everything in the text editing area, the user types

CLE[AR]

This command is useful in avoiding the cumulative effect of successive LOAD commands (see "LOAD and WEAVE", above).

[†]CP-V only.

In addition to the capability existing in Batch/BTM BASIC of clearing all program context, CP-V BASIC permits clearing of arrays or strings only. This form of the command is

$$\text{CLE[AR]}\begin{bmatrix}\text{ARR[AYS]}\\\text{STR[INGS]}\end{bmatrix}$$

It releases storage for the indicated option.

SET

To set the values of vector and matrix arrays, the programmer types

SET letter = digits[,letter = digits]...

where the value of the digit string (interpreted as a decimal integer) is assigned to the appropriate declared letter parameter as indicated by the alphabetic character. These parameters are used by the compiler for compilation of DIM statements that contain set letters.

Example:

A program includes the statement

```
20 DIM C(B,B+5)
```

B was set to 8. Absolute dimensions will be (8, 13).

SET can also be used in CP-V BASIC to establish maximum string length for allocation of string storage. The default value is 72 and 132 is the highest possible value.

This form of the command is

SET $ = positive integer

Example:

```
>SET $=12
```

allocates maximum storage of 12 characters for any string. Strings with more than 12 characters are truncated and length count for such strings is reduced to 12.

In CP-V the SET command also causes subsequent reallocation of array-string storage as required for new parametric dimensioning or new string lengths. During array-string storage reallocation, context is saved as new maximum dimensions permit.

Note: SET does not assign values to simple variables. If SET is used for arrays that will be filled by FOR-NEXT loops, the parameterized DIM should be established by a SET and the FOR-NEXT loop by a direct LET. Example:

```
>SET X=5, Y=10
>LET X=5, Y=10
```

ENTER BASIC

If the programmer types

ENT[ER BASIC][L]

the extended precision print indicator will be set or reset, depending on whether the optional L is or is not typed.

WIDTH

To change the PRINT width from its default setting of 72, the user types

WID[TH]digits

where the value of the digit string, interpreted as a decimal integer, must be within the allowed maximum and minimum values of 132 and 32, respectively.

WIDTH affects file I/O as well as terminal I/O, and excess characters are printed on the following line, WIDTH cannot exceed the PLATEN setting (see the CP-V User's Guide, 90 16 92) for terminal I/O but file I/O is not affected by the PLATEN setting.

RUN and FAST

To change from the editing mode to the compilation and execution mode the user types

$$\begin{Bmatrix}\text{RUN}\\\text{FAS[T]}\end{Bmatrix}\text{[time]}$$

This causes the program to be copied onto the runfile, if necessary.[†] The computer then compiles from the runfile or from the text edit area. If RUN was specified, compilation will take place in the safe mode, in which all variable subscripts will be checked against absolute dimensions; otherwise, compilation will proceed in the fast mode. If no errors are found, the program will be executed. If the compilation contains errors, the editing mode is restored and a prompt character is issued at the console. The user may specify a maximum number of seconds of CPU time expended by the program. If this time is exceeded, execution is aborted with the message

EXEC TIME LIMIT

If a FAST or RUN command is given in the editing mode prior to the creation of a program, BASIC will enter the execution mode and will allow direct execution of statements from the console as described in Appendix D under "Direct Statements". If input is required from the programmer during execution, the BASIC program will issue a question mark at the console. When execution of a program or a direct statement is complete, a prompt character is issued. Additional statements then may be directly executed, or the programmer may revert to the editing mode by typing an editing command.

[†]BPM/BTM only.

Under BTM or BPM, both RUN and FAST reset account information, if any exists. If the standard runfile is used, password information is reset, if any was set. Under CP-V, RUN and FAST do not affect runfile nomenclature.

BTM BASIC starts each execution with the time and date. CP-V BASIC also gives the name of the program followed by three periods. When CHAIN LINK is used (in CP-V BASIC) and a program is called from the files, its name will be similarly printed; if a program restarts itself by a CHAIN LINK, its name is not repeated. The default name for the runfile has the form RUNxxxx, where the x's represent letters. Example:

```
    >RUN
    11:09    JUN 09   RUNMAAA...
```

BREAK and ESC

A single activation of the BREAK key (CP-V BASIC) or a double activation of the ESC key (BTM BASIC) causes the current operation to be stopped. BASIC then requests a new command by issuing a > prompt character to the terminal. If the BREAK key is depressed again (or the ESC key is depressed twice again) control passes to the operating system executive program and a ! prompt is given.

STATUS

To determine the status of his program at any time, the programmer types

STA[TUS]

The system will respond with one of the three messages: EDITING, COMPILING, or RUNNING, as appropriate. To facilitate loop detection, the RUNNING message is preceded by an appropriate line number if execution was interrupted while one of the instructions in the compiled program was being executed.

BASIC

The BASIC command[†] displays the following parameters which may be set by other commands: WIDTH, PRC, $ (maximum string length), and any letters that have been SET to provide parameterized array dimensions. This display is automatically generated at the start of a batch run in BPM/BTM BASIC. The command has the form

BAS[IC]

[†]CP-V only.

NULL

The NULL command[†] sets the value of specified items to zero but does not release or reallocate storage, nor does it modify current or maximum dimensioning of arrays. It has the form

$$\text{NUL[L]}\begin{bmatrix}\text{ARR[AYS]}\\ \text{STR[INGS]}\\ \text{SIM[VARS]}\end{bmatrix}$$

where

ARRAYS sets the value of all array elements to zero for current dimensioning.

STRINGS sets the length of all string scalars and string array elements to zero.

SIMVARS sets all simple variables to zero.

NULL performs all three options.

NULL does not modify source text or object code; however, zeroing simple variables affects FOR-NEXT statements in that this resets any loop control variables.

EXECUTE

The EXECUTE command[†] performs two distinctly different functions: it specifies a single, numbered statement for direct execution; and it specifies a starting point for execution, with a halt at a specified line number. The first use is for the programmer's convenience, saving him time by letting him specify a line number instead of requiring him to type the entire line. The second use is a valuable aid in debugging and verifying program segments without executing the entire program, as would be the case with a direct GO TO statement.

The command has the form

EXE[CUTE]line$_1$ [-line$_2$]

where the optional line 2 specifies the line number before which execution will halt. In the program example shown below, EXE 40 would be equivalent to typing PRINT A1, A2, A3, A4, A5; and EXE 40-70 would be equivalent to typing 70 STOP Ⓡ and GO TO 40, except that in the case of EXE 40-70 the halt at 70 is temporary; that is, the line is restored to normal after the halt has occurred. In this example, if program returns from subroutine 110, line 60 is executed, followed by the message

```
70 -EXEC- HALT
```

EXECUTE with only line$_1$ is subject to all restrictions applying to direct statements (see Appendix D). EXECUTE with line$_2$ option is not restricted.

Example:

```
>40 PRINT A1,A2,A3,A4,A5
>50 GOSUB 110
>60 PRINT A2
>70 GOSUB 140
```

PROCEED

If, following escape from execution, the command

PRO[CEED]

is typed after the prompt character, processing continues with the core memory (but not file) conditions that were in effect immediately preceding the most recent activation of the BREAK or ESC key. Alternatively, the user may give a direct command such as GOTO or EXECUTE (see above).

SYSTEM, BYE, and OFF

To exit from BASIC, the programmer depresses the BREAK or ESC key four times and waits for the executive level prompt (!).

Alternatively, the programmer may use the SYSTEM command to return to the executive program. The command has the form

SYS[TEM]

At this point, he may invoke another processor that operates under BTM or CP-V control, or he may vacate the terminal by typing

! BY[E] (or OFF under CP-V)

If the user wants to log out of the system directly from BASIC[t], he may do so by typing either

```
>BYE
```

or

```
>OFF
```

without using the SYSTEM command.

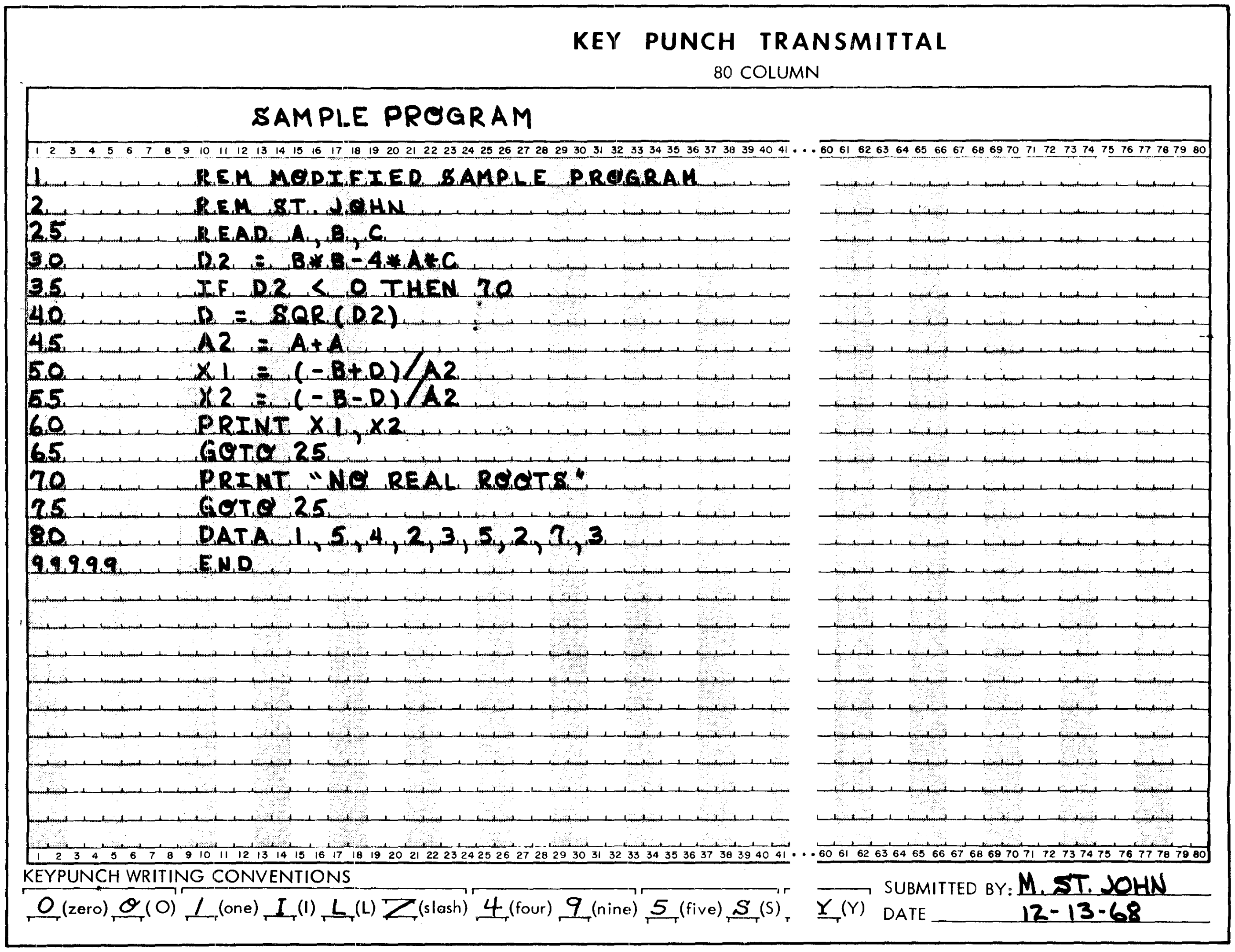
[t]CP-V only.

5. BATCH PROCESSING

The programmer prints his program on a coding sheet (see Figure 4). The coding sheet is divided into 80 columns (corresponding to a standard data-processing punch card) and a number of rows or horizontal lines. Each line represents one card and each interval on a line represents one character space in the corresponding column of the card. Note that all characters take up only one column each. Conventions have been adopted for manually printing certain confusing characters, especially characters 0 and O, 2 and Z, and 1 and I (see Figure 4). It is advisable to check with the computer facility's keypunch operator for notation conventions.

When completed, the coding sheet is sent to the programmer's Keypunch facility. There the program is punched onto cards, and the deck of cards returned to the programmer. The deck, along with required BPM control cards, can now be sent to the computer facility to be run on the computer. Figure 5 shows a typical deck structure for a BASIC program that is run under BTM or BPM. Note that the dashed lines indicate optional control cards. Card parameters are explained below the drawing of the card deck.

Figure 6 shows a typical deck structure for a BASIC program that is run under CP-V. Note that a subset of on-line editing commands is used in this deck structure instead of the special control cards (items 4, 5, 7, and 9 in Figure 5) that are used in a BPM or BTM deck structure. Other on-line commands, such as LOAD, may be used in CP-V BASIC in batch mode, but they are oriented primarily toward conversational operations and should be used with caution.



KEY PUNCH TRANSMITTAL

80 COLUMN

SAMPLE PRØGRAM

```
1       REM MØDIFIED SAMPLE PRØGRAM
2       REM ST. JØHN
25      READ A, B, C
30      D2 = B*B-4*A*C
35      IF D2 < 0 THEN 70
40      D = SQR(D2)
45      A2 = A+A
50      X1 = (-B+D)/A2
55      X2 = (-B-D)/A2
60      PRINT X1, X2
65      GØTØ 25
70      PRINT "NØ REAL RØØTS"
75      GØTØ 25
80      DATA 1, 5, 4, 2, 3, 5, 2, 7, 3
99999   END
```

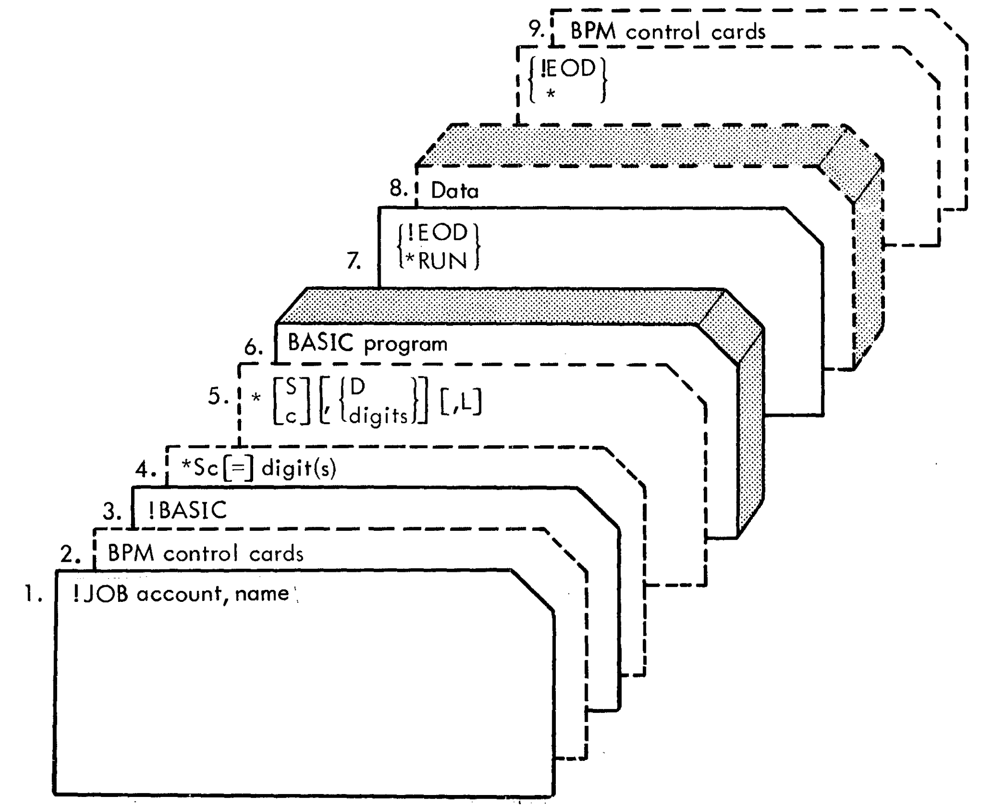
KEYPUNCH WRITING CONVENTIONS

0 (zero) Ø (O) 1 (one) I (I) L (L) Z (slash) 4 (four) 9 (nine) 5 (five) S (S) Y (Y)

SUBMITTED BY: M. ST. JOHN

DATE 12-13-68

Figure 4. Coding Sheet with Sample Program



Card	Parameter	Description
1	!JOB account, name	Signals the beginning of a job. This card is required.
2	BPM control cards	May include BPM control cards such as !LIMIT and !ASSIGN.[†] These cards are optional. For information concerning DCB assignments see Appendix F.
3	!BASIC	Calls the BASIC processor. This control card is required.
4	*Sc[=]digit(s)	One or more parameter setting cards to assign the designated constant value to the declared letter (used for compilation of DIM statements that contain declared letters). Column 1 must contain an *, followed by an S in column 2. Column 3 contains the single-letter designation of an array. Column 4, always interpreted as =, is followed by a constant value.
5	*[S / c]	Optional declaration card must have an * in column 1, followed by a blank. An S in column 3 indicates the safety mode for array references. If the safety mode is specified, the compiler checks the subscripts of subscripted variable references against the dimensions. Any other character (indicated by c) in column 3 indicates the fast mode. The default condition is "fast".
	[,{D / digits}]	D is the default case for the printer line width (100). If a digit appears in column 5, successive columns of the cards are scanned until a nondigit

[†]For information on the format of these control commands, see the Xerox BPM/BP, RT Reference Manual, 90 09 54 and the Xerox BPM/OPS Reference Manual, 90 11 98.

Figure 5. Deck Setup for BASIC Batch Processing (BTM or BPM)

Card	Parameter	Description
5	$\left[,\begin{Bmatrix}D\\digits\end{Bmatrix}\right]$ (cont.)	character is encountered. The digits are then interpreted as the printer line width (subject to maximum and minimum width parameters, currently 131 and 32, respectively). It is recommended that the D or digit(s) be followed by a comma. If neither a D nor a digit appears in column 5, the rest of the card is ignored.
	[,L]	An L in the column following the comma causes listing of all records following the option declaration card up to an end-of-file mark, or an !EOD or *RUN record. (*RUN will be listed, but !EOD will not.) This card is optional.
6	BASIC program	The programmer's BASIC program, in line-number order.
7	$\begin{Bmatrix}!EOD\\*RUN\end{Bmatrix}$	End of program. If no errors are detected by the compiler, the program will be executed, beginning at the statement with the lowest line number. If this card is omitted, the program will not be executed, even if no errors are detected. This allows compilation for diagnosis only.
8	Data	Data to be used in response to INPUT statements. The data should be followed by an end-of-file mark, an !EOD, or a record with an * in column 1.
9	BPM control cards	Other BPM control cards as required. In particular, the temporary files created via the O option of OPEN statements must be copied to permanent files if the programmer wants to retain the information.

Figure 5. Deck Setup for BASIC Batch Processing (BTM or BPM) (cont.)

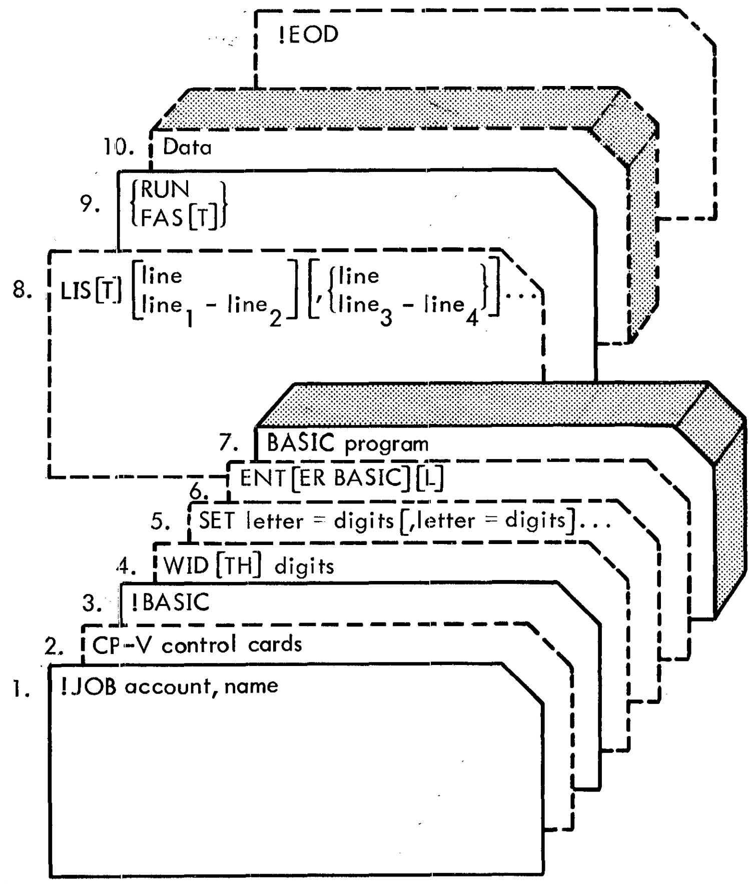


Figure 6. Deck Setup for BASIC Batch Processing (CP-V)

Card	Parameter	Description
1	!JOB account, name	Signals the beginning of a job stack. This card is required.
2	UTS control cards	May include CP-V control cards such as !LIMIT and !ASSIGN.[†] These cards are optional. See Appendix F for information concerning DCB assignments.
3	!BASIC	Calls the BASIC processor. This control card is required.
4	WID[TH] digits	Changes the width of the printer from its default value of 72 to the specified value (digits). All commands may be started in column 1 or may be preceded by blank columns.
5	SET letter = digits [,letter = digits]...	Sets the parameter values of the dimensions of vector and matrix arrays. The value of the digit string, interpreted as a decimal integer, is assigned to the appropriate declared letter parameter as indicated by the alphabetic character. These parameters are used by the compiler for compilation of DIM statements that contain SET letters.
6	ENT[ER BASIC] [L]	Sets the extended precision print indicator when L is included; resets the indicator when L is not included. The default is normal precision.
7	BASIC program	This is the programmer's BASIC program. In CP-V, it need not be ordered sequentially. Note that any run-time error or END, STOP, or PAUSE statement causes an exit from BASIC after a diagnostic printout.
8	$\text{LIS[T]}\begin{bmatrix}\text{line}\\ \text{line}_1 - \text{line}_2\end{bmatrix} \left[, \begin{Bmatrix}\text{line}\\ \text{line}_3 - \text{line}_4\end{Bmatrix}\right]\ldots$	Lists a line or a series of lines from the program. If LIS[T] is used alone, all lines of the program will be listed.
9	$\begin{Bmatrix}\text{RUN}\\ \text{FAS[T]}\end{Bmatrix}$	If no errors are detected by the compiler, these commands cause the program to be executed, beginning at the statement with the lowest line number. If this card is omitted, the program will not be executed, even if no errors are detected. The RUN card causes the program to be compiled in the safe mode, in which all variable subscripts will be checked against absolute dimensions. The FAS[T] card causes the program to be compiled in the fast mode.
10	Data	Data cards to be used in response to INPUT statements. The data cards should be followed by an !EOD card.

[†]For information on the format of these control commands, see the Xerox CP-V/BP Reference Manual, 90 17 64.

Figure 6. Deck Setup for BASIC Batch Processing (CP-V) (cont.)

6. BASIC MESSAGES

This chapter lists BASIC error messages and other messages in alphabetical order. Messages that do not apply to all three monitors are designated "CP-V only" or "not in CP-V". In the messages, xxxxx represents step number and x represents array name, function letter, or declared letter. Except as noted in comments, an error causes termination of program execution.

There is a special type of error message in BTM BASIC, resulting from disk input/output errors. Disk input/output error messages are transferred from the Monitor and are printed by BASIC. A sample error message is

```
I/O     ERROR    57
DCB     NAME     M;EO
FPT     CODE     X'11'
```

BASIC remains in control for continued operation. See the Xerox BPM/BP, RT Reference Manual, 90 09 54 for a detailed description of this type of error message.

xxxxx ACONST EXPECTED

CP-V only. A variable contains numeric data when it should contain an aconst.

xxxx ARG NO. ERR FNx

Conflict between the number of arguments defined and the number of arguments used with the function.

ARRAY CLASS CONFLICT

CP-V only. The indicated letter is used for more than one type of array. Example: B used for a string vector but dimensioned as a numeric vector.

xxxxx ASN-ACS ARG ERROR

CP-V only. The argument is outside the allowable limits ±1.

BAD BYTE

CP-V only. In executing a CHANGE statement, a value was not in the range 0-255.

xxxxx BAD CHAR

Statement xxxxx contains an illegal character (for example, !, ?,@, etc.). Note, however, that all characters having the EBCDIC value of blank or greater are allowed in Image statements, text strings, and alphanumeric constants.

xxxxx BAD CONST

Line xxxxx contains an improperly formed numeric or alphanumeric constant. Probable causes are

Numeric

1. Extra decimal points.
2. More than two digits in exponent fields (for example, .001E100).
3. Underflow or overflow in conversion to floating-point form.
4. Missing operator after the constant.

Alphanumeric

1. Missing quotation mark.
2. Single (or double) closing quotation mark does not match double (or single) opening quotation mark.
3. More than six characters between quotes.
4. Contains a character having an EBCDIC value less than that of the blank character (for example, the O character).

xxxx BAD FORMAT

This message covers a wide range of syntax errors. The user should reexamine statement definition if the error is not obvious.

xxxxx BAD FORMULA

An arithmetic expression error has been detected. This message covers a wide range of error situations such as missing operators, missing operands, misspelled function names, misspelled keywords, etc.

BAD LINE # AFTER STMT xxxxx

A line number contains a nonnumeric character or more than five digits.

xxxxx BAD STMT

The type of statement is not recognizable; most frequently, the command keyword has been misspelled.

BAD STMT ORDER AT xxxxx

Not in CP-V. A line number is out of sequence.

xxxxx BAD STREAM NO.

An I/O stream number is outside of the legal range (1-4).

xxxxx BAD SUBSCR

A known subscript value is too small.

BAD $UBSTRING PARAM

CP-V only. Run-time error. A substring index is nonpositive or starts beyond the maximum string length.

xxxxx BAD TEXT

A text string either contains a New Line character (user probably forgot the end-quote mark), or has an unmatching quote (as in 123 PRINT "DOUBLE QUOTE'), or contains a character having an EBCDIC value lower than 15.

BAD TEXT STRING

Not in CP-V. A test string contains a character having an EBCDIC value less than 15.

CANNOT OPEN

Unable to OPEN a file.

DATA MIX-UP, $STRING VS NUMERIC

CP-V only. Either numeric data is being input to a string, or text exceeding aconst length is being input to a simple or subscripted variable (via READ, INPUT, GET, MAT READ, MAT INPUT, or MAT GET).

xxxxx DEFD TWICE

A function defined in DEF statement xxxxx was also defined by an earlier DEF statement.

xxxxx DIM ERR

A DIM statement formula contains one of the following:

1. User function call.
2. Simple variable that is not SET to some value.
3. Subscripted variable reference.

3. Subscripted variable reference.

xxxxx DIMD TWICE x

Multiple-dimensioning has been attempted. Revise DIM statements.

DIM TOO BIG

Run-time error. A dimension is too large in a matrix operation.

xxxxx DIV BY ZERO

Run-time or compile error. A zero denominator was encountered in expression evaluation.

xxxxx ERROR IN KEYED I/O

Reference to illegal key or attempt to access an unkeyed file in the keyed mode.

-EXEC- HALT

CP-V only. Not an error condition. Execution has reached the halt point (second line number) set by an EXECUTE n1-n2 statement.

EXEC TIME LIMIT

The execution time limit specified in a RUN or FAST command was exceeded.

xxxxx EXP OVERFL

Floating-point overflow during exponentiation.

xxxxx EXTRA COMMA

Error indirectly associated with bad comma. Examples:

Syntax	Explanation
X + (Y,Y)	Array reference without array designator.
SIN (A, B)	Too many arguments in intrinsic function.
M(X,Y,Z)	Too many subscripts in array reference.

xxxxx EXTRA INPUT

Contents of input record not exhausted when check symbol '*' encountered.

xxxxx FILE I/O ERROR

Monitor indication of error in attempting to write or read on file.

xxxxx FILE NOT OPEN IN PROPER MODE

I/O operation attempted on file which is closed, or open in a conflicting mode, on the I/O stream in use.

xxxxx FOR-NEXT ERR

Message covers FOR-NEXT errors illustrated below:

Case 1. Wrong Variable Reference

```
FOR A
NEXT B              error
```

Case 2. Improperly Sequenced Statements

```
FOR I
FOR J
NEXT I              error
NEXT J              error
```

Case 3. No Corresponding FOR Statements

```
NEXT A              error
```

These messages may be compounded as in

```
FOR A
FOR B
FOR C
NEXT B              error, FOR C is cancelled
NEXT C              error, FOR B is cancelled
NEXT A
```

or alternatively

```
FOR A
FOR B
FOR C
NEXT B              error
NEXT C              error
.
.
.
END                 error, MISSING NEXTSTMT
```

xxxxx HALT

Normal message at termination of run.

xxxxx HAS BAD LINE NO.

A GOTO, GOSUB, IF, ON, PRINTUSING, or RESTORE contains a line number having more than five digits.

xxxxx HSN-HCS OVERFL)

CP-V only. Hyperbolic sine or hyperbolic cosine overflow.

ILLEGAL

Illegal editorial type-in. Variety of possible causes. This may be a misspelled edit command, aconst too long, illegal PROCEED, bad format on RENUMBER, etc.

ILLEGAL FILE ID

File name, password, or account identifier too long.

ILLEGAL INPUT

Illegal input in off-line mode. (See RETYPE message, below, for on-line mode.)

ILLEGAL INPUT FROM FILE

Illegal input from a file in on-line or off-line mode.

xxxxx INCOMPAT DIMS

Dimensions not compatible in matrix operation. Examples: matrix identity or inversion on nonsquare matrix, wrong dimensioning for matrix multiplication or addition, etc.

xxxxx INPUT DATA LOST

CP-V only. An input record is too big or has a parity error and has been discarded.

xxxxx KEY NOT FOUND

A keyed read was attempted on a file not containing a record with the specified key.

xxxxx LINE # ERR

An illegal line number (>99999) occurred immediately after the statement at xxxxx. If the first line is incorrect, xxxxx is zero.

xxxxx LINE TOO LONG

A RENUMBER operation has created a source line too long for input/output, or more than 85 characters in a line.

xxxxx LOG OF NON-POS ARG

The argument in a logarithmic operation is not greater than zero.

xxxxx MISSING ARRAY x

Not in CP-V. An array was not allocated by the "originating" BASIC program. (Occurs only in direct-execution mode.)

MISSING NEXTSTMT

At least one FOR statement occurred without a matching NEXT statement; that is, there were more FORs than NEXTs.

MISSING STEPS

No list of line numbers was found in a GOTO. . .ON or ON. . .GOTO statement.

MTRX/VEC x

Not in CP-V. (see ARRAY CLASS CONFLICT" for CP-V). This message indicates conflicting use of array x.

xxxxx NEG BASE TO NON-INTEGER POWER

Fractional exponentiation was indicated but the number is negative.

NO DIMSTMT ARRAY x

An array is used in MAT statement or (CP-V) string statements, but not dimensioned.

NO FILE SPACE

User's allocation or public storage exhausted. User can delete any unwanted files and continue.

NON-EXISTENT LINE #

In the execution mode, a direct statement references a line number that is not in the compiled program.

xxxxx NON-NUMERIC VAL

CP-V only. A VAL function argument (string expression) does not represent a number.

xxxxx NON-POS DIM

Run-time error. A zero or negative dimension was encountered in a matrix operation.

NO PROGRAM

The source program does not exist for specified lines or ranges in editing commands. Example: command was LIST 10-50, but no lines exist with numbers between 10 and 50.

NO STMTS

Source input contained no BASIC statements. (Not used in on-line compilations.)

NOT A DIRECT STMT

In Batch/BTM BASIC the following statements are not allowed for direct execution:

DATA	Image
DEF	NEXT
DIM	ON
FOR	PRINTUSING
GOTO...ON	

CP-V BASIC does not allow

DATA	Image
DEF	NEXT
FOR	

OLD FILE

SAVE ON was attempted on an existing closed file.

xxxxx OUT OF DATA

Not enough data for INPUT, READ, or GET statements in the off-line mode.

xxxxx OUT OF RANGE REF. TO ARRAY x

A matrix operation compiled in the "safe" mode has referenced an index greater than the maximum dimension for array x.

xxxxx OVERFLOW

Floating-point overflow.

xxxxx PAREN ERR

This message indicates a parenthesis imbalance.

POWER OVERFLOW

Overflow in exponentiation.

xxxxx PROG TOO BIG

The program exceeds available memory. (Under CP-V, try FILE PACK and possibly CLEAR arrays and strings to recover wasted space.)

RECOMP FOR DIR ST ABORTED

CP-V only. An error was encountered during recompilation for a dependent direct statement or EXECUTE n1-n2. Recompilation is aborted.

xxxxx RECORD NOT IN -GET-FORMAT

An attempt to GET has encountered a record not in the proper format (see Appendix E).

xxxx RESTORE A NON-DATA LINE

A RESTORE statement indicates a line other than a DATA line.

xxxxx RESTORE A NON-EXISTENT LINE

A RESTORE statement indicates a nonexistent line number.

xxxxx RETURN BEFORE GOSUB

A RETURN statement was reached but the return stack is empty. Indicates improper nesting or branching on GOSUB-RETURNs.

RETYPE

An input error was encountered in the on-line mode. The arrow points to the character at which the error was noted. Execution is not aborted. Retype input starting at beginning of the erroneous datum. Do not type a carriage return or line feed before retyping.

xxxxx RUN INTERRUPTED

CP-V only. This message is issued after a "break" (not an error condition).

xxxxx SEC-CSC OVERFL

CP-V only. Secant or cosecant operation overflow.

xxxxx SHOULD BE DATA STMT

Line xxxxx was referenced in a READ or RESTORE statement but was not a recognizable DATA statement.

xxxxx SHOULD BE IMAGE STMT

Line xxxxx was referenced in a PRINTUSING statement but was not a recognizable Image statement.

xxxxx SINGULAR MATRIX

An inversion or simultaneous equation solution was attempted on a singular matrix.

xxxxx SQR ROOT OF NEG ARG

The argument of a square root function is negative.

xxxxx $TRING EXPR ERR

CP-V only. An incorrectly formatted string expression has been detected.

```
xxxxx TOO MANY GOSUBS BEFORE A RETURN
```

The return stack for GOSUB-RETURN logic is full and a GOSUB has been encountered.

```
xxxxx UNABLE TO OPEN xxxxx
```

An attempt to open the named file failed. The file is probably not present or has another account number or name.

```
xxxxx UNDEF FNx
```

No DEF statement appeared for user function FNx.

```
xxxxx ZERO TO NEG POWER
```

An exponentiation operation attempted to raise zero to a negative power.

APPENDIX A. SUMMARY OF BASIC STATEMENTS

The complete set of BASIC statements is shown below. Capital letters indicate syntax that is required as shown. Lowercase letters designate general items. Command parameters enclosed by braces ({ }) indicate a required choice. Parameters enclosed by brackets ([]) are optional. Ellipsis marks (...) denote multiple occurrences of the preceding bracketed parameter. Unless otherwise noted, "variable" means either a simple or a subscripted variable. If the initial line number of a statement is enclosed by brackets, the statement may be executed directly in the on-line mode of operation.

<u>Statement</u>

line: [#s and/or characters to 132 maximum]

$$\text{[line] CHAIN xname} \begin{bmatrix} \text{;password} \\ \text{;password:acct} \\ \text{:acct} \\ \text{:acct;password} \end{bmatrix}$$

$$^\dagger\text{[line] CHAIN LINK xname} \begin{bmatrix} \text{;password} \\ \text{;password:acct} \\ \text{:acct} \\ \text{:acct;password} \end{bmatrix}$$

$$\text{[line] CHANGE} \left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{string} \\ \text{xtrep} \end{array} \right\} \text{TO letter} \\ \text{letter TO string} \end{array} \right\}$$

$$\text{[line] CLOSE} \left\{ \begin{array}{l} \text{:stream} \\ \left\{ \begin{array}{l} \text{I} \\ \text{O} \end{array} \right\} \text{[characters to end of line]} \end{array} \right\}$$

$$\text{line DATA} \begin{bmatrix} [\pm]\text{constant} \\ \text{aconst} \\ \text{tstring}^\dagger \end{bmatrix} \left[, \begin{bmatrix} [\pm]\text{constant} \\ \text{aconst} \\ \text{tstring} \end{bmatrix} \right] \ldots$$

line DEF FN letter (simple variable[,simple variable]...) = expression

$$\text{[line] DIM} \left\{ \begin{array}{l} \text{letter\$}^\dagger \\ \text{letter} \end{array} \right\} \text{(dimx[,dimx])} \left[, \left\{ \begin{array}{l} \text{letter\$}^\dagger \\ \text{letter} \end{array} \right\} \text{(dimx[,dimx])} \right] \ldots$$

[line] END

$$\text{[line] ENDFILE:stream,} \left\{ \begin{array}{l} \text{line} \\ \text{E} \end{array} \right\}$$

line FOR simple variable = expression TO expression [STEP expression]

$$\text{[line] GET} \left[\text{:stream[;key],} \right] \left\{ \begin{array}{l} \text{variable} \\ \text{string}^\dagger \end{array} \right\} \left[, \left\{ \begin{array}{l} \text{variable} \\ \text{string}^\dagger \end{array} \right\} \right] \ldots$$

[line] GOSUB line

[line] GOTO line

†[line] GOTO line [,line]... ON expression

$$\text{[line] IF} \left\{ \begin{array}{l} \text{expr} \\ \text{aconst} \\ \text{string}^\dagger \end{array} \right\} \text{operator} \left\{ \begin{array}{l} \text{expr} \\ \text{aconst} \\ \text{strexp}^\dagger \\ \text{xstrexp}^\dagger \end{array} \right\} \left\{ \begin{array}{l} \text{THEN} \\ \text{GOTO} \end{array} \right\} \text{line}$$

†CP-V only.

[line] INPUT [:stream[;key],] $\begin{Bmatrix}\text{variable}\\ \text{string}^{\dagger}\end{Bmatrix}\left[,\begin{Bmatrix}\text{variable}\\ \text{string}^{\dagger}\end{Bmatrix}\right]\ldots$

[line] INPUT = $\begin{Bmatrix}\text{\$}\\ \text{any other character}\end{Bmatrix}$

[line] [LET] $\left\{\begin{array}{l}\text{variable[,variable]}\ldots = \begin{Bmatrix}\text{expression}\\ \text{aconst}\\ \text{xstrexp}^{\dagger}\end{Bmatrix}\\ \text{string = strexp}^{\dagger}\end{array}\right\}\left[,\left\{\begin{array}{l}\text{variable[,variable]}\ldots = \begin{Bmatrix}\text{expression}\\ \text{aconst}\\ \text{xstrexp}^{\dagger}\end{Bmatrix}\\ \text{string = strexp}^{\dagger}\end{array}\right\}\right]$

[line] MAT aname = (expression)*aname

[line] MAT aname = aname

[line] MAT aname = aname $\{\pm\}$ aname

[line] MAT aname = aname * aname

[line] MAT aname = CON[(dimx[,dimx])]

[line] MAT aname = IDN $\begin{bmatrix}\text{(dimx)}\\ \text{(dimx), any characters to end of line}\end{bmatrix}$

[line] MAT aname = INV (aname[,simple variable])

[line] MAT aname = SIM (aname[,simple variable])

[line] MAT aname = TRN (aname)

[line] MAT aname = ZER[(dimx[,dimx])]

[line] MAT GET [:stream[;key],] adescr[,adescr] . . .

[line] MAT INPUT [:stream[;key],] adescr[,adescr]. . .

[line] MAT PRINT [:stream[;key],] aname $\left[\begin{Bmatrix},\\ ;\end{Bmatrix}\text{aname}\right]\ldots\begin{bmatrix},\\ ;\end{bmatrix}$

[line] MAT PUT [:stream[;key],] adescr[,adescr] . . .

[line] MAT READ adescr[,adescr] . . .

[line] MAT SIZE aname(dimx[,dimx]) [,aname(dimx[,dimx])] . . .

line NEXT simple variable

†[line] ON expression $\begin{Bmatrix}\text{GOTO}\\ \text{THEN}\end{Bmatrix}$ line[,line] . . .

[line] OPEN fileid [,] $\left\{\begin{array}{l}\text{I}\\ \text{O}\\ \text{TO :stream,}\left\{\begin{array}{l}\begin{Bmatrix}\text{GET}\\ \text{INPUT}\end{Bmatrix}\ [,]\ [\text{UPDATE}]\\ \begin{Bmatrix}\text{PUT}\\ \text{PRINT}\end{Bmatrix}\ [,]\ \begin{Bmatrix}\text{ON}\\ \text{OVER}\end{Bmatrix}\end{array}\right\}\ [[,]\ \text{TFILE}]\end{array}\right\}$

†CP-V only.

[line] PAGE

[line] PAUSE

$$[\text{line}]\left\{\begin{array}{l}\text{PRINT}\\ ;\end{array}\right\}\ [\text{:stream}[\text{;key}],]\ \left[\begin{array}{l}, \\ ; \\ \text{text string}\\ \text{expression}\\ \text{xstrexp}^{\dagger}\end{array}\right]\ \left[\begin{array}{l} ,\\ ;\\ \text{text string } [\text{xstrexp}]\\ \left\{\begin{array}{l},\\ ;\end{array}\right\}\ \text{xstrexp}\left[\begin{array}{l}\text{expression}\\ \text{xstrexp}\end{array}\right]^{\dagger\dagger}\\ \left\{\begin{array}{l},\\ ;\end{array}\right\}\text{expression}\end{array}\right]\ \ldots$$

$$[\text{line}]\left\{\begin{array}{l}\text{PRINT}\\ ;\end{array}\right\}\ [\text{:stream}[\text{;key}],]\ \text{USING line}\ \left[,\left\{\begin{array}{l}\text{xstrexp}^{\dagger}\\ \text{expression}\\ \text{text string}\end{array}\right\}\right]\ldots$$

$$[\text{line}]\ \text{PUT}\ [\text{:stream}[\text{;key}],]\ \left\{\begin{array}{l}\text{expression}\\ \text{aconst}\\ \text{xstrexp}^{\dagger}\end{array}\right\}\ \left[,\left\{\begin{array}{l}\text{expression}\\ \text{aconst}\\ \text{xstrexp}^{\dagger}\end{array}\right\}\right]\ldots$$

$$[\text{line}]\left\{\begin{array}{l}*\\ \text{REM}\end{array}\right\}[\text{characters to end of line}]$$

$$[\text{line}]\ \text{READ}\left\{\begin{array}{l}\text{variable}\\ \text{string}^{\dagger}\end{array}\right\}\ \left[,\left\{\begin{array}{l}\text{variable}\\ \text{string}^{\dagger}\end{array}\right\}\right]\ldots$$

[line] RESTORE[line]

[line] RETURN

[line] STOP

† CP-V only.

†† Must not begin with a '+'.

APPENDIX B. SUMMARY OF BASIC COMMANDS

The BASIC commands are shown below. Capital letters indicate syntax that is required as shown. Lowercase letters designate generic items. Command parameters enclosed by braces ({ }) indicate a required choice. Parameters enclosed by brackets ([ ]) are optional. Ellipsis marks (...) denote multiple occurrences of the preceding bracketed parameter.

Command

ACC[OUNT] [name]

BAS[IC]†

BYE†

CAT[ALOG]†

$$\text{CLE[AR]}\begin{bmatrix}\text{ARR[AYS]}\\ \text{STR[INGS]}\end{bmatrix}^{\dagger}$$

$$\text{DEL[ETE]}\begin{Bmatrix}\text{line}\\ \text{line}_1\text{ - line}_2\end{Bmatrix}\left[,\begin{Bmatrix}\text{line}\\ \text{line}_3\text{ - line}_4\end{Bmatrix}\right]\ldots$$

DEL[ETE]xname

ENT[ER BASIC] [L]

EXE[CUTE]† line_1[-line_2]

$$\text{EXT[RACT]}\begin{bmatrix}\text{line}\\ \text{line}_1\text{ - line}_2\end{bmatrix}\left[,\begin{Bmatrix}\text{line}\\ \text{line}_3\text{ - line}_4\end{Bmatrix}\right]\ldots$$

FAS[T][time]

$$\text{FIL[E]}\Big[\text{PAC[K]}\Big]^{\dagger}$$

$$\text{LIS[T]}\begin{bmatrix}\text{line}\\ \text{line}_1\text{ - line}_2\end{bmatrix}\left[,\begin{Bmatrix}\text{line}\\ \text{line}_3\text{ - line}_4\end{Bmatrix}\right]\ldots$$

LOA[D] [xname]

NAM[E] [xname]

$$\text{NUL[L]}^{\dagger}\begin{bmatrix}\text{ARR[AYS]}\\ \text{STR[INGS]}\\ \text{SIM[VARS]}\end{bmatrix}$$

OFF†

PAS[SWORD][name]

PRO[CEED]

$$\text{REN[UMBER]}\Big[\text{line}_1\text{[,line}_2\text{[,incr]]}\Big]$$

†CP-V BASIC only.

Command

RUN[time]

$$\text{SAV[E O]}\begin{Bmatrix}\text{N}\\ \text{VER}\end{Bmatrix}\text{xname}\begin{bmatrix}\text{line}\\ \text{line}_1 - \text{line}_2\end{bmatrix}\left[,\begin{Bmatrix}\text{line}\\ \text{line}_3 - \text{line}_4\end{Bmatrix}\right]\ldots$$

$$\text{SET}\begin{Bmatrix}\text{\$ = positive integer}^{\dagger}\\ \text{letter = digits [,letter = digits]} \ldots\end{Bmatrix}$$

STA[TUS]

SYS[TEM]

WEA[VE]† [xname]

WID[TH]digits

†CP-V BASIC only.

APPENDIX C. BASIC INTRINSIC FUNCTIONS

Function	Result
SIN(arg)	Calculates sine of argument in radians.
COS(arg)	Calculates cosine of argument in radians.
TAN(arg)	Calculates tangent of argument in radians.
ATN(arg)	Calculates arctangent of unitless argument in radians.
EXP(arg)	Calculates exponential functions, that is $e^{(\text{argument})}$.
ABS(arg)	Calculates absolute value of argument.
LOG(arg)	Calculates natural logarithm (base e) of the argument.
LGT(arg)	Calculates common logarithm (base 10) of the argument.
SQR(arg)	Calculates square root of argument.
INT(arg)	Acquires the integer part of the argument, that is, the greatest integer that is less than or equal to the argument.
SGN(arg)	Identifies algebraic sign of argument, and produces a -1 for negative arguments, a 0 for 0, and a +1 for positive arguments.
RND(arg)	Produces, for each call, the next element of a sequence of uniformly distributed random numbers that are greater than 0 but less than 1. If arg is 0 for the first RND call of a program, the identical sequence of random numbers will be generated if the program is rerun and arg is not changed. Otherwise, an unrepeatable sequence will be generated.
DAY(arg)	Supplies the calendar day. If the argument is 0, the BTM output form is mm/dd (as in 03/07 for March 7) and the BPM and CP-V output form is mon/dd (as in MAR 07). If the argument is nonzero, the output form is a floating-point number whose integer part represents the month, and whose fractional part represents the day of the month divided by 100. For example, 3.07E0 represents March 7.
TIM(arg)	Supplies the time of day. If the argument is 0, the output form is hh:mm, as in 15:09. If the argument is nonzero, the output form is a floating-point number whose integer part represents the hour and whose fractional part represents the minutes divided by 60. For example, 15.15E0 represents 3:09 PM.
YER(arg)	Supplies the year. If the argument is 0, the output form is 19yy, as in 1969. If the argument is nonzero, the output form is a floating-point number whose value is equal to the year, as in 1969.0E0.
MAX(arg_n)	Returns the maximum value in the list of arguments.
MIN(arg_n)	Returns the minimum value in the list of arguments.
TAB(arg)	Advances the print device to the column designated by the argument, and should only be used in a PRINT statement. TAB cannot be used to backspace the print device.
PRC(arg)	Specifies the number of significant digits in printed output, and is used only in a PRINT statement. An argument of 0 specifies 6-significant-digit output format, and a nonzero argument specifies 16-significant-digit output.

The following functions are in CP-V BASIC only.

Function	Result
CSC (arg)	Calculates cosecant of an argument in radians. Overflow results in an error message and termination of execution.
SEC(arg)	Calculates secant of an argument in radians. Overflow results in an error message and termination of execution.
COT(arg)	Calculates cotangent of an argument in radians. Overflow results in an error message and termination of execution.
ASN(arg)	Calculates arcsine of a unitless argument, in radians. If the absolute value of the argument is greater than 1.0, an error message is printed and execution is terminated. Resolution of results is restricted to the two quadrants from $-\pi/2$ to $\pi/2$.
ACS(arg)	Calculates the arccosine of a unitless argument, in radians. If the absolute value of the argument is greater than 1.0, an error message is printed and execution is terminated. Resolution of results is restricted to the two quadrants from 0 to π.
HSN(arg)	Calculates hyperbolic sine of an argument. Overflow results in an error message and termination of execution.
HCS(arg)	Calculates hyperbolic cosine of an argument. Overflow results in an error message and termination of execution.
HTN(arg)	Calculates hyperbolic tangent of an argument.
LTW(arg)	Calculates logarithm, base two, of an argument.
DEG(arg)	Converts argument from radians to degrees.
RAD(arg)	Converts argument from degrees to radians.
LEN(strexp)	Returns current number of characters in string expression, as floating-point number.
VAL(strexp)	Returns numeric value of string expression as floating-point value. Error exit if string expression not numeric.
STR(expression [, rstring])	Converts numeric value of expression to string format. Optional rstring argument permits specific formatting. If second argument is not used, standard print output format is used.
KEY(arg)	Returns the value of the key most recently accessed on the I/O stream specified by the argument.

APPENDIX D. SUMMARY OF BASIC OPERATING PROCEDURES

BASIC is always running in one of three modes; edit, compile, or execute. Compile is a transitory mode, leading directly to execution, if successful, or to edit, if failure occurs. In on-line operation, transition from mode to mode may occur frequently because of operator actions.

The variable context of principal interest during BASIC operations includes source text, object code, array and string storage, DATA list pointing control, GOSUB-RETURN status control, printer line width control, output precision control, runfile identification parameters, parametric dimensioning, and maximum string length.

The lists below present a summary of the effects of operator action (command and statement input) on operating mode and context.

GROUP 1

The following commands do not change operating mode and affect context only as the individual command explicitly indicates:

BASIC	No context change.
CATALOG	No context change.
LIST	No context change.
STATUS	No context change.
WIDTH	Set printer width.
ENTER BASIC	Set output precision control.
NAME	Runfile identification.
ACCOUNT	Runfile identification.
PASSWORD	Runfile identification.
SAVE (ON or OVER)	Generates text output file.
FILE (not FILE PACK)	Outputs runfile.
NULL (any option)	Zeros selected context.
DELETE xname	Deletes named file.

GROUP 2

Input of any of the following commands leaves BASIC in edit mode, requiring recompilation or normal compilation of object code before subsequent execution. If a Group 2 command is input while in execution mode, DATA list pointing control and GOSUB-RETURN status are sav
Other context is modified as indicated.

DELETE	Removes line from active use.
EXTRACT	Removes program except specified line(s).
Line insertion	Adds line to active use.
FILE PACK	Releases storage of deleted lines.
LOAD (while in edit mode) or WEAVE	Acts as series of line insertions.
CLEAR ARRAYS or STRINGS	Releases storage.
SET	Sets parametric dimensioning letter, or maximum string length.

GROUP 3

Input of any of the following commands leaves BASIC in edit mode and requires normal compilation of object code before subsequent execution. Context of DATA list pointing and GOSUB-RETURN control is set to initial conditions. Other context is modified as indicated.

LOAD (while in execute mode)	New text.
CLEAR	Clears arrays, strings, text, and object code.
RENUMBER	Generates output file of text, does CLEAR, and loads numbered text.

GROUP 4

Input of either FAST or RUN initiates normal compilation of object code from the current program text. Compilation is in fast or safety mode as indicated. In addition to object code generation, string-array space is reallocated, but context is saved, as possible, where indicated by new absolute dimensioning.

If errors occur during compilation, messages are generated and object code generation is aborted, as is array-string storage reallocation. Return is in the edit mode, as per Group 3.

If no errors occur, execution is initiated at start of program with DATA list pointing and GOSUB-RETURN control initialized.

DIRECT STATEMENTS

There are three categories of direct statements, in terms of operational impact: DIM statements, statements with included line-number references, and statements with no line-number references.

DIM, as a direct statement, causes updating of storage requirements for arrays and strings and returns control in edit mode. Rules for command Group 2 apply.

Direct statements without line references are compiled and executed independently of other program text or object code. If errors occur, messages are output and execution is aborted. Control is returned to console in same mode as before input of direct statement.

Direct statements with line references require presence of object code. If mode at input is execute, the direct statement is compiled and executed. If mode is edit, recompilation of program is initiated. Compilation is aborted by any error, in which case return is in edit mode. If recompilation is successful, direct statement is compiled and executed. Return to console is in execute mode.

ON-LINE VERIFICATION

On-line verification is a general term for various functions that may be performed in checking out and debugging a program. The following examples are typical:

>GO TO 35	Branch to a desired statement within a program and proceed with execution.
>PRINT A(1,1), A(1,2),A(5,5)	Verify assignment of values in an array.
>LET A1 = 12.5 >GO TO 20	Modify a value then continue at selected point.

Under CP-V, any valid BASIC statement, except for DATA, DEF, Image, FOR, or NEXT, may be typed without a leading line number. This will indicate that the statement should be executed immediately.

In BTM BASIC, the statements DATA, DEF, DIM, Image, FOR, NEXT, GOTO ... ON, and ON ... GOTO, may not be direct.

Under BTM, when in the editing mode, attempted use of the direct statement capability will result in the error message

```
RUN? ILLEGAL
```

DESK CALCULATOR MODE

In CP-V BASIC, the desk calculator mode can be entered. from the editing mode simply by typing direct statements.

In BTM BASIC, the programmer enters the desk calculator mode by giving a CLEAR command followed by RUN or FAST.

The desk-calculator mode uses the computer in the simplest and most direct manner, working without a stored program or stored data. The problem to be solved is usually combined with a PRINT statement, preceded if necessary by a LET or another form of an assign statement. Typical examples of BASIC in the desk-calculator mode are

```
>PRINT  1.085 3.6

>PRINT  SQR (12 * 12 + 15 * 15)

>PRINT  LOG(SIN (5.12))

>PRINT  SQR (87)
```

PROCEED

PROCEED is a command to return to an interrupted activity (see Chapter 4). Three responses are possible: (1) immediate return to interrupt point, (2) recompile and return to "next statement", or (3) indicate ILLEGAL and return to console in edit mode. The response depends on the categories of commands that have been input since the interrupt. In general, PROCEED is legal only if execution of a program has been interrupted; it always terminates the interrupt state. If any Group 3 or 4 commands have been input since BREAK key activation, PROCEED is illegal.

If Group 2 commands have been input since BREAK key activation, and PROCEED is legal, recompilation is initiated. Errors during recompilation cause error message generation, abort object code generation and array-string storage reallocation, and cause return to console in edit mode. If recompilation is successful, execution is initiated at the "next statement" of the interrupted and modified program.

If only Group 1 commands have been input since BREAK key activation, PROCEED causes return to the interrupted activity.

BREAK-PROCEED LOGIC

In Batch/BTM BASIC, BREAK may interrupt execution at any point, but PROCEED resumes execution only if intervening console actions have not modified object code. In CP-V BASIC, only CLEAR, LOAD, or RENUMBER, or failure to correct a diagnosed error prohibit PROCEED from resuming execution following a BREAK.

If a BREAK occurs during execution of a statement, that statement is fully executed and the line number of the next statement to be executed is saved before user is given console control. PROCEED then returns to the saved line number. If that line has been edited out or is not an executable statement, execution resumes at the next executable line. If a BREAK occurs in response to an input statement (after a ? prompt character), execution is halted.

APPENDIX E. FORMAT OF BINARY DATA FILES FOR PUT AND GET OPERATIONS

The PUT and MAT PUT operations in BASIC create data files in the internal format described in Table 3 with a physical record size of 120 bytes.

Table 3. Internal Format of Data Files

Byte	Coding	Meaning
0	X'3C'	Physical record.
1	Checksum	Sum of bytes in record, not counting checksum byte.
2,3	Record size	Number of bytes used (120 or less), including control bytes.
4...n	Data	Either doubleword floating-point or aconst doubleword or both.
n+1	X'3C'	End of physical or logical record.
n+2	X'BD'	
n+3, n+4	Physical record number	In numerical order, from 0.

Normally a record contains 112 noncontrol bytes (14 floating-point values or aconsts). The last record in a file may contain fewer used bytes but still contains 120 total bytes. The control word – bytes n+1 to n+4 – is repeated in this case as bytes 116 to 119.

Figure 7 shows a file containing three records of numeric and aconst data, with the record contents given in hexadecimal format. The values were created with the program shown in Figure 8. In Figure 7, the value 1 occupies words 1 and 2 of record 1000, the aconst ABCDEF occupies words 13 and 14 of record 2000, and the aconst 7890 occupies words 25 and 26 of the same record but followed in word 27 by an end-of-record control word forced there by the flush operation.

```
KEY= X'001000'    -        120 BYTES

00000    3C090078    41100000    00000000    41200000
00004    00000000    41300000    00000000    41400000
00008    00000000    41500000    00000000    41600000
0000C    00000000    41700000    00000000    41800000
00010    00000000    41900000    00000000    00000000
00014    00000000    00000000    00000000    41900000
00018    00000000    41800000    00000000    41700000
0001C    00000000    3CBD0000

KEY= X'002000'    -        120 BYTES

00000    3CA60070    41600000    00000000    41500000
00004    00000000    41400000    00000000    41300000
00008    00000000    41200000    00000000    41100000
0000C    00000000    0001C1C2    C3C4C5C6    0001C7C8
00010    C9D1D2D3    0001D4D5    D6D7D8D9    0001E2E3
00014    E4E5E6E7    0001E8E9    00000000    0001F1F2
00018    F3F4F5F6    0001F7F8    F9F00000    3CBD0001
0001C    00000000    3CBD0001
```

Figure 7. Contents of Sample File

```
KEY= X'003000'     -        120 BYTES

00000    3C030028    41100000    00000000    41200000
00004    00000000    41300000    00000000    41400000
00008    00000000    3CBD0002    00000000    41100000
0000C    00000000    0001C1C2    C3C4C5C6    0001C7C8
00010    C9D1D2D3    0001D4D5    D6D7D8D9    0001E2E3
00014    E4E5E6E7    0001E8E9    00000000    0001F1F2
00018    F3F4F5F6    0001F7F8    F9F00000    3CBD0001
0001C    00000000    3CBD0002
```

Figure 7. Contents of Sample File (cont.)

```
100 OPEN 'PUT',0
110 PUT 1,2,3,4,5,6,7,8,9,0
120 PUT 0,9,8,7,6,5,4,3,2,1
130 PUT 'ABCDEF','GHIJKL','MNOPQR','STUVWX','YZ'
140 PUT '123456','7890'
150 PUT**
160 PUT 1,2,3,4,**
170 CLOSE 0
180 END
```

Figure 8. Program Used to Generate Figure 7

APPENDIX F. I/O CONTROL

Source statements and INPUT (except INPUT :stream) data is read via M:SI. Data output produced by LIST and PRINT (except PRINT :stream) is written via M:SL. These DCBs are normally defaulted to the terminal in the on-line mode.

BASIC will not honor assignments to or from magnetic tape.

If M:SI or M:SL has been assigned by a CP-V SET command (on-line) or !ASSIGN (off-line) the source input or list output will be modified accordingly. It is generally inadvisable to alter these I/O assignments.

Normal DCB assignments are shown in Table 4. An example of using SET to assign a DCB is

```
!SET M:SL/FILE6
```

The above SET command given before calling BASIC causes LIST or PRINT output to be assigned to FILE6.

Since BASIC expects an I/O stream to be explicitly opened to a file specified in an OPEN statement, it is not possible to SET any of the four DCBs used for stream I/O before calling BASIC.

Table 4. Normal DCB Assignments

Monitor	DCB	Definition	Assignment
CP-V & BPM	M:SI	Source Input	Users Console or Card Reader
	M:EI	Stream 1	File
	M:EO	Stream 2	File
	M:CI	Stream 3	File
	M:LO	Stream 4	File
	M:DO	Diagnostic Output	Users Console or Line Printer
	M:SO	Source Output (Save, File, Load, Renumber)	File
	M:SL	List/Print Output	Users Console or Line Printer
BTM	M:EO	Stream 1	File
	M:EI	Stream 2	File
	M:CI	Stream 3	File
	M:SO	Source Output (Save, File, Load, Renumber)	File

APPENDIX G. BASIC CONCORDANCE PROGRAM

The BASIC Concordance program takes any BASIC source program and produces a listing of the following items along with the line numbers on which they are used:

1. Line number references
2. User-defined functions
3. Arrays
4. Strings
5. Simple variables

For each use of one of these items, the concordance output will contain an entry of the line number of the use. Thus, multiple appearances of a line number may occur for a single item in the output.

The program does not discriminate between string scalars and string arrays, but rather classifies both as strings. (Note that the same letter cannot be used as both a string scalar and a string array, thus no confusion can result.) The program does discriminate between nonstring arrays and simple variables with the same name, since these are two different entities.

Dummy arguments in user-defined function definitions are not displayed, in order to prevent confusion between these dummies and normal simple variables of the same name.

The source language for the Concordance program is Xerox Extended FORTRAN IV. It exists as one program containing the main program and seven internal subprograms. The source deck must be compiled into a relocatable object module (ROM). The ROM must then be linked to form a load module. During the linking process, a record for the DCB F:1 must appear in the assign/merge record for the log-on account. This can be accomplished by using the !SET command (if on-line) or the !ASSIGN command (if in batch) for F:1 prior to the LINK command.

To use the program, the user must set or assign F:1 to indicate the location of the BASIC source program to be used as input. This may be a file on disk or labeled magnetic tape or any input device capable of transmitting a BASIC source program. After setting F:1, the load module created by the linking process is invoked. Output will appear on the default destination for M:LO. If the user wishes to divert the output, he may do so by setting or assigning F:108 prior to invoking the load module.

At the completion of processing, the following terminal message will be output:

```
*STOP*   1
```

The program expects as input a BASIC source program that can be compiled by the Xerox BASIC compiler supplied with either CP-V or BTM with error diagnosis. Certain errors that would be detected by those compilers are also detected by the Concordance program. If such is the case, the output line will be preceded by the message

```
ILLEGAL SYNTAX ON LINE XXX
```

where xxx is the sequential record number of the line in question in the input file.

Certain other syntactic errors that would be diagnosed by the compiler are not detected by the Concordance program and may cause extraneous entries in the output.

If an irrecoverable read error occurs in processing the input, the following message will be output:

```
IRRECOVERABLE READ ERROR ON LINE XXX
```

where xxx is as described above.

Example:

```
!FORT4 SOURCE ON BO
OPTIONS> NS
!SET F:1
!LINK BO ON BACON
!SET F:1 PROG
!BACON.
```

The first and second lines of the above example cause compilation of the concordance program from the file SOURCE, previously created by a batch job (see the BPM/BP,RT Reference Manual, 90 09 54). Next, the F:1 DCB is set into the assign/merge record. The load module BACON is then formed by the !LINK command, using the file BO (created by the FORTRAN compilation) as input. The user may then assign F:1 to any BASIC program file (e.g., PROG) by use of a !SET command (on-line) or !ASSIGN command (off-line). The concordance program is then executed by giving the command !BACON. This is followed by a concordance listing such as the one shown below. Note that the linking process need be done only once, but F:1 must be reassigned each time that a new concordance is to be produced.

```
LINE NUMBER REFERENCES

              8:   42  44
             66:    6  54  62

USER DEFINE FUNCTIONS
  NONE

ARRAYS
  NONE

STRINGS
  NONE
```

```
SIMPLE VARIABLES

  A:      12  14  14
  I:       2   8  20  20  20  20
          22  24
  X:      37
  A1:      3   3   5   5   7

*STOP*  1
```

The BASIC Concordance program can be run as an executing Extended FORTRAN IV program in any Xerox operating system that contains the Xerox Extended FORTRAN IV processor.

A source deck (-34 element) and binary relocatable deck (-24 element) of the BASIC concordance are available from the Xerox Software Library (Catalog No. 706292).

APPENDIX H. EBCDIC CHARACTER CODES

EBCDIC Value	Character	Meaning	Remarks
0	NUL	null	0 through 31 are control codes.
1	SOH	start of header	On 2741 terminals, SOH is PRE.
2	STX	start of text	On 2741 terminals, STX is BY.
3	ETX	end of text	On 2741 terminals, ETX is RES.
4	EOT	end of transmission	On 2741 terminals, EOT is ATTN.
5	HT	horizontal tab	0,6,7,9-11 and 14-15 are idles for 2741 terminals.
6	ACK	acknowledge (positive)	
7	BEL	bell	
8	BS or EOM	backspace or end of message	EOM is used only on Xerox Keyboard/
9	ENQ	enquiry	Printers Models 7012, 7020, 8091,
10	NAK	negative acknowledge	and 8092.
11	VT	vertical tab	
12	FF	form feed	
13	CR	carriage return	CR outputs CR and LF.
14	SO	shift out	
15	SI	shift in	
16	DLE	data link escape	
17	DC1	device control 1	On Teletype terminals, DC1 is X-ON.
18	DC2	device control 2	On 2741 terminals, DC2 is PN.
19	DC3	device control 3	DC3 is RS on 2741s, X-OFF on Teletypes.
20	DC4	device control 4	On 2741 terminals, DC4 is PF.
21	LF or NL	line feed or new line	LF outputs CR and LF.
22	SYN	sync	
23	ETB	end of transmission block	On 2741 terminals, ETB is EOB.
24	CAN	cancel	
25	EM	end of medium	
26	SUB	substitute	Replaces characters with parity error.
27	ESC	escape	
28	FS	file separator	16, 17, 22, 24, 25, and 27-30 are idles for
29	GS	group separator	2741 terminals.
30	RS	record separator	
31	US	unit separator	
32	LF only	line feed only	32 through 47 are used for output only.
33	FS		These codes are duplicates of the label
34	GS		entries that caused activation. These
35	RS		entries output a single code only and
36	US		are not affected by any special func-
37	EM		tional processing
38	/		
39	↑		
40	=		
41	CR only	carriage return only	
42	EOT		
43	BS		
44	)		
45	HT	tab code only	
46	LF only	line feed only	
47	SUB		
48	ESC F	end of file	48 through 63 cause special functions
49	CANCEL	delete all input and output	to be performed.
50	ESC X	delete input line	

EBCDIC Value	Character	Meaning	Remarks
51	ESC P	toggle half-duplex paper tape mode	
52	ESC U	toggle restrict upper case	
53	ESC (	upper case shift	
54	ESC)	lower case shift	
55	ESC T	toggle tab simulation mode	
56	ESC S	toggle space insertion mode	
57	ESC E	toggle echo mode	
58	ESC C	toggle tab relative mode	
59	ESC LF	line continuation	59 toggles the backspace edit mode for 2741 terminals.
60	X-ON	start paper tape	
61	X-OFF	stop paper tape	
62	ESC R	retype	
63	ESC CR	line continuation	
64	SP	blank	
65			
66			
67			
68			
69			
70			
71			
72			
73			
74	¢ or `	cent or accent grave	Accent grave used for left single quote. On Model 7670, ` not available, and ¢ = 180.
75	.	period	
76	<	less than	
77	(	left parenthesis	
78	+	plus	
79	| or ¦	vertical or broken bar	On Model 7670, ¦ not available, and ! = 90.
80	&	ampersand	
81			
82			
83			
84			
85			
86			
87			
88			
89			
90	!	exclamation point	On Model 7670, ! is |.
91	$	dollars	
92	*	asterisk	
93	)	right parenthesis	
94	;	semicolon	
95	~ or ¬	tilde or logical not	On Model 7670, ~ is not available, and ¬ = 106.
96	-	minus, dash, hyphen	
97	/	slash	
98			
99			
100			
101			
102			
103			
104			
105			

EBCDIC Value	Character	Meaning	Remarks
106		circumflex	On Model 7670 ^ is ¬. On Model 7015 ^ is ∧ (caret).
107	,	comma	
108	%	percent	
109	_	underline	Underline is sometimes called "break character"; may be printed along bottom of character line.
110	>	greater than	
111	?	question mark	
112			
113			
114			
115			
116			
117			
118			
119			
120			
121			
122	:	colon	
123	#	number	
124	@	at	
125	'	apostrophe (single quote)	
126	=	equals	
127	"	quotation mark	
128			
129	a		
130	b		
131	c		
132	d		
133	e		
134	f		
135	g		
136	h		
137	i		
138			
139			
140			
141			
142			
143			
144			
145	j		
146	k		
147	l		
148	m		
149	n		
150	o		
151	p		
152	q		
153	r		
154			
155			
156			
157			
158			
159			
160			
161			

EBCDIC Value	Character	Meaning	Remarks
162	s		
163	t		
164	u		
165	v		
166	w		
167	x		
168	y		
169	z		
170			
171			
172			
173			
174			
175	|	logical and	
176			
177	\	backslash	
178	{	left brace	On 2741 terminals, { is output as (.
179	}	right brace	On 2741 terminals, } is output as).
180	[	left bracket	On Model 7670, [is ¢. On Model 7015, [is |.
181	]	right bracket	On Model 7670,] is |. On Model 7015,] is ¬.
182			
183			
184			
185			
186			
187			
188	[	left bracket	
189	]	right bracket	
190	lost date	lost data	
191	¬	logical not	
192	SP	blank	Output only.
193	A		
194	B		
195	C		
196	D		
197	E		
198	F		
199	G		
200	H		
201	I		
202			
203			
204			
205			
206			
207			
208			
209	J		
210	K		
211	L		
212	M		
213	N		
214	O		
215	P		
216	Q		
217	R		

EBCDIC Value	Character	Meaning	Remarks
218			
219			
220			
221			
222			
223			
224	-	minus	Output only.
225			
226	S		
227	T		
228	U		
229	V		
230	W		
231	X		
232	Y		
233	Z		
234			
235			
236			
237			
238			
239			
240	0		
241	1		
242	2		
243	3		
244	4		
245	5		
246	6		
247	7		
248	8		
249	9		
250			
251			
252			
253			
254			
255			

APPENDIX I. SAMPLE BASIC PROGRAMS

This appendix contains sample programs of varying degrees of complexity, showing a few of the many possible applications of Xerox BASIC. The reader may find it helpful to consider alternative methods or other applications of the techniques used in these programs.

SNOWFLAKE SIMULATION

The program shown in Figure 9 simulates the almost infinite variety of geometric forms assumed by snowflakes. Each "flake" is formed by adding random accretions adjacent to two radii approximately 60 degrees apart in a square matrix. A second sector is formed by "rotating" the first one clockwise by 60 degrees and the pattern is then replicated in the remaining quadrants of the matrix. The contents of the matrix are then printed, with an asterisk representing the value 1. The BREAK key is used to stop the snowstorm.

WORD GUESSING GAME (CP-V)

The program shown in Figure 10 picks a word at random from a previously created file named WORDS. Such a file is easily built using the Edit subsystem; see the Edit BUILD command in the CP-V/TS Reference Manual 90 09 07. The terminal user attempts to guess what word has been chosen. Any letters guessed correctly are printed by the program, until the entire word is typed correctly by the user. It should be easy to think of ways to make the game more interesting.

LOAN INTEREST CALCULATION (CP-V)

This program, shown in Figure 11, calculates monthly payments for loans made at a given rate of interest and repaid over a specified time period. Note that the printing of the initial explanatory information is omitted if the user simply gives a carriage return following the first prompt (see lines 100-140). The user is requested to type values for principal, monthly payment, interest rate, and total number of months. If one of these parameters is entered as zero, its value is calculated from the other three.

CELLULAR AUTOMATA

The program shown in Figure 12 demonstrates the behavior of a type of cellular automata. Such automata are represented by using numeric values to simulate the states of "cells" in a regular geometric pattern such as a grid of unit squares. The automata simulated by this program we devised by mathematician John Horton Conway and described by Martin Gardner in the October, 1970 issue of Scientific American magazine.

In this simulation, a cell is either "alive" or "dead". Living cells are displayed by printing an asterisk and dead cells are shown as blanks. Each cell has a neighborhood comprising 8 contiguous cells (4 orthogonal and 4 diagonal). A dead cell becomes alive only if it has exactly 3 living neighbors, and a living cell remains alive only if it has either 2 or 3 living neighbors. All changes in state are considered to occur simultaneously throughout the entire grid. Each successive configuration is called a "generation".

The terminal user defines the initial configuration by typing the X, Y coordinates of all living cells (e.g., 1,1 2,1 3,1 4,1 5,1 or 1,1 2,1 2,2 2,3 3,2). The program then applies the transition rules for each generation and displays the result. The BREAK key is used to halt the program.

This program stores the X, Y coordinates of all living cells as a "sparse matrix", to reduce array storage requirements, and the display is automatically positioned to minimize the printing of blank lines or columns.

MAGIC SQUARES

The program shown in Figure 13 generates and displays "magic squares" of odd order, by a method attributed to Bachet de Meziriac. Using this method, integers 1 through N**2 are arranged in N diagonal rows. This diamond-shaped pattern is then transformed into a square array by converting row and column coordinates to their residues modulo N. The result is a magic square of order N in which the summation of each of the columns, rows, and main diagonals is equal to (N**3+N)/2.

ARITHMETIC COMBINATIONS

This program, shown in Figure 14, shows the various ways in which positive integers can be added to form a given sum. Note that combinations such as 1+2 and 2+1 are not considered identical, although the program can easily be modified to treat such combinations as being the same.

```
100 DIM A(63,63) & *SET POINT PROBABILITY DENSITY
120 P=RND(1)/2+.1 & MAT A=ZER & *SET FLAKE SIZE
150 FOR R=2*INT(RND(1)*10+1) TO 32 STEP 2 & *BUILD TWO RADII
170 A(R,R/2+16),A(R,48-R/2)=1 & NEXT R & *BUILD ACCRETIONS
200 FOR R=2 TO 30 STEP 2 & FOR C=R/2+2 TO 32 & IF A(R,C-2)=0 THEN 250
230 IF RND(1)<P THEN 250 & A(R,C),A(R,64-C)=1
250 NEXT C & NEXT R & *ROTATE SECTOR 60 DEGREES
280 FOR R=2 TO 30 STEP 2 & FOR C=18+R/2 TO 48-R/2 STEP 2 & IF A(R,C)=0 THEN 320
310 A(R+C-R/2-16,49-R/2+.5*(C-(18+R/2)))=1
320 NEXT C & NEXT R & *REPLICATE QUADRANT
350 FOR R=2 TO 32 STEP 2 & FOR C=32 TO 62 & IF A(R,C)=0 THEN 390
380 A(64-R,C),A(64-R,64-C),A(R,64-C)=1
390 NEXT C & NEXT R & ; & *PRINT FLAKE
430 FOR R=2 TO 62 STEP 2 & F=0 & FOR C=2 TO 62 & IF A(R,C)=0 THEN 490 & F=1
480 ;TAB(C)"*"TAB(0)
490 NEXT C & IF F=0 THEN 520 & ;
520 NEXT R & ; & ; & ; & GOTO 120
```

Figure 9. Snowflake Simulation

```
100 OPEN "WORDS" TO :1, INPUT & ENDFILE:1,130 & INPUT :1;9999.999,K
130 K=KEY(1)
140 INPUT :1;INT(RND(1)*K+1),R$ & R=LEN(R$)
160 ;"I'M THINKING OF A"R"-LETTER WORD." & ;"WHAT DO YOU THINK IT IS";
180 INPUT G$ & IF G$=R$ THEN 340 & G=LEN(G$) & ON SGN(G-R)+2 GOTO 220, 270, 240
220 ;"TOO SHORT,"; & GOTO 250
240 ;"TOO LONG,";
250 ;"TRY AGAIN"; & GOTO 180
270 FOR I=1 TO G & IF G$(:I,1)=R$(:I,1) THEN 300 & G$(:I,1)='-'
300 NEXT I & ;"YOU GOT THESE RIGHT:"; & ;G$ & GOTO 250
340 ;"THAT'S RIGHT!"; & ;"WOULD YOU LIKE TO TRY ANOTHER"; & INPUT A$
370 IF A$(:1,1)="Y" THEN 140 & ;"THANKS FOR PLAYING THE GAME..." & ;
```

Figure 10. Word Guessing Game (CP-V)

```
100 ; & ;'EXPLAIN'; & INPUT A$ & IF A$(:1,1)='Y' THEN 160
140 IF A$(:1,1)='y' THEN 160 & GOTO 310
160 ;'The value of any variable typed as zero will be calculated,'
170 ;'using the other three inputs to obtain the result.  Type all'
180 ;'non-numeric inputs as either YES or NO.  If no variable is'
190 ;'input as zero, a schedule will be printed.  If a yearly'
200 ;'schedule is wanted, you may have the interest calculated for'
210 ;'the tax year or for any other 12-month period.  Respond to'
220 ;'the question "MONTH STARTED" by typing the number (1 to 12)'
230 ;'for the month of the first payment.  To obtain a cumulative'
240 ;'12-month schedule, respond by typing the value 1.' & ;
260 * S=SCHEDULE
270 * P=PRINCIPAL
280 * M=MONTHLY PAYMENT
290 * I=INTEREST PER YEAR (e.g., 6 and 1/2 percent is 6.5)
300 * N=NUMBER OF MONTHS
310 ; & ;'PRINCIPAL='; & INPUT P & ;'MONTHLY PAYMENT='; & INPUT M
320 ;'ANNUAL INTEREST RATE='; & INPUT I & I=INT(I*1E4/12+.5)/1E6
330 ;'NO. OF MONTHS='; & INPUT N & IF P=0 GOTO 570 & IF M=0 GOTO 620
430 IF I=0 GOTO 670 & IF N=0 GOTO 460 & GOTO 830
460 N1 =(LOG(M)-LOG(M-P*I))/LOG(1+I) & N=INT(N1+.99) & ;
490 ;USING 500,N
500 :NO. OF MONTHS = ###
510 I2=M*N1-P & T1=I2+P & ; & ;USING 550,I2,P,T1
550 :TOT INT = #####.##   TOT PRIN = #####.##   TOTAL = ######.##
560 GOTO 770
570 P=M*((1+I)**N-1)/((1+I)**N*I) & ; & ;USING 600,P
600 :PRINCIPAL = $#####.##
610 GOTO 770
620 M=INT((P*(1+I)**N*I/((1+I)**N-1)+.005)*100)/100 & ;
640 ;USING 650,M
650 :PAYMENT = $###.##
660 GOTO 770
670 I=23*(N*M-P)/(N*P)/12 & FOR U=1 TO 5
680 I1=((I+1)**N*I)/((I+1)**N-1)-(M/P)
700 I2=((I+1)**(2*N)-(I+1)**N-I*N*(I+1)**(N-1))/((I+1)**N-1)**2
710 I3=I-I1/I2 & I=I3 & NEXT U & ; & ;USING 760,I*1200
760 :ANNUAL INTEREST = ##.##%
770 ; & ;'SCHEDULE'; & INPUT S$ & IF S$(:1,1)='Y' THEN 830
810 IF S$(:1,1)='y' THEN 830 & GOTO 1410
830 A=1,I2=0,I3=0,L=36,L1=0,L2=1,P1=0,P2=0,P3=0,P8='MONTH',T1=0
840 ;'MONTH STARTED='; & INPUT C & U=13-C & ;'PRINTED MONTHLY';
880 INPUT S$ & IF S$(:1,1)='Y' THEN 920 & IF S$(:1,1)='y' THEN 920
910 P8=' YEAR',C=1
920 ; & ; & ;USING 600,P & ;USING 650,M & ;USING 760,I*1200
970 ;USING 500,N & ;
990 I1=INT((P*I+.005)*100)/100 & P4=I1+P & IF P>M GOTO 1030 & M=P4
1030 B=P4-M & I2=I1+I2,I3=I1+I3 & P1=M-I1,P2=P1+P2,P3=P1+P3
1060 T1=T1+M & IF L1=0 GOTO 1120 & IF L1<=L GOTO 1170 & L=48 & ; & ;
1120 ; & ;USING 1140,P8
1140 :#####    INT      PRIN     TOT INT   TOT PRIN     TOTAL    BALANCE
1150 L1=1 & ;
1170 IF A=N GOTO 1360 & IF B=0 GOTO 1360 & IF S$(:1,1)='Y' THEN 1240
1200 IF S$(:1,1)='y' THEN 1240 & IF A<U GOTO 1340 & U=U+12
1230 GOTO 1310
1240 IF L2=1 GOTO 1280 & IF A<=U GOTO 1310 & U=U+12,C=1,I2=I1,P2=P1
1270 ;
1280 ;'YEAR'L2 & ; & L2=L2+1
1310 ;USING 1320,C,I3,P3,I2,P2,T1,B
1320 :###  ####.##  ####.##  #####.##  #####.##  #####.##  #####.##
1330 C=C+1,L1=L1+1,I3=0,P3=0
1340 P=B,A=A+1 & GOTO 990
1360 ;USING 1320,C,I3,P3,I2,P2,T1,B & IF B>0 GOTO 1410 & ;
1390 ;USING 1400,M
1400 :LAST PAYMENT = $###.##
1410 END
```

Figure 11. Loan Interest Calculation (CP-V)

```
100 DIM X(1000), Y(1000), S(1000) & B1,B2,B3=0 & GOSUB 1170
130 F,F1=0 & ;'How many generations per display'TAB(0) & INPUT F2
160 ;'How many points will you define'TAB(0) & INPUT N1 & N2=N1
190 ;'Enter X,Y coordinates for'N1' points' & FOR A=1 TO N1
210 INPUT X(A),Y(A) & S(A)=1 & NEXT A & B3=N1, F=F+1, F1=F1+1
250 GOSUB 800
260 FOR J=1 TO N1 & X=X(J), Y=Y(J) & GOSUB 610
290 IF K>3 THEN 330 & IF K<2 THEN 330 & B1=B1+1 & GOTO 340
330 S(J)=0, B2=B2+1
340 FOR M=1 TO 3 & FOR N=1 TO 3 & IF M*N=4 THEN 410
370 X=X(J)+M-2, Y=Y(J)+N-2 & GOSUB 610
390 IF K<>3 THEN 410 & GOSUB 730
410 NEXT N & NEXT M & NEXT J & J=1 & FOR I=1 TO N2
460 IF S(I)=0 THEN 480 & S(J)=1, X(J)=X(I), Y(J)=Y(I), J=J+1
480 NEXT I & N1,N2=J-1, F=F+1, F1=F1+1 & IF F1<F2 THEN 530
510 GOSUB 800
520 F1=0
530 B1,B2,B3=0 & IF Q=0 THEN 560 & GOTO 260
560 ;'All cells are empty!'
570 ;'Do you want to define a new configuration'TAB(0) & INPUT C9
590 IF C9='YES' THEN 130 & STOP
610 K=0 & FOR I=1 TO N1 & X2=X(I), Y2=Y(I) & IF Y2<>Y THEN 660
650 IF X2=X THEN 710
660 IF X2<X-1 THEN 710 & IF X2>X+1 THEN 710 & IF Y2<Y-1 THEN 710
690 IF Y2>Y+1 THEN 710 & K=K+1
710 NEXT I & RETURN
730 FOR Z=1 TO N2 & IF X<>X(Z) THEN 770 & IF Y<>Y(Z) THEN 770
760 GOTO 790
770 NEXT Z & N2=N2+1, Y(N2)=Y, X(N2)=X, S(N2)=1, B3=B3+1
790 RETURN
800 R1,C1=1000, R2,C2=-R1 & ; & ; & ;'Generation'F & FOR I=1 TO N1
850 R=Y(I), C=X(I) & IF R>R1 THEN 880 & R1=R
880 IF R<R2 THEN 900 & R2=R
900 IF C>C1 THEN 920 & C1=C
920 IF C<C2 THEN 940 & C2=C
940 NEXT I & C3=2 & IF (C2-C1)<35 THEN 990 & C3=1
930 ;'Horizontal compression in display!'
990 IF (C2-C1)<70 THEN 1020 & C4=(C1+C2)/2, C1=C4-35, C2=C4+35
1010 ;'Horizontal display overflow!'
1020 FOR R=R1 TO R2 & FOR C=C1 TO C2 & FOR I=1 TO N1
1050 IF Y(I)<>R THEN 1080 & IF X(I)<>C THEN 1080
1070 ;TAB((C-C1+1)*C3)'*'TAB(0)
1080 NEXT I & NEXT C & ; & NEXT R & Q=B1+B3
1130 ;USING 1140, B1,B2,B3,B1+B3
1140 :### Survived, ### Died, ### Born, ### Total
1150 B1,B2,B3=0 & RETURN
1170 ; & RETURN
```

Figure 12. Cellular Automata

```
100 DIM A(15,15) & ;'TYPE AN ODD NUMBER FROM 3 TO 15';
120 INPUT N & IF N<3 THEN 380 & IF N>15 THEN 120
150 IF INT(N/2)=N/2 THEN 120 & R0=INT(N+1+N/2), C0=INT(N-N/2)
170 FOR R=1 TO N & FOR C=1 TO N & R1=R0+R+N-C, R1=R1-INT(R1/N)*N
200 C1=C0+R+C, C1=C1-INT(C1/N)*N & IF R1>0 THEN 230 & R1=N
230 IF C1>0 THEN 250 & C1=N
250 A(R1,C1)=C+R*N-N & NEXT C & NEXT R & ; & FOR I=1 TO N
300 FOR J=1 TO N & ;TAB(4*J),A(I,J)''TAB(0) & NEXT J & ; & ;
350 NEXT I & ; & GOTO 120
380 END
```

Figure 13. Magic Squares

```
100 DIM K(15) & MAT K=ZER
120 ;'THIS PROGRAM LISTS THE WAYS IN WHICH POSITIVE INTEGERS CAN BE'
130 ;'SUMMED TO OBTAIN ANY INTEGER FROM 2 THROUGH 15'
140 ;'PLEASE TYPE AN INTEGER FROM 2 TO 15'; & INPUT N & ;
170 K(1)=N, L=2, M=0
180 IF K(1)=1 THEN 220 & K(1)=K(1)-1, K(2)=K(2)+1 & GOSUB 340
210 GOTO 180
220 FOR J=2 TO L & IF K(J)<>1 THEN 260 & NEXT J & GOTO 310
260 K(1)=K(J)-1, K(J)=1, K(J+1)=K(J+1)+1 & IF J<L THEN 290 & L=L+1
290 GOSUB 340
300 GOTO 180
310 ; & ;'TOTAL NUMBER OF WAYS TO SUM'N' IS'2**(N-1)-1 & END
340 *SUBR TO PRINT SUMMATION
350 FOR I=1 TO L & ;K(I)''TAB(0) & IF I=L THEN 400 & ;' +'TAB(0)
390 NEXT I
400 ;' ='N & RETURN
```

Figure 14. Arithmetic Combinations

INDEX

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

22227, 5C1178, Printed in U.S.A.

XK14, Rev. 0