BASIC/370

A Manual for BASIC, the elementary
algebraic language designed for use
with Hercules and the MVS
Batch System
Or Z/OS

*** V1.0 ***
May 27, 2025

BASIC/370 © Copyright 2020, 2022, 2025
by Edward G Liss
All Right Reserved

Table of Contents

BASIC

A Manual for BASIC, the elementary
algebraic language designed for use
with the Dartmouth Time Sharing System.

1 October 1964

Copyright 1964 by the Trustees of
Dartmouth College. Reproduction for
non-commercial use is permitted provided
due credit is given to Dartmouth College.

The development of the BASIC Language,
and of this Manual, has been supported
in part by the National Science Foundation
under the terms of Grant NSF GE 3864.

I. Introduction

BASIC/370 is an implementation of the BASIC computer programming language. BASIC370 is based on the original document produced at Dartmouth in 1964. BASIC370 is a close implementation but it not a exact implementation. The most significant difference is Dartmouth's BASIC was interactive using teletypewriters where as BASIC370 is batch based using "cards" as the input source.

The source for this document was an "nth" generation copy of the original mimeographed manual that was OCR scanned into MS Word. The document was reformatted and revised to describe BASIC370.

Throughout this document, the terms BASIC and BASIC370 will be used interchangeably.

II. What's New In BASIC370 Version 1.0.0

BASIC360 was originally intended for use by students learning the BASIC language. A "batch" of program could be submitted and limits applied to stop run away loops (BASICMON). BASIC360 was later cloned into a "stand alone" (BASIC1UP) with some limits could be overridden.

BASIC370

- drops support for the "batch" (BASICMON) since today's environment is not "batch program" oriented
- is a rename of BASIC1UP
- by default ignores sequence numbers in columns 73-80 since today's environment is not card oriented
- adds the file i/o statements GET and PUT
- adds the ability to override the default limits so larger programs could be written.

III. WHAT IS A PROGRAM?

A program is a set of directions, a recipe, that is used to provide an answer to some problem. It usually consists of a set of instructions to be performed or carried out in a certain order. It starts with the given data and parameters as the ingredients, and ends up with a set of answers, as the cake. And, as with ordinary cakes, if you make a mistake in your program, you will end up with something else -- perhaps hash!

Any program must fulfill two requirements before it can even be carried out. The first is that it must be presented in a language that is understood by the "computer." If the program is a set of instructions for solving linear equations, and the "computer" is a person, the program will be presented in some combination of mathematical notation and English. If the person solving the equations is a Frenchman, the program must be in French. If the" computer" is a high speed digital computer, the program must be presented in a language the computer can understand.

The second requirement for all programs is that they must be completely and precisely stated. This requirement is crucial when dealing with a digital computer, which has no ability to infer what you meant -- it can act only upon what you actually present to it.

We are of course talking about programs that provide numerical answers to numerical problems. To present a program in the English language, while easy on the programmer, poses great difficulties for the computer because English, or any other spoken language is rich with ambiguities and redundancies, those qualities which make poetry possible but computing impossible. Instead, you present your program in a language that resembles ordinary mathematical notation, which has a simple vocabulary and grammar and which permits a complete and precise specification of your program. The language that you will use is BASIC (Beginner's All purpose Symbolic instruction Code) which is at the same time precise, simple, and easy to understand.

Your first introduction to the BASIC language will be through an example. Next you will learn how to use the Dartmouth Time Sharing System to execute BASIC programs. Finally, you will study the language in more detail with emphasis on its rules of grammar and on examples that show the application of computing to a wide variety of problems.

IV. A BASIC PRIMER

1. An Example

The following example is a complete BASIC program for solving two simultaneous linear equations in two unknowns with possibly several different right hand sides. The equations to be solved are

$$A_1X_1 + A_2X_2 = B_1$$
$$A_3X_1 + A_4X_2 = B_2$$

Since there are only two equations, we may find the solution by the formulas

$$X_1 = \frac{(B_1A_4 - B_2A_2)}{(A_1A_4 - A_3A_2)} \qquad X_2 = \frac{(A_1B_2 - A_3B_1)}{(A_1A_4 - A_3A_2)}$$

It is noted that a unique solution does not exist when the denominator $(A_1A_4 - A_3A_2)$ is equal to zero. Study the example carefully -- in most cases the purpose of each line in the program is self-evident.

```
10 READ AI, A2, A3, A4
15 LET D = Al * A4 - A3 * A2
20 IF D = 0 THEN 65
30 READ B1, B2
37 LET Xl = (B1*A4 - B2*A2) / D
42 LET X2 = (A1*B2 - A3*B1)/D
55 PRINT Xl, X2
60 GO TO 30
65 PRINT "NO UNIQUE SOLUTION"
70 DATA 1, 2,4
```

```
80 DATA 2, -7, 5
85 DATA 1, 3, 4, -7
90 END
```

We immediately observe several things about the above sample program. First, all lines in the program start with a line number. These serve to identify the lines in the program, each one of which is called a statement; thus a program is made up of statements, most of which are instructions to be performed by the computer. These line numbers also serve to specify the order in which the statements are to be performed by the computer, which means that you could enter your program in any order. Before the program is run by the computer, it lines must be in order by line number.

The second observation is that each statement starts, after its line number with an English word. This word denotes the type of the statement. There are fifteen types of statements in BASIC, nine of which are discussed in this chapter. Of these nine, seven appear in the sample program above.

The· third observation is that we use only capital letters, and that the letter “Oh” is distinguished from the numeral "Zero" by shape of the letter – O vs. 0. This feature is made necessary by the fact that in a computer program it is not always possible to tell from the context whether the letter or the number was intended unless they have a different appearance. This distinction is made automatically by the terminal, which also has a special key for the number "One" to distinguish it from the letter "Eye" or lower case “L”.

A fourth observation, though perhaps less obvious than the first three, is that spaces have significance in BASIC. This results in a more readable program. Line number must be separated from keywords by at least one space. Keywords must be separated from the balance of a statement by at least one space. For instance, statement 15 could have been typed as

```
15 LET D=Al*A4-A3*A2
```

a fully equivalent though less readable form.

Turning now to the individual statements in the program, we observe that the first statement, numbered 10, is a READ statement. When the computer encounters a READ statement while executing your program, it will cause the variables whose names are listed after the READ to be given values according to the next available numbers in the DATA statements. Thus, in the example, when statement 10 is first encountered, it will cause the variable, A1 to be given the value 1, the variable A2 to be given the value 2, the variable A3, to be given the value 4, and the variable A4 to be given the value 2.

The next statement, numbered 15, is a LET statement. It causes the computer to compute the value of the expression ($A_1A_4 – A_3A_2$) and to assign this value to the variable D. The expression computed in a LET statement can range from the very simple (consisting of only a single variable) to the very complex. The rules for forming these expressions are given in detail in the next section, but for now we point out that:

1. Variable names consist of a capital letter possibly followed by up to 7 capital letters and/or digits;

2. The symbol * (asterisk) is always used to denote multiplication;

3. Parentheses may be needed to specify the order of the computation because the entire expression must appear on a single line;

4. No subscripts[1] or superscripts as such are permitted, also because the expression must appear on a single line.

In line 20 the computer asks a question "Is D equal to 0" If the answer is yes, then the next statement to be executed by the computer is the one numbered 65. If the answer is no, the computer continues to statement in line 30, the next higher numbered one after 20.

In line 30 the computer causes the variables B1 and B2 to be given the values next appearing in the DATA statements elsewhere in the program. Since the first four data have already been used up, B1 is given the fifth value -7, and B2 is given the sixth value 5.

The statements numbered 37 and 42 complete the computation of the solution, X1 and X2. Notice that the denominator has been previously evaluated as the variable D. Thus it is not necessary to repeat the formula given in statement 15. Notice also how parentheses are used to specify that the numerator of the fraction consists of the entire quantity B1*A4 – B2*A2. If the parentheses had been omitted by mistake, the expression computed as

$$B1*A4 - \frac{B2*A2}{D}$$

which is incorrect.

Now that the answers have been computed, they will be printed out for you to see when the computer encounters statement 55. Notice that the comma is used to separate the Individual items in the list of quantities to be printed out at that time.

Having completed the computation, the statement 60 tells the computer to execute next statement number 30. We observe that the second encounter of statement 30 will cause the variables B1 and B2 be given the values 1 and 3, respectively, the next available ones in the DATA statements.

After completing the computation for the second set of right hand sides and printing the answers, the computer will give the last values, 4 and -7 to the variables B1 and B2, compute and print the third set of answers and then stop, because there is no more data when the READ statement 30 is encountered for the fourth time.

If D, the determinant of the coefficients, is zero, we know that the set of equations does not have a unique solution. In this case, statement 20 will cause the computer to execute statement 65 next. Statement 65 is again a PRINT statement, but instead of numerical answers being printed out, it will produce the English message

NO UNIQUE SOLUTION

We could have used any other recognizable message between the two quotation marks that would have indicated to us that no unique solution was possible for, the given coefficients.

[1] This refers to the small lowered digits like ₂ and not references to an occurrence in a list or table.

After printing the warning message the computer will execute next statement 90, an END statement, which stops the running of the program. (The running will also be stopped when a READ statement is encountered for which there is not sufficient data.) It is extremely important to remember that all programs must have an END statement. It does not always have be the highest numbered statement in the program. When executed, it simply tells BASIC your program has ended execution. The intervening DATA statements are never executed by the computer; therefore, they may be placed anywhere in your program. The only requirement is that DATA statements are numbered in the order in which you wish the data to be used by the various READ statements in your program.

2. Expressions, Numbers, and Variables

Expressions in BASIC look like mathematical formulas, and are formed from numbers, variables, operators, and functions.

A number may contain up to six digits with or without a decimal point, and possibly with a minus sign. For example, the following numbers are acceptable in BASIC:

5 2.5 123456 .123456 -123456

To extend the range of numbers, a factor of a power of ten may be attached, using the letter E to stand for "times ten to the power". Again, the following examples are all acceptable forms for the same number in BASIC:

-12.345 -12345E-3 -.12345E+2 -123450E-6 -.00012345E+5

It should be noted, however, that the E notation cannot stand alone; 1000 may be written 10E2 or 1E3 but not E3 (which looks like a variable and is so interpreted in BASIC.) It should also be noted that .000123456789 is illegal and must be written as, say .123456E-3. Also, the last digit and the E should not have space between them.

A variable in BASIC is denoted by any a capital letter possibly followed by up to 7 capital letters and/or digits. For instance, these are acceptable variable names:

A X N5 X0 K9 SUM R2D2 C3P0

The difference between 0[2] and O and between I and 1 should be observed. Thus, I0 is acceptable while any of 1O. 1O and 10 are not (the last one is the number ten.)

A variable in BASIC stands for a number, usually one that is not known to the programmer at the time the program was written. Variables are given or assigned values by LET or READ statements. The value so assigned will not change until the next time a LET or READ statement is encountered that names that variable.

Expressions are formed by combining variables and numbers together with arithmetic operations and parentheses just as in ordinary mathematical formulas. The symbols

[2] This is the digit zero

+ - * / ^

stand for "plus", "minus", "times", "divided by", and "to the power") respectively. Parentheses are used in the usual way, as in (A1 + X)*(B – C^D7).

Because expressions must be presented as a single line, parentheses are often required where they might not be needed in ordinary mathematical notation. Thus,

$$\frac{A-B}{C} \quad \text{is written as} \quad (A - B)/C$$

to show that the entire quantity A - B is to be divided by C. Omitting the parentheses would leave A - B/C, which is interpreted as A - (B/C).

Another example that arises quite often is

$$\frac{A}{B*C} \quad \text{which is written as} \quad A/(B*C) \text{ or } A/B/C$$

A/B*C will be interpreted the same as (A/B)*C or (A*C)/B.

The way that expressions are interpreted can be summarized in terms of several rules, which correspond to standard mathematical notation. These are:

1. The expression inside a parentheses pair is computed before the parenthesized quantity is used in further computations.

2. Raising to a power is computed before multiply and/or divide, which in turn are computed before addition and/or subtraction, in the absence of parentheses.

3. Several multiply-divides, or several addition-subtractions, are computed from left to right.

The first rule tells us that in (A + B)*C we compute A + B first, then multiply the result by C, an obvious interpretation. The second rule tells us that in A + B * C ^ D we first compute C^iD , then multiply by B, and finally add to A. An equivalent expression is A + (B * (C ^ D)).

The third rule states that A - B - C is interpreted as (A - B) - C and as A - (B - C) . Applied to multiplies and divides, the rule tells us to interpret A/B/C as (A/B)/C and not as A/(B/C). For raising to a power, A ^ B ^ C means (A ^B) ^ C or, equivalently, A ^ (B*C). If you intend A ^ (B ^ C) , you must use that form.

In addition to the arithmetic operations, some of the more common standard functions are available.

For example, to compute $\sqrt{1 + X^2}$ you would use SQR(1 + X ^ 2). The other standard functions are used in this same way, that is, the BASIC name of the function followed by the argument enclosed in parentheses .

Function Name	Purpose
SIN(X)	Sine of X. X must be in radians.
COS(X)	Cosine of X. X must be in radians.
TAN(X)	Tangent of X. X must be in radians.
ABS(X)	Absolute value of X
SQR(X)	Square root of X
RND(X)	Generate Random number. The value of X controls "seeding" of the generator.
INT(X)	Integral Part of X – truncated at decimal no rounding
INR(R)	Integral Part of X rounded and truncated at decimal
EXP(X)	Raises the natural logarithm to the X power
LOG(X)	Finds the natural logarithm of x

Figure 1 Standard Basic Function

The functions, RND(X), INT(X), and INR(X) are explained on page 23 – Functions and Subroutines. A sample program using the INT and INR can be found on page 35 - Figure 2 Sample Looping, Printing and INT/INR use.

The argument of a function may be any expression, no matter how complicated. For example:

```
SQR( B^2- 4*A*C ) - 17
Z - EXP( X1 +LOG( A/X1 )) * TAN(A)
SQR( SIN( Q)^2 + COS(Q)^2)
```

are all acceptable in BASIC.

The use of the LOG and SQR functions requires a word of caution. In each case if the argument is negative, the BASIC program will be terminated before applying the function, since neither function is defined for negative arguments. Many times, though not always, an attempt to have the computer extract the square root of a negative number implies a fundamental error in the program.

The user may define new function, using the DEF statement, which is discussed on page 23 – Functions and Subroutines.

3. Loops

Perhaps the, single most important programming idea is that of a loop. While we can write useful programs in which each statement is performed only once, such a restriction places a substantial limitation on the power of the computer. Therefore, we prepare programs that have portions which are performed not once but many times, perhaps with slight changes each time. This "looping back" is present in the first program, which can be used to solve not one but many sets of simultaneous linear equations having the same left hand sides.

Making tables of, say, square roots is another example where a loop is necessary. Suppose that we wish to have the computer print a table of the first hundred whole numbers and their square roots. Without loops, one can easily see that a program would require 101 lines, all but the last having the form:

```
17 PRINT 17, SQR(17)
```

And if one wished to go not to 100 but to 50 only, a new program would be required. Finally, if one wanted to go to 10,000 the program would be absurd even if someone could be found to write it all down.

We notice that the basic computation, in this case a very simple printing, is practically the same in all cases - - only the number to be printed changes. The following program makes use of a loop.

```
10 LET X = 0
20 LET X = X + 1
30 PRINT X, SQR(X)
40 IF X < 100 THEN 20
50 END
```

Statement 10, which gives to X the value 0, is the initialization of the loop.

Statement 20. which increases the value of X by unity, is the statement that insures that the loop is not merely repeating exactly the same thing -- an infinite loop! Statement 30 is the body of the loop, the computation in which we are interested. And statement 40 provides an exit from the loop after the desired computation has been completed. All loops contain these four characteristics: initialization, modification each time through the loop, the body of the loop, and a way to get out.

Because loops are so important and because loops of the type shown in the example arise so often, BASIC provides two statements to enable one to specify such a loop much more concisely. They are the FOR and the NEXT statements, and would be used as follows in the example above:

```
10 FOR X = 1 TO 100
20 PRINT X, SQR(X)
30 NEXT X
40 END
```

Statement 10 contains both the initial and final values of X. Statement 30 specifies that X be increased to its next value. In this case, the value by which X is increased each time is implied to be unity. If instead we wished to print the square roots of the first 50 even numbers, we would have used

```
10 FOR X = 2 TO 100 STEP 2
20 PRINT X, SQR(X)
30 NEXT X
40 END
```

Omitting the STEP part is the same as assuming the step size to be unity.

To print the square roots of the multiples of 7 that are less than 100, one might use for line number 10

```
10 FOR X = 7 TO 100 STEP 7
```

The loop will be performed for all values of X that are less than or equal to 100, in this case, for X equal to 7, 14, ... , 91, 98.

A sample program showing loops can be found on page 35 - Figure 2 Sample Looping, Printing and INT/INR use.

4. Errors and Debugging

It may occasionally happen that the first run of a new problem will be error-free and give the correct answers. But it is much more common that errors will be present and have to be corrected. Errors are of two types: Errors of form, or grammatical errors, that prevent even the running of the program; Logical errors in the program which cause wrong answers or even no answers to be printed.

Errors of form will cause error messages to be printed out instead of the expected answers. These messages give the nature of the error and the line number in which the error occurred. Logical errors are often much harder to uncover, particularly when the program appears to give nearly correct answers. But after careful analysis and when the incorrect statement or statements are discovered, the correction is made by retyping the incorrect line or lines, by inserting new lines, or by deleting existing lines.

5. Summary Elementary BASIC Statements

This section gives a short and concise but complete description of each of the nine types of BASIC statements discussed earlier in this chapter.

The notation <L…> is used to denote a particular unspecified instance of the thing referred to inside the <>. Thus, <line number> is used to stand for any particular line number. <variable> refers to any variable, which is a single letter possibly followed by a single digit. <expression> stands for any particular expression, no matter how complicated, so long as it follows the rules for forming expressions given in section ”Expressions, Numbers, and Variables” starting on page 11. <number> stands for any constant or data number.

a) LET

Form: <line number> LET <variable> = <expression>

Example: 100 LET X = X + 1
259 LET W7 = (W - X4 ^ 3)*(Z – A1/(A - B)) - 17

Comment: The LET statement is not a statement of algebraic equality, but is rather a command to the computer to perform certain computations and to assign the answer to a certain variable. Thus, the first

example tells the computer to take the current value of X, add 1 to it, and assign the answer to the variable X. In other words, X is increased by unity.

b) READ, DATA and RESTORE

Form: <line number> READ list of <variable>

Example: 150 READ X, Y, Z, Xl, Y2, Z(K+I, J)

Form: <line number> DATA list of <numbers>

Example: 300 DATA 4, 2, 1. 5, 0.6734E-2, -174.3Z1

Form: <line number> RESTORE

Example: 200 RESTORE

Comment: A READ statement causes the variables listed in it to be given in order the next available numbers in the collection of DATA statements.

Before the program is run, the computer takes all the DATA statements in the order in which they appear and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data, with a READ statement still asking for more, the program is terminated with an error message. The RESTORE statement is used to reset the DATA list so the program may process the data again.

c) PRINT

Form: <line number> PRINT <list of expressions to be printed>

Example:· 100 PRINT X, Y, Z, B*B - 4*A*C, EXP(LOG(17))

Form: <line number> PRINT " <any string of characters> "

Example: 200 PRINT" THIS PROGRAM IS NO GOOD. "
150 PRINT "COMPUTES X + Y = Z"

Comment: The numerical quantities printed need not be simple variables, they may be any expressions. The expression is first evaluated, then printed. There may be any number of expressions separated by commas or semi colons.

Example: 150 PRINT "X", "Y", "Z"

Comment: Several messages may be included in the list separated by commas. The effect is to print the letter X in the first column, the letter Y in the 15th column, and the letter Z in the 29th column. Note the print line is BASIC is divided into nine 14 character zones or tabs. See More About PRINT on page 20.

Example: 200 PRINT "X = ", X, "Y = ", Y

Comment: Labels and expressions may appear in the same print statement.

Comment: Much more variety is permitted in PRINT statements than is shown here. The additional flexibility is explained in More About PRINT on page 20.

d) GO TO and IF - THEN

Form: <line number> GO TO <line number>

Example: 150 GO TO 75
240 GO TO 850

Comment: Sometimes called an unconditional go to, GO TO is used to interrupt the normal sequence of executing statements in the increasing order of their line numbers.

Form: <line number> IF <expression> <relation><expression> THEN <line number>

Example: 140 IF X > Y + Z THEN 200
85 IF X * SIN(X) >= 1 THEN 100

Comment: Sometimes called a conditional go to, the IF-THEN statement provides a way to select one of two sequences in the program depending on the results of some previous computation. If the condition is met, the implied go to is performed; if the condition is not met, the next statement in sequence is performed.

Any of the six standard relations may be used.

Symbol	Meaning
<	Less than
<=	Less than or equal
=	Equal
>=	Greater than or equal
>	Greater than
<>	Not Equal

e) FOR and NEXT

<line number> FOR <variable> = <expression> TO <expression>
or

<line number> FOR <variable> = <expression> TO <expression> STEP <expression>

Example: 120 FOR X4 = (17 + COS(A))/3 TO 3*SQR(10) STEP 1/4
(This represents the body of the loop.)
235 NEXT X4

Comment: Omitting the STEP part of the FOR statement is equivalent to having the step size equal to unity.

Comment: The above example will, assuming A to be equal to 0, cause the body of the loop to be performed several times, first with X4 equal to 6, next with X4 equal to 6.25, then 6.50, and so on. The last time the body of the loop will be performed is with X4 equal to 9. 25, which is less than or equal to the final value 9.486 (approximately).

The FOR statement goes into the body of a loop if the variable has a value less than or equal to the final value (in the case of a positive stepsize), or if the be variable has a value greater than or equal to the final value (in the case of a negative stepsize.)

Upon leaving the loop the program continues with the statement following the NEXT; the variable used in the FOR statement then has the value it had during the last passage through the loop r 9.2.5 in the above example.

Example: 240 FOR X= 8 TO 3 STEP -1

Comment: The body of the loop is performed with X equal to 8, 7, 6, 5, 4, and 3, and X has the value 3 upon Ieaving the loop.

Example: 456 FOR J = -3 TO 12 STEP 2

Comment: The body of the loop will be performed with J equal to -3, -1, 1, 3, 5, 7, 9, and 11. J will have the vaIue 11 upon leaving the loop.

Example: 50 FOR Z = 2 TO -2

Comment: The body of the loop will not be performed. Instead, the computer will proceed to the statement immediately following the corresponding NEXT. The value of Z will then be 1, which is the initial value (2) minus the step size (1).

f) END

Form: <line number> END

Example: 999 END

Comment: An END statement is required in all programs. It indicate normal end of execution of a BASIC program. More than one END statement may appear in a BASIC program.

g) STOP

Form: <line number> STOP

Example: 999 STOP

Comment: An STOP statement is optional in all programs. It indicate abnormal end of execution of a BASIC program.

h) RANDOMIZE

Form: <line number> RANDOMIZE

Example: 123 RANDOMIZE

Comment: A RANDOMIZE statement is optional in all programs. It indicates that the RND function is to be "reseeded" to the last 4 digits of the system time of day. An implicit RANDOMIZE is performed prior to the execution of all BASIC program. It is also equivalent to a RND(X) when X<0.

i) INPUT

Form: <line number> INPUT list of numeric or string variables

Example: 123 INPUT X, Y, Z$,A$

Comment: Similar to the READ statement, the INPUT statement allows the user to enter numbers or strings into running BASIC programs. The major difference is the source of the data. The READ takes the data from the list of values supplied in the DATA statement. The INPUT statement accepts data from an external source. This external source is referred to as the INPUT DD statement which is usually 80 character "card" images. Data is extracted from columns 1 to 80 for each "card".

The INPUT file is processed like a stream of characters and not as records. When the first INPUT starts, a "pointer" is set to the first column of the first card. If character at pointer is blank, the pointer advances to the next non-blank character. All characters to the next blank or comma represent the value going to the variable being INPUTed. Individual values end at the column 80 and cannot span over 2 cards.

Example: An input card is read (where b is a blank): b123 456 ZEBRA "TWO WORDS"
When INPUT X is executed, the number 123 will assigned to X. When INPUT Y is executed, the number 456 will be assigned to Y. When INPUT Z$ is executed, the string ZEBRA will be assigned to Z$.

There is a special case when a double quote (“) character is encountered as the first non blank character of the input. All characters to the next double quote (“) character is assigned to a string variable. When INPUT Z$ is executed, the string “TWO WORDS” will be assigned to Z$

V. MORE ADVANCED BASIC

a) GET

Form: <line number> GET list of numeric or string variables

Example: 123 GET X, Y, Z$,A$

Comment: Similar to the INPUT statement, the GET statement allows the user read data from a file. The file is processed like a stream of characters and not as records. When the first GET starts, a “pointer” is set to the first column of the first card. If character at pointer is blank, the pointer advances to the next non-blank character. All characters to the next blank or comma represent the value going to the variable being “GETed”. Individual values end at the column 80 and cannot span over 2 cards. The dd name “GETFILE” is the source of data. See INPUT statement for how file is processed.

b) PUT

Form: <line number> PUT list of numeric or string variables

Example: 123 PUT X, Y, Z$,A$

Comment: The PUT statement allows the user write data to a file. The data is written to a file in a format compatible with the GET and INPUT statement.

c) More About PRINT

One of the conveniences of BASIC is that the format of answers is automatically supplied for the beginner. The PRINT statement does, however, permit a greater flexibility for the more advanced programmer who wishes to specify a more elaborate output.

The print line is divided into nine zones of fourteen spaces each by BASIC, allowing the printing of up to nine numbers per line. Three simple rules control the use of these zones.

1. A label, in quotes, is printed just as it appears.
2. A comma is a signal to move to the next print zone, or to the first print zone of the next line if it has just filled the ninth print zone.
3. A semi colon is a signal to move to the next character in the current print zone.

4. The end of a PRINT statement signals a new line, unless a comma or semi colon is the last symbol.

Each number occupies one zone. Each label occupies a whole number of zones; if it occupies part of a zone, the rest of the zone is filled with blanks. If a label runs through the last zone, a new line is started before the label is printed.

The example in Figure 3 Advanced Printing Example on page 36 illustrates some of the various ways in which the PRINT statement can be used. It should be noted that a blank PRINT statement causes the printer to move to the next line, as is implied by rule 4 above.

In line 37, the underlines for the 4 columns is printed with one PRINT. The comma causes the next PRINT to print in the next zone. The PRINT in line 39 advances to the next line.

In line 50, the four values are printed, one to a zone because of the comma after each value. The format in which the BASIC PRINT statement prints numbers is not under the control of the user. However, the following rules may be used to guide the programmer in interpreting the results.

1. No more than six significant digits are printed (except for integers - - see rule 4.)
2. Any trailing zeros after the decimal point are not printed.
3. For numbers less than 0.000001, the form X. XXXXXE-Y is used unless the entire significant part of the number can be printed as a six decimal number. Thus, .03456 means that the number is exactly .0345600000, while 3.45600E-2 means that the number has been rounded to .0345600 .
4. If the number is an exact integer, the decimal point is not printed. Furthermore, integers of up through nine digits are printed in full.

In line 120, note the TAB(J) in the print statement. The TAB function causes the next item to print to start in column(J) of the print line. If J is invalid or greater than 120, an error message is issued and the program terminated. If J is after the current print position, the next print position is set to column J. Otherwise, a new line is started and the next print position is set to column J. Note the “,” after the TAB does not skip to the next print zone. It functions like a “;”.

A packed form of output is available by using the character ";" instead of ",". Briefly, whereas "," tells the computer to move to the next zone for the next answer, “;” tells the computer to move to the next character in the current zone for the next answer instead of to the next zone. One can thus pack many more than nine numbers on a line if the numbers themselves require less than a full zone to print.

With packed output using the semi-colon, mixtures of the three in subsequent lines may not line up, as the example shows. The user should be careful about using the semi-colon with full length numbers which might occur near the end of a print line. BASIC checks to see if there is enough room on a line. If not, a new line is started and the item printed on the new line.

d) PRINT USING

One of the conveniences of BASIC is that the format of answers is automatically supplied for the beginner. The PRINT USING statement does permit a greater flexibility for the more advanced

programmer who wishes to specify a more elaborate output. PRINT USING allows the program to specify how the values or strings are printed.
Form: <line number> PRINT USING <edit string>,<items to print>

Example: 999 PRINT USING "THE AMOUNT DUE IS $ #,###.##",100

Comment: This would print as THE AMOUNT DUE IS $ 100.00

The <items to print> is similar the same as for a PRINT statement. The exceptions are that "," and ";" are ignored except as the last character in the PRINT USING. The dangling "," and ";" are the same as with PRINT – either skip to new zone or next character in print line. TAB() is also ignored in PRINT USING.

The edit string controls the spacing. Currently, edit strings are limited to what can fit on a "card", about 60 characters.

Items to print are mapped from left to right into the edit string. Each pattern in the edit string must have a corresponding item in the items to print list AND the types (numeric and string) must match.

The first "#" encountered defines the beginning of number pattern. The next space, if any, ends the number pattern. In the example above, the number pattern is "#,###.##". The first item to print must be numeric since this pattern is numeric. The value 100 will be edited into the number pattern"#,###.##" to yield "bb100.00" where b=blank. The edit rules are:

1. if fraction is to print, indicated by a "." in the pattern, the corresponding digit in the value is printed;
2. leading zeros are suppressed. If the number is negative, the "-" will be placed in front of the 1st significant digit (i.e. -100 would edit as "b-100.00")
3. Any character other than a "#" is a fill character and will be included in the result unless there are no significant digits to the left. (i.e. 100 would edit as bb100.00 there are no significant digits to the left of it).
4. The first "." Starting at the left is considered the decimal point. Any other "." Are considered fill characters per rule 3.

The first "&" encountered defines the beginning of a string pattern. The next space, if any, ends the string pattern. Any characters between the 1st & and the space define the string pattern. The corresponding print item must be a string item. Edit rules for string patterns are:

1. If the pattern is a single &, the entire string is substituted;
2. If the pattern is more one &, the string is adjusted to fit into the number of characters in the pattern;
3. No fill characters are supported for strings.

Referring to Figure 4 Print Using Sample 1 on page 37, the sample program demonstrates the interaction of the PRINT and PRINT USING statements. Lines 200 to 260 print the output using the ",". In Figure 5 Print Using Sample 2 on page 38, the program is the same except the PRINT statement use the ";" to end the lines. Notice the difference in how the items are positioned.

e) EJECT Statement

The EJECT statement causes the printer skip to the top of a new page. When an EJECT is executed, it is equivalent to a PRINT statement followed by a skip to top of page.

```
20 EJECT
```

f) Lists and Tables

In addition to the ordinary variables used by BASIC, there are variables that can be used to designate lists or tables. For instance, A(7), would denote the seventh item in a list called A; B(3,7) denotes the item in the third row and seventh column of the table called B. We commonly write A_7 and $B_{3,7}$ for those same items, and use the term subscripts to denote the numbers that point to the desired items in the list or table. (The reader may recognize that lists and tables are called, respectively, vectors and matrices by mathematicians.)

The name of a list or table must follow the naming rule for a variable. See Expressions, Numbers, and Variables on page 11 for rules. The subscript may be any expression, no matter how complicated, as long as they have non-negative integer values.

BASIC provides that each list has a subscript running from 0 to 10, inclusive. Each subscript in a table may run from 0 to 10. All lists and tables must be defined with a DIM statement. For example,

```
10 DIM A(17)
```

indicates to the computer that the subscript of the list A runs from 0 to 17, inclusive; similarly,

```
20 DIM S(3,4)
```

means that the first subscript of the table S runs from 0 through 3 and the second subscript of the table S runs from 0 through 4. The numbers used to denote the size of a list in a DIM statement must be integer numbers. The DIM statement is used not only to indicate that lists and tables are larger than 0-10 in each subscript, but also to allocate storage space in very large programs by telling the computer that only, say, 4 spaces are needed for the list 8 as shown above.

It should be mentioned that using a DIM statement does not require the user to use all of the spaces so allocated.

Figure 6 Sample of a list on page 39 is an example of using lists. A list of values is created in the first loop and the second list prints that list in reverse order.

g) Functions and Subroutines

Three additional functions that are in the BASIC repertory but which were not described in Expression, Numbers and Variables (see page 11) are INT, INR and RND. INT is used to determine the integer part of a number that might not be a whole number. Thus INT(7.8) is equal to 7. As with the other functions, the argument of INT may be any expression. One use of INT is to round numbers[3] to the nearest whole

[3] The INR function is INT with rounding built in.

integer. If the number is positive, use INT(X + .5). The reader should verify that this process is equivalent to the familiar process of rounding. If the number is negative, INT(X - .5) must be used. The reason is that INT(-7.8) is -7, not -8. INT always operates by chopping off the fractional part, whether the number is positive or negative.

INT can be used to round to any specific number of decimal places. Again, for positive numbers,

INT(100*X + . 5) /100

will round X to the nearest correct two decimal number.

The INR function is the INT function with rounding built in. INR(7.7) is equal to 8. INR(-7.7) is -8.

The function RND produces a random number between 0 and 1. The form of RND requires an argument; thus, we commonly choose a single letter such as X and use RND(X). The value of X is significant.

- When X=0, a new seed is generated by the RND function.
- When X>0, the value of X become the new seed. It is recommended that the value of X be greater than 1000 for best results.
- When X<0, the random number is "seeded", as described by the RANDOMIZE statement on page 19 .

The property of RND is that it produces a new and different random number each time it is used in a program. Thus, to produce and print 20 random digits, one might write a program like that shown in Figure 1 - Random Number Program on page 34.

Additional flexibility is provided in BASIC by three statements that permit the use of user defined functions and subroutines.

The DEF statement permits the user to define a function other than the standard functions listed in Figure 1 Standard Basic Function (page 13) so that it doesn't have to be repeated for the function each time it is used in the program.

The name of a defined function must be three letters, the first two of which are FN. The user thus may define up to 26 functions. The following examples illustrate the form of the DEF statement:

25 DEF FNF(Z) = SIN(Z*P) (where P has the value of 3. 14159265/180)

40 DEF FNL(X)= LOG(X)/ LOG(10)

Thus, FNF is the sine function measured in degrees, and FNL is the function log-to-the-base-ten.

The DEF statement may occur anywhere in the program before it is used. The user needs to be cautioned that the variable used in the DEF statement must not be subscripted, and that it is used every time that function is used. Thus, in a program containing FNF as above defined, it is best not to use the variable Z elsewhere in the program.

The expression on the right of the equal sign' can be any expression that can be fit into one line. It could involve many other variables besides the one denoting the argument of the function; Thus,

```
60  DEF FNX(X) = SQR(X*X + Y*Y)
```

may be used to set up a function that computes the square root of the sum of the squares of X and Y. To use FNX, one might use the following:

```
10 LET Y = 30
20 LET S1 = FNX(40)
```

Of course, S1 would end up having the value 50. It should be noted that one does not need DEF unless the defined function must appear at two or more locations in the program. Thus,

```
10 DEF FNF(Z) = SIN(Z*P)
20 LET P = 3. 14159265/180
30 FOR X = 0 TO 90
40 PRINT X, FNF(X)
50 NEXT X
60 END
```

might be more, efficiently written as

```
20 LET P = 3. 14159265/180
30 FOR X = 0 TO 90
40 PRINT X, SIN(X*P)
50 NEXT X
60 END
```

to compute a table of values of the sine function in degrees.

The use of DEF is limited to those cases where the value of the function can be computed within a single BASIC statement. Often much more complicated functions, or perhaps even pieces of program that are not functions, must be calculated at several different points within the program. For this, the GOSUB statement may frequently be useful.

The form of a GOSUB statement is illustrated as follows:

```
25 GOSUB 180
```

The effect of the GOSUB is exactly the same as a GOTO except that note is taken by the, computer as to where the GOSUB statement is in the program. As soon as a RETURN statement is encountered, the computer automatically goes back to the statement immediately following the GOSUB. As a skeleton example:

```
100 LET X=3
110 GOSUB 400
120 PRINT U, V, W
```

```
200 LET X = 5
210 GOSUB 400

220 LET Z = U + 2*V + 3*W



400 LET U = X*X
410 LET V = X*X*X
420 LET W = X*X*X*X + X*X*X + X*X + X
430 RETURN
```

When statement 400 is entered by the GOSUB 400 in line 110, the computations in lines 400, 410, and 420 are performed, after which the computer goes back to statement 120. When the subroutine is entered from staternent 210, the computer goes back to statement 220.

As a complete illustration, the next page contains a program that determines the Greatest Common Divisor of three integers, using the celebrated Euclidean algorithm as a subroutine. The subroutine is contained in lines 200 to 310, and is applied to two integers only. The main routine applies this subroutine to the first two integers, and then to the GCD of these and the third integer. The GCD is then printed, and a new case considered.

h) Some Ideas for More Advanced Programmers

An important part of any computer program is the description of what it does, and what data should be supplied. This description is commonly called documentation. One of the ways a computer program can be documented is by supplying remarks along with the program itself. BASIC provides for this capability with the REM statement. For example:

```
10 PRINT "A", "B", "C", "GCD”
20 READ A, B, C
30 LET X = A
40 LET Y = B
50 G0SUB 200
60 LET X = G
70 LET Y = C
80 G0SUB 200
90 PRINT A, B, C, G
100 G0 T0 20
110 DATA 60, 90, 120
120 DATA  38456, 64872, 98765
130 DATA 32, 384, 72
200 LET Q = INT(X/Y)
```

```
210 LET R : X - Q*Y
220 IF R = 0 THEN 300
230 LET X = Y
240 LET Y = R
250 G0TO 200
300 LET G = Y
310 RETURN
999 END
```

Each user quickly learns how much documentation he needs to permit him to understand his program, and where to put REM statements. But it is certain that REM's are needed in any saved program. It should be emphasized that REM's have absolutely no effect on the computation.

Sometimes a program will have two or more natural ending points. In such a case the programmer might use a GOTO to the END statement. Such a statement can be replaced by a STOP with nothing following the word STOP. Thus,

```
400   GOTO  999
710   GOTO  999
999   END
```

may be replaced by

```
400   STOP
710   STOP
999   END
```

BASIC allows GO TO and IF - THEN statements to point to REM and DATA statements. The effect is to perform a vacuous statement having that number and proceed to the next numbered statement. In the case of DATA statements, the END statement might eventually be reached. However, for REM statements the programmer might deliberately have his GO TO's point to REM statements, the remark part identifying that part of the program.

One of the most important and difficult problems in computing is that of round-off error. It exerts its influence in subtle ways, and sometimes in ways not so subtle. A full treatment of the effects of round-off error is beyond the scope of this manuaI, but one fairly common situation will be discussed.

Most programmers eventually write or encounter a program something like this:

```
5 LET S = 0
10 LET X =0
20 LET S = S + X
30 IF X >= 2 THEN 60
40 LET X = X + .1
50 GO TO 20
60 PRINT S
70 END
```

for computing the sum of all the non-negative multiples of .1 less than or equal to 2. The correct answer is 21, but invariably the program will produce 23. 1 as the answer. What is wrong? Round-off has reared its ugly head high enough for us to see. The explanation is that the computer works in the binary

number system, and cannot express .1 exactly. Just as 1/3 cannot be expressed in terms of a single decimal number, neither can .1 be expressed in terms of a single, binary number. It turns out that. 1 in the computer is a number very slightly less than. 1. Thus, when the loop in the above example has been performed 21 times, the value of X is not 2 exactly, but is very slightly less than 2. The IF statement in line 30 determines that the final value, exactly 2, has not been achieved or exceeded, and so calls for one more passage through the loop.

If the programmer had known that the computer treats .1 as a number slightly less, he could have compensated by writing 1.95 in place of 2 in statement 30. A better way rests on the fact that the computer performs exactly correct arithmetic for integers. The user may thus count the number of times through the loop with integers. The example may be rewritten as follows:

```
5 LET S = 0
10 LET N = 0
20 LET S = S + N/10
30 IF N> 20 THEN 60
40 LET N = N + 1
50 GOTO 20
60 PRINT S
70 END
```

Better still a FOR statement can shorten the program to

```
10 LET X = 0
20 FOR N = 1 TO 20
30 LET S = S + N/10
40 NEXT N
50 PRINT S
60 END
```

One of the most exasperating problems confronting programmers is that of a fairly long and complex program that looks like it should work simply refuses to do so. (Presumably, all errors of form have been detected and removed.) The locating and removing of logical error is called debugging, and the methods to be used depend on the nature of the program and also on the programmer himself. All important part of debugging is intuition, but it is possible to suggest some approaches that might be useful in many cases.

The first thing to do with an apparently incorrect program is to check very carefully the method used. If that does not uncover the bug, then examine very carefully your programming to see if you have mixed up any of the variables. It is often difficult to spot such errors because one tends to see in a program what he expects to see rather than what is there.

Another method that is extremely useful in providing clues as to the nature and location of the bug or bugs is tracing. In BASIC this tracing may be accomplished by inserting superfluous PRINT statements at various places in your program to print the values of some of the intermediate quantities. When the program is then, RUN, the values of these intermediate quantities often suggest the exact nature of the error. When the program has been debugged and is working properly, these statements are removed.

There are some matters that do not affect the correct running of programs, but pertain to style and neatness. For instance, as between two or more similar ways to prepare a part of a program, one should select the one that is most easily understood unless there is an important reason not to do so.

More experienced programmers will tend to group the data in DATA statements so that it reflects the READ statements that correspond. The first example on linear equations represents bad style, but was done purposely to illustrate that one can arbitrarily group the data in the DATA statements .

One tends after a while to place his data statements near the end of the program, or near the beginning, but at least in one group to avoid confusing himself with DATA statements spread throughout the program. Some programmers also tend to give the END statement a number like 9999 to insure that it will be the one with the highest number.

No doubt the user will be able to devise ways to make a program neat and readable. But again, the important consideration in style is to program in a way that makes it more understandable and useful to both oneself and others in the future.

BASIC370 has the ability to put multiple statements per line by replacing line numbers with the "\" character. For example, this sample code fragment produce the same results.

10 LET X = 0 20 FOR N = 1 TO 20 30 LET S = S + N/10 40 NEXT N 50 PRINT S 60 END	10 LET X = 0 20 FOR N = 1 TO 20\LET S = S + N/10\NEXT N 50 PRINT S\END

VI. BASIC Option Statements

BASIC supports a number options statements to direct actions to take place. These options can appear in the BASIC program and are identified with a "*" in column 1. Below is a list options and explainations.

OPTION	Meaning
*REM	Alternate way of specifying a remark.
*LIST	Print the program listing (default)
*NOLIST	Do not print the program listing.
*NOEXEC	Compile the BASIC program but don't execute it.
*RENUM	Renumber the programs line numbers. First line will be 10 stepping by 10.
*RENUM10	Renumber the programs line numbers. First line will be 10 stepping by 10.
*RENUM100	Renumber the programs line numbers. First line will be 100 stepping by 10.
*RENUM1000	Renumber the programs line numbers. First line will be 1000 stepping by 10.

*RENUM10000	Renumber the programs line numbers. First line will be 10000 stepping by 10.
*LIB=HERC01.BASICLIB.BAS	Specifies an alternate library for *APPEND and *SAVE. If not specified, default is shown.
*APPEND=M1234567	Specifies the program name to be added to the end of the current program.
*SAVE=M1234567	Specifies the name of the current program should be saved as.
*NUM	Program has sequence numbers in col 73-80
*NONUM	Process all 80 columns of program code (default)
*TABLE	Debugging tool not normally used.
*DUMP	Debugging tool not normally used.
*STACK	Debugging tool not normally used.
*ICODE	Debugging tool not normally used.
*TRACE	Debugging tool not normally used.

VII. Library Access

BASIC360 provides a facility for storing and retrieving BASIC code. There are 3 option statements described on page 29.The *LIB allows the current program to access a library other than the current default. The default library is HERC01.BASICLIB.BAS. To change the default for all programs in a batch run, the library name is passed via the parm on the exec statement. For example, the change the default to AUSER.PROJECT.STUFF for all programs in a run, code this for line 2 in the above example:

```
//  EXEC  BASICMON,LIB='AUSER.PROJECT.STUFF'
```

The *APPEND causes the named item to be copied from the library and appended to the end of the current program. The line numbers in appended code must be larger than the last line in current program. You may use as many *APPENDs as you like as long as the above rule is followed. Referring to Figure 7 Example using *APPEND on page 40, the submitted program ends at line 180. Everything after the *REM was copied from library item TRIGPLOT.

The *SAVE causes the current program less any appended code to be added or replaced in the library. Referring to Figure 8 Example using *SAVE on page 41, all the lines up to line number 180 would be saved in the library as MYPROG. Everything starting with the *REM will not be saved.

VIII. Using BASIC370

BASIC370 V1.0.0 is implemented to be run on MVS 3.8 Hercules mainframe environment using the TK5 system. It was successfully test on a Z/OS system[4]. BASIC programs will be prepared and saved in a dataset, usually a PDS (partitioned data set). In this document, these will be referred to as libraries. Programs will be prepared using a TSO editor (like RPF, FSE or SPFLite) and be submitted to run as batch jobs. The output can then be reviewed using a output display tool (like OUTPUT command or the LISTING tool). This is all outside of the scope of this manual.

BASIC370, like its predecessor BASIC1UP, is designed to execute only 1 program per job step and optionally override all of the defaults parameters.

Below is a sample of a typical job. The first 2 lines tell MVS what to do with the job. The last two lines are the data being supplied to process the INPUT statement.

```
//HERC01B JOB CLASS=A,MSGLEVEL=(1,1),MSGCLASS=A
// EXEC BASIC370
//BASIC.SYSIN DD *
10 REM TEST INPUT STATEMENT
20 PRINT "CONTINUE (Y/N)?"
30 INPUT A$
35 PRINT "BLANK1"
40 PRINT "ANSWER IS ";A$
45 PRINT "BLANK2"
50 END
//INPUT DD *
MAYBE
```

IX. Defaults

BASIC370 has a number of default limits.

DATA_STACK	500	MAX NUMBER OF DATA NUMBERS AND STRING
MAX_LINES	500	MAX NUMBER OF LINES IN BASIC PROGRAM
MAX_SYM	200	MAX NUMBER DATA ELEMENTS IN SYMBOL TABLE
MAX_PCODE	500	MAX NUMBER OF PCODES
MAX_EXECS	5000	MAX PCODES BEFORE ABORTED AS LOOPED. I If zero, no loop checking takes place.
DEFAULT_DSN	HERC01.BASICLIB.BAS	LIBRARY PDS

[4]Since BASIC370 was compiled with PL/I (F), for systems other than TK5, 'SYS1.PL1LIB' must be in the STEPLIB, JOBLIB or linklib.

6. Changing Default Limits

Any of these default limits can be overriden for a BASIC program. Note - most indexes are defined as half words so you should not make any of the defaults greater that 32767. MAX_EXECS is the exception. It is defined as a full word and it may be set up 16M. If set to zero, no limit is imposed.

a) Override Default limits via the PROC

This is the simplest way to override the default limits. Listed below is the definition of the proc and the keywords.

```
//BASIC370 PROC SOUT='*',
//             LIB='HERC01.BASIC370.BAS',
//             STACK=500,               MAXIMUM DATA STACK
//             SYM=200,                 MAXIMUM NUMBER OF SYMBOLS
//             LINES=500,               MAXIMUM NUMBER OF BASIC LINES
//             PCODE=500,               MAXIMUM NUMBER OF PCODES
//             EXECS=5000               NUMBER OF PCODES EXECUTED TO
//*                                        DECLARE RUNAWAY PROGRAM
```

There are seven keywords defined in the PROC BASIC370. To change any or all of them, code them on the EXEC statement. Example:

```
//step EXEC BASIC370,EXECS=10000,PCODE=1000
```

b) Override Defaults via PARM

If your desire, you can code the limit overrides via the EXEC PARM. This is demonstrated by the BASIC370 proc.

c) Override Defaults via OPTIONS file

If your desire, you can code the limit overrides via an options file. Note – if a EXEC PARM is supplied, the options file is NOT used. To use the options file, make sure the PARM is null and the OPTION DD is added to the JCL. Example:

```
//step EXEC BASIC370,PARM.BASIC=
//BASIC.OPTIONS DD …...
```

The option file is a single “card” 80 bytes long. The options are coded as “keyword=value”. Example:

```
//BASIC.OPTIONS DD *
 MAX_EXECS=10000 MAX_PCODE=1000 LIB=”some PDS DSN”
```

d) Permanent Override Changes

To permanently change the default limits, edit the userid.BASIC370.SOURCE(BASIC370). Near the bottom of the program, the defaults are set. Recompile the userid.BASIC370.SOURCE($COMPILE).

7. Changing Default Library

The default library mits can be overriden for a BASIC program. There are three ways to do it.t indexes are defined as half words so you should not make any of the defaults greater that 32767.

a) Override Default library via the PROC

This is the simplest way to override the default limits. Example:inition of the proc and the keywords.

```
//step EXEC BASIC370,LIB='some PDS DSN'
```

b) Override Default Library via program options.

The default library can be set using int *LIB option in the program. It is included for backward compatibility with BASIC360 but using it is not recommended.

c) Override Default Library via PARM

If your desire, you can code the limit overrides via the EXEC PARM. This is demonstrated by the BASIC370 proc. Another option is to update the default in the BASIC370 proc.

d) Override Default Library via OPTIONS file

If your desire, you can code the library via an options file. Note – if a EXEC PARM is supplied, the options file is NOT used. To use the options file, make sure the PARM is null and the OPTION DD is added to the JCL. Example:

```
//step EXEC BASIC370,PARM.BASIC=
//BASIC.OPTIONS DD …...
```

The option file is a single "card" 80 bytes long. The options are coded as "keyword=value". Example:

```
//BASIC.OPTIONS DD *
 MAX_EXECS=10000 MAX_PCODE=1000 LIB="some PDS DSN"
```

e) Permanent Override Changes

To permanently change the default limits, edit the userid.BASIC370.SOURCE(BASIC370). Near the bottom of the program, the defaults are set. Recompile the userid.BASIC370.SOURCE($COMPILE).

X. Sample Programs

A dataset containing sample programs (including the ones in the Figures section) can be found in 'userid.BASIC370.DATA'. To run them, submit the member 'userid.BASIC370.DATA($EXECUTE)'.

XI. Figures

1. Random Number Example

```
OFFSET
000001                  10 REM
000002                  11 REM   DEMO FOR RND FUNCTION
000003                  12 REM
000004                  20 PRINT "RND FUNCTION TEST"
000007                  30 FOR I=1 TO 20
000012                  40 PRINT RND(0)
000018                  50 NEXT I
000020                  60 END

**** END OF COMPILATION **** NO ERRORS FOUND

RND FUNCTION TEST
 0.404849
 0.428079
 0.924833
 0.696285
 0.854206
 0.858671
 0.464167
 0.05696
 0.16426
 0.472914
 0.359143
 0.898627
 0.159475
 0.869204
 0.779945
 0.856835
 0.1215
 0.017485
 0.011411
 0.911096


**** PROGRAM EXECUTION COMPLETE -      173 INSTRUCTIONS EXECUTED ****
```

Figure 1 - Random Number Program

2. Sample Looping, Printing and INT/INR

```
OFFSET

000001                  10 REM
000002                  20 REM  DEMO PROGRAM FOR BASIC/360
000003                  21 REM    DEMOS FOR..NEXT, PRINT, AND FUNCTIONS
000004                  30 REM
000005                  31 PRINT "I","I*I","SQR(I)","ABS(I)"
000014                  32 FOR I=1 TO 4
000019                  34 PRINT "=====",
000022                  35 NEXT I
000024                  36 PRINT
000026                  40 FOR I=1 TO 10
000031                  50 PRINT I,I*I,SQR(I),ABS(I)
000049                  60 NEXT I
000051                  70 PRINT
000053                  100 REM
000054                  101 PRINT
000056                  102 PRINT "J","K","INT(K)","INR(K)"
000065                  110 LET J=1
000068                  115 LET K=SQR(J)
000074                  120 PRINT J,K,INT(K),INR(K)
000089                  130 LET J=J+1
000095                  140 IF J<=10 THEN 115
000099                  150 REM
000100                  9000 END

**** END OF COMPILATION **** NO ERRORS FOUND

I             I*I           SQR(I)        ABS(I)
=====         =====         =====         =====
 1             1             1             1
 2             4             1.414213      2
 3             9             1.73205       3
 4             16            2             4
 5             25            2.236067      5
 6             36            2.449489      6
 7             49            2.64575       7
 8             64            2.828427      8
 9             81            3             9
 10            100           3.162277      10


J             K             INT(K)        INR(K)
 1             1             1             1
 2             1.414213      1             1
 3             1.73205       1             2
 4             2             2             2
 5             2.236067      2             2
 6             2.449489      2             2
 7             2.64575       2             3
 8             2.828427      2             3
 9             3             3             3
 10            3.162277      3             3


**** PROGRAM EXECUTION COMPLETE -      575 INSTRUCTIONS EXECUTED ****
```

Figure 2 Sample Looping, Printing and INT/INR use

3. Advanced Printing

```
OFFSET

000001                  10 REM
000002                  20 REM  VALUDATION PROGRAM FOR BASIC/360
000003                  21 REM    ADVANCED PRINTING
000004                  30 REM
000005                  34 PRINT
000007                  35 PRINT "I","I*I","SQR(I)","ABS(I)"
000016                  36 FOR I=1 TO 4
000021                  37 PRINT "======",
000024                  38 NEXT I
000026                  39 PRINT
000028                  40 FOR I=1 TO 10
000033                  50 PRINT I,I*I,SQR(I),ABS(I)
000051                  60 NEXT I
000053                  70 PRINT
000055                  100 REM
000056                  104 PRINT
000058                  105 PRINT "J","J*J","SQR(J)"
000065                  106 FOR I=1 TO 3
000070                  107 PRINT "======",
000073                  108 NEXT I
000075                  109 PRINT
000077                  110 LET J=1
000080                  120 PRINT TAB(J),J,J*J,SQR(J)
000098                  130 LET J=J+1
000104                  140 IF J<=10 THEN 120
000108                  9000 END

**** END OF COMPILATION **** NO ERRORS FOUND


I             I*I           SQR(I)        ABS(I)
======        ======        ======        ======
 1             1             1             1
 2             4             1.414213      2
 3             9             1.73205       3
 4             16            2             4
 5             25            2.236067      5
 6             36            2.449489      6
 7             49            2.64575       7
 8             64            2.828427      8
 9             81            3             9
 10            100           3.162277      10


J             J*J           SQR(J)
======        ======        ======
  1            1             1
   2           4             1.414213
    3          9             1.73205
     4         16            2
      5        25            2.236067
       6       36            2.449489
        7      49            2.64575
         8     64            2.828427
          9    81            3
           10  100           3.162277


**** PROGRAM EXECUTION COMPLETE -      566 INSTRUCTIONS EXECUTED ****
```

Figure 3 Advanced Printing Example

4. Print Using Sample 1

```
OFFSET

000001                   10 REM
000002                   20 REM   DEMO PRINT USING #1
000003                   30 REM
000004                   40 REM   A LOT OF POSSIBLE COMBINATIONS OF PRINT AND PRINT USING
000005                   50 REM   ARE SHOWN IN THIS PROGRAM.  THE OUTPUT IS A JUMBLE OF
000006                   60 REM   STUFF TO DEMONSTRATE THE INTERACTIONS OF PRINT AND PRINT
000007                   70 REM   USING.
000008                   80 REM
000009                   90 LET T$="  EMPLOYEE     HOURS    RATE      NET PAY"
000012                  100 LET U$="  --------      ----   -----    ---------"
000015                  110 LET P$=  "&&&&&&&&&&&&  ##.#   ##.##  $ ##,###.##"
000018                  120 PRINT T$
000021                  130 PRINT U$
000024                  140 READ NAME$,HOURS,RATE
000028                  150 IF HOURS>0 THEN 190
000032                  160 PRINT
000034                  170 PRINT "END OF JOB"
000037                  180 END
000039                  190 LET NET=HOURS*RATE
000045                  200 PRINT "->";
000048                  210 PRINT USING P$,NAME$,HOURS,RATE,NET,
000061                  220 PRINT USING P$,NAME$,HOURS,RATE,NET,
000074                  230 IF HOURS <= 40 THEN 260
000078                  240 PRINT "OVERTIME"
000081                  250 GO TO 140
000083                  260 PRINT
000085                  270 GOTO 140
000087                  280 DATA "MOE HOWARD",41,7.25
000088                  290 DATA "LARRY FINE",32,7.25
000089                  300 DATA "CURLEY HOWARD",40,.02
000090                  310 DATA "THE BIG BOSS MAN",40,14
000091                  320 DATA "END",0,0

**** END OF COMPILATION **** NO ERRORS FOUND

  EMPLOYEE      HOURS    RATE      NET PAY
  --------      ----   -----    ---------
->MOE HOWARD     41.0    7.25  $    297.25 MOE HOWARD    41.0    7.25  $    297.25   OVERTIME
->LARRY FINE     32.0    7.25  $    232.00 LARRY FINE    32.0    7.25  $    232.00
->CURLEY HOWAR   40.0    0.01  $      0.79 CURLEY HOWAR  40.0    0.01  $      0.79
->THE BIG BOSS   40.0   14.00  $    560.00 THE BIG BOSS  40.0   14.00  $    560.00

END OF JOB


**** PROGRAM EXECUTION COMPLETE -      235 INSTRUCTIONS EXECUTED ****
```

Figure 4 Print Using Sample 1

5. Print Using Sample 2

```
OFFSET

000001                   10 REM
000002                   20 REM   DEMO PRINT USING #2
000003                   30 REM
000004                   40 REM   A LOT OF POSSIBLE COMBINATIONS OF PRINT AND PRINT USING
000005                   50 REM   ARE SHOWN IN THIS PROGRAM.  THE OUTPUT IS A JUMBLE OF
000006                   60 REM   STUFF TO DEMONSTRATE THE INTERACTIONS OF PRINT AND PRINT
000007                   70 REM   USING.
000008                   80 REM
000009                   90 LET T$="  EMPLOYEE     HOURS    RATE      NET PAY"
000012                  100 LET U$="  --------      ----   -----    ---------"
000015                  110 LET P$=  "&&&&&&&&&&&&  ##.#   ##.##  $ ##,###.##"
000018                  120 PRINT T$
000021                  130 PRINT U$
000024                  140 READ NAME$,HOURS,RATE
000028                  150 IF HOURS>0 THEN 190
000032                  160 PRINT
000034                  170 PRINT "END OF JOB"
000037                  180 END
000039                  190 LET NET=HOURS*RATE
000045                  200 PRINT "->";
000048                  210 PRINT USING P$,NAME$,HOURS,RATE,NET;
000061                  220 PRINT USING P$,NAME$,HOURS,RATE,NET;
000074                  230 IF HOURS <= 40 THEN 260
000078                  240 PRINT "OVERTIME"
000081                  250 GO TO 140
000083                  260 PRINT
000085                  270 GOTO 140
000087                  280 DATA "MOE HOWARD",41,7.25
000088                  290 DATA "LARRY FINE",32,7.25
000089                  300 DATA "CURLEY HOWARD",40,.02
000090                  310 DATA "THE BIG BOSS MAN",40,14
000091                  320 DATA "END",0,0

**** END OF COMPILATION **** NO ERRORS FOUND

  EMPLOYEE     HOURS    RATE      NET PAY
  --------      ----   -----    ---------
->MOE HOWARD    41.0    7.25  $    297.25MOE HOWARD    41.0    7.25  $    297.25OVERTIME
->LARRY FINE    32.0    7.25  $    232.00LARRY FINE    32.0    7.25  $    232.00
->CURLEY HOWAR  40.0    0.01  $      0.79CURLEY HOWAR  40.0    0.01  $      0.79
->THE BIG BOSS  40.0   14.00  $    560.00THE BIG BOSS  40.0   14.00  $    560.00

END OF JOB


**** PROGRAM EXECUTION COMPLETE -      235 INSTRUCTIONS EXECUTED ****
```

Figure 5 Print Using Sample 2

6. Sample of a List

```
OFFSET

000001                  1 REM
000002                  2 REM  DEMONSTRATE DIM AND SUBSCRIPTED VARIABLES
000003                  3 REM
000004                  10 DIM I(10)
000005                  20 DIM J(10),K(10)
000006                  100 FOR X=1 TO 10
000011                  110 LET I(X)=X
000017                  120 LET J(X)=X*X
000026                  130 LET K(X)=SQR(X)
000035                  140 NEXT X
000037                  150 FOR X=10 TO 1 STEP -1
000045                  160 PRINT I(X),J(X),K(X)
000064                  170 NEXT X
000066                  180 END

**** END OF COMPILATION **** NO ERRORS FOUND

 10            100           3.162277
 9             81            3
 8             64            2.828427
 7             49            2.64575
 6             36            2.449489
 5             25            2.236067
 4             16            2
 3             9             1.73205
 2             4             1.414213
 1             1             1


**** PROGRAM EXECUTION COMPLETE -      490 INSTRUCTIONS EXECUTED ****
```

Figure 6 Sample of a list

7. Example using *APPEND

```
OFFSET

000001                  *APPEND=TRIGPLOT
000001                  10 REM
000002                  20 REM   DEMO APPEND - PLOT SIN CURVE
000003                  30 REM
000004                  40 DEF FNX(X)=SIN(X)
000013                  50 LET T$="SIN(X)"
000016                  60 GOSUB 10000
000018                  70 PRINT "BACK"
000021                  180 END
000023                  *REM APPENDED CODE
000023                  10000 REM -------------------------------------------------------------
000024                  10010 REM
000025                  10020 REM   SUBPROGRAM PLOT A TRIG FUNCTION USING CODE FROM A LIBRARY
000026                  10030 REM
000027                  10040 REM   TO USE, DEF FNX(X) TO THE TRIG FUNCTION
000028                  10050 REM   AND SET T$ TO THE TITLE OF THE PLOT
000029                  10060 REM   AND THEN GOSUB TO THE FIRST LINE OF THIS CODE
000030                  10070 REM
000031                  10080 PRINT "X";TAB(68);T$
000041                  10090 REM
000042                  10100 FOR X=0 TO 6.28 STEP .1
000047                  10110 LET Y=FNX(X)
000053                  10120 LET Y2=Y*40+70
000062                  10130 PRINT X,Y2;
000067                  10140 IF Y2>70 THEN 10180
000071                  10150 IF Y2<70 THEN 10200
000075                  10160 PRINT TAB(70);"*"
000083                  10170 GOTO 10210
000085                  10180 PRINT TAB(70);"|";TAB(Y2);"*"
000100                  10190 GOTO 10210
000102                  10200 PRINT TAB(Y2);"*";TAB(70);"|"
000117                  10210 NEXT X
000119                  10220 RETURN

**** END OF COMPILATION **** NO ERRORS FOUND

X                                                                       SIN(X)
 0             70                                                         *
 0.099999      73.993331                                                  |  *
 0.199999      77.946762                                                  |      *
 0.299999      81.8208                                                    |          *
 0.399999      85.576721                                                  |              *
 0.499999      89.177001                                                  |                  *
 0.599999      92.585678                                                  |                     *
 0.699999      95.768692                                                  |                        *
 0.799999      98.694229                                                  |
{output truncated}
```

Figure 7 Example using *APPEND

8. Using *SAVE

```
OFFSET

000001                  *APPEND=TRIGPLOT
000001                  *SAVE=MYPROG
000001                  10 REM
000002                  20 REM   DEMO APPEND - PLOT SIN CURVE
000003                  30 REM
000004                  40 DEF FNX(X)=SIN(X)
000013                  50 LET T$="SIN(X)"
000016                  60 GOSUB 10000
000018                  70 PRINT "BACK"
000021                  180 END
000023                  *REM APPENDED CODE
000023                  10000 REM -------------------------------------------------------------
000024                  10010 REM
000025                  10020 REM   SUBPROGRAM PLOT A TRIG FUNCTION USING CODE FROM A LIBRARY
000026                  10030 REM
000027                  10040 REM   TO USE, DEF FNX(X) TO THE TRIG FUNCTION
000028                  10050 REM   AND SET T$ TO THE TITLE OF THE PLOT
000029                  10060 REM   AND THEN GOSUB TO THE FIRST LINE OF THIS CODE
000030                  10070 REM
000031                  10080 PRINT "X";TAB(68);T$
000041                  10090 REM
000042                  10100 FOR X=0 TO 6.28 STEP .1
000047                  10110 LET Y=FNX(X)
000053                  10120 LET Y2=Y*40+70
000062                  10130 PRINT X,Y2;
000067                  10140 IF Y2>70 THEN 10180
000071                  10150 IF Y2<70 THEN 10200
000075                  10160 PRINT TAB(70);"*"
000083                  10170 GOTO 10210
000085                  10180 PRINT TAB(70);"|";TAB(Y2);"*"
000100                  10190 GOTO 10210
000102                  10200 PRINT TAB(Y2);"*";TAB(70);"|"
000117                  10210 NEXT X
000119                  10220 RETURN

**** END OF COMPILATION **** NO ERRORS FOUND

X                                                                       SIN(X)
 0             70                                                         *
 0.099999      73.993331                                                  |  *
 0.199999      77.946762                                                  |      *
 0.299999      81.8208                                                    |          *
 0.399999      85.576721                                                  |              *
 0.499999      89.177001                                                  |                  *
 0.599999      92.585678                                                  |                     *
 0.699999      95.768692                                                  |                        *
 0.799999      98.694229                                                  |
{output truncated}
```

Figure 8 Example using *SAVE

9. Using INPUT Statement

```
OFFSET

000001                  10 REM TEST INPUT STATEMENT
000002                  20 PRINT "CHANGE MAKER FOR USA"
000005                  21 DIM COIN$(5),CENTS(5)
000006                  22 FOR I=1 TO 5
000011                  23 READ COIN$(I),CENTS(I)
000020                  24 NEXT I
000022                  30 PRINT
000024                  40 PRINT "HOW MUCH IS THE PURCHASE?"
000027                  50 INPUT PUR
000029                  60 PRINT "HOW MUCH WAS TENDERED?"
000032                  70 INPUT TEN
000034                  80 LET CHG=TEN-PUR
000040                  90 LET PU$="PURCHASE #,###.##  TENDERED #,###.##  CHANGE #,###.##"
000043                  100 PRINT USING PU$,PUR,TEN,CHG
000054                  110 IF CHG < 0 THEN 1000
000058                  120 IF CHG = 0 THEN 900
000062                  130 REM CONVERT CHG TO ALL CENTS DUE TO FLOATING POINT INACCURACIES
000063                  140 LET CHG=INT(CHG*100)
000072                  150 FOR I=1 TO 5
000077                  160 LET NC=INT(CHG/CENTS(I))
000091                  170 PRINT USING "### &",NC,COIN$(I)
000105                  171 IF NC=0 THEN 190
000109                  180 LET CHG=CHG-(NC*CENTS(I))
000123                  190 NEXT I
000125                  200 GOTO 2000
000127                  900 PRINT "NO CHANGE DUE"
000130                  910 GOTO 2000
000132                  1000 PRINT USING "SHORT #,###.##",CHG
000139                  2000 PRINT "MAKE MORE CHANGE (Y/N)?"
000142                  2010 INPUT R$
000144                  2020 IF R$="Y" THEN 30
000148                  2030 IF R$="N" THEN 9999
000152                  2040 PRINT "PLEASE ENTER Y OR N"\GOTO 2000
000156                  8000 DATA "DOLLAR(S)",100
000157                  8010 DATA "QUARTER(S)",25
000158                  8020 DATA "DIME(S)",10
000159                  8030 DATA "NICKLE(S)",5
000160                  8040 DATA "PENNIES",1
000161                  9999 END

**** END OF COMPILATION **** NO ERRORS FOUND
****   162 INSTRUCTIONS GENERATED,    53 SYMBOLS DEFINED    10 DATA ITEMS DEFINED ****

CHANGE MAKER FOR USA

HOW MUCH IS THE PURCHASE?
1.68 ENTERED VIA INPUT
HOW MUCH WAS TENDERED?
10 ENTERED VIA INPUT
PURCHASE     1.67  TENDERED    10.00  CHANGE     8.32
  8 DOLLAR(S)
  1 QUARTER(S)
  0 DIME(S)
  1 NICKLE(S)
  2 PENNIES
MAKE MORE CHANGE (Y/N)?
Y ENTERED VIA INPUT

HOW MUCH IS THE PURCHASE?
9.99 ENTERED VIA INPUT
HOW MUCH WAS TENDERED?
20 ENTERED VIA INPUT
PURCHASE     9.98  TENDERED    20.00  CHANGE    10.01
 10 DOLLAR(S)
  0 QUARTER(S)
```

```
  0 DIME(S)
  0 NICKLE(S)
  1 PENNIES
MAKE MORE CHANGE (Y/N)?
N ENTERED VIA INPUT


**** PROGRAM EXECUTION COMPLETE -      615 INSTRUCTIONS EXECUTED ****
```

Figure 9 Example Of Input Statements

10. Get/Put File I/O Example

```
OFFSET

000001                  10 REM GET/PUT TEST
000002                  15 LET X$="SOMETHING DIFFERENT."
000005                  20 FOR X=1 TO 100
000010                  21 LET Y=X*X
000016                  22 LET Z=SQR(X)
000021                  30 PUT X,Y,Z,X$
000026                  35 NEXT X
000028                  40 END

**** END OF COMPILATION **** NO ERRORS FOUND
****    29 INSTRUCTIONS GENERATED,    20 SYMBOLS DEFINED     0 DATA ITEMS DEFINED ****


**** PROGRAM EXECUTION COMPLETE -     1811 INSTRUCTIONS EXECUTED ****


OFFSET

000001                  10 REM GET/PUT TEST
000002                  20 FOR I=1 TO 100
000007                  30 GET X,Y,Z,X$
000012                  40 PRINT X$,X,Y,Z
000021                  50 NEXT I
000023                  60 END

**** END OF COMPILATION **** NO ERRORS FOUND
****    24 INSTRUCTIONS GENERATED,    19 SYMBOLS DEFINED     0 DATA ITEMS DEFINED ****

SOMETHING DIFFERENT.         1             1             1
SOMETHING DIFFERENT.         2             4             1.414212
SOMETHING DIFFERENT.         3             9             1.732049
SOMETHING DIFFERENT.         4             16            2
SOMETHING DIFFERENT.         5             25            2.236066
SOMETHING DIFFERENT.         6             36            2.449488
SOMETHING DIFFERENT.         7             49            2.645749
SOMETHING DIFFERENT.         8             64            2.828426
SOMETHING DIFFERENT.         9             81            3
SOMETHING DIFFERENT.         10            100           3.162276
SOMETHING DIFFERENT.         11            121           3.316623
SOMETHING DIFFERENT.         12            144           3.4641
SOMETHING DIFFERENT.         13            169           3.60555
SOMETHING DIFFERENT.         14            196           3.741656
SOMETHING DIFFERENT.         15            225           3.872981
SOMETHING DIFFERENT.         16            256           4
SOMETHING DIFFERENT.         17            289           4.123105
SOMETHING DIFFERENT.         18            324           4.242639
SOMETHING DIFFERENT.         19            361           4.358898
SOMETHING DIFFERENT.         20            400           4.472134
SOMETHING DIFFERENT.         21            441           4.582574
SOMETHING DIFFERENT.         22            484           4.690414
SOMETHING DIFFERENT.         23            529           4.79583

(output truncated)
```