Extended BASIC User's Manual

MANUAL REVISION HISTORY

Revision	Date	Change
A		Original Release
B	3/80	Effective: Version 13 of 6809 Extended Basic Version 11 of 6800 Extended Basic P. 85-86 Add errors 37 and 72, remove error 95. p. 62 Remove restriction on only one virtual array reference per expression. p. 64 Add statement on closing and reopening virtual array files. p. 38 Correct documentation of DPOKE. p. 45 Correct documentation of DPEEK. P. 55 Add documentation of error 37.
C	6/80	Effective: Version 14 of 6809 Extended Basic Version 12 of 6800 Extended Basic p. 85 Enhance scope of error 45 p. 50 Document maximum size of a line that may be read from a file. P. 27 Document effect of TTYSET width value.

CONTENTS PAGE

1. INTRODUCTION

Technical Systems Consultants' EXTENDED BASIC is a very fast and Complete BASIC. It is used like most BASIC interpreters in that lines are entered from the keyboard to build the program and the resulting program may be run at anytime by using the RUN command. It contains all of the normal interactive features of BASIC including a direct execution mode which allows BASIC to be used as a calculator.

All lines entered into a BASIC program must begin with a line number. All lines are automatically put in numerical sequence which allows for simple editing. Lines which already exist in a program may be deleted by simply typing the line number of the line followed by a carriage return. It is recommended that the user read this entire manual before attempting to use Technical Systems Consultants' EXTENDED BASIC. It is assumed that the reader is familiar with BASIC programming, so detailed programming examples are not given.

2. DEFINITIONS

Several terms which will be used frequently in this manual are defined below to avoid any confusion in their meaning.

a. COMMAND — A Command is an instruction for BASIC to immediately perform some specific operation. A Command is usually issued after the user has received the 'READY' prompt from BASIC. When this prompt appears on the screen, the computer is "READY and WAITING" for the user to tell it what to do.

b. LINE — In BASIC, each line begins with a line number and ends with a carriage return. A line may contain only a single statement or may consist of several statements separated by colons (:) or backslashes (/).

c. PRINT — BASIC prints on the CRT.

d. STATEMENT — A Statement is an instruction for the BASIC interpreter. A Statement is usually executed by BASIC as it is encountered in a program that is being executed. In contrast to this, a Command is executed outside of a BASIC program.

e. TYPE — The user types on the keyboard.

f. ^C — This is notation for CONTROL C. It is typed by holding down on the "CONTROL" key and keeping it pressed while you strike the "C" key. This entry is used to interrupt BASIC as it is performing some operation. It can, for instance, be used to interrupt a program that is currently being executed. If this is done, then the statement that is in the process of being executed will be completed; then BASIC returns to the command level to await further instruction.

G.^H — CONTROL H is typed to erase the last character typed in. If this is typed when the cursor is at the beginning of a line, then the result will be a Carriage Return Line Feed.

h. ^X — CONTROL X deletes the line currently being input and performs a Carriage Return Line Feed.

i. CRLF — This represents Carriage Return and Line Feed characters which cause a new line to be started.

3. CONVENTIONS

To make the definitions of BASIC statements and commands more easily understood, several conventions will be used throughout this manual.

The statement or command being described, and any other which is used in the definition, will be printed in capital letters. Angle brackets (<>) will be used to enclose essential components of the statement. Square brackets ([]) will surround optional components. Once again the types of special enclosures are:

```
<essential element>
[optional element]
```

4. FUNDAMENTALS OF BASIC

4.1 Lines

Each line of a BASIC program begins with a LINE NUMBER. The line may contain one or more statements separated by colons or backslashes and is terminated by a CARRIAGE RETURN. The length of a line can not exceed 127 characters. Lines may be numbered from 1 through 32767 and each one must have a unique number. When a program is executed by BASIC, it starts with the smallest line number and progresses toward the largest.

When writing programs it is usually wise to number lines in increments of 10, 20, or more so that additional ones may be added during program debugging or modification. Spaces may be used as desired to make a program easier to read. Examples of these features are given below. Note especially that statements 30 and 40 are equivalent except that line 40 may be a little easier to read.

Spaces:

```
30 X=32*X/3+7.3*X/2+6.54*X-.1
40    X = 32*X/3 + 7.3*X/2 + 6.54*X -.1
```

multiple statements on a line:

```
120 INPUT "SPEED,TIME";S,T:D=S*T:PRINT "DIST=";D
300 A=310*X : B=K-A:C=A*2.9/11:PRINT "CF=";A;B;C
```

4.2 Constants

4.2.1 Floating Point Constants

All floating point values are internally represented with a seven byte (56 bit) signed magnitude mantissa and a single byte (8 bit) excess 128 notation exponent. Externally this is approximately a 17 decimal digit mantissa with a dynamic range of 10**38.

Often a more convenient form of representing floating point numbers is scientific notation. That is the signed mantissa followed by 'E'. Or 'e' and the signed exponent. On conversion to decimal, BASIC automatically converts any number to scientific notation if its magnitude is greater than or equal to 1E6 or less than 1E2. Some examples of floating point numbers are:

98.345	-1.23E+6
3.14159265	1E04
0.000001E6	1.0

It should be noted, for a constant to be internally represented in floating point format it must (1) contain a decimal point or (2) be in scientific notation or (3) be too large to be represented in an integer format. If none of the above conditions are met then the constant will be converted to an integer. This can be very beneficial for programs with many constants both in terms of storage requirements and speed of execution.

4.2.2 INTEGER CONSTANTS

All integers are internally represented as a 16 bit two's complement number with a decimal range of -32768 to 32767. Some examples of integer constants are:

100	32000
-12	1

4.2.3 STRING CONSTANTS

Another type of constant is the character string. It is different from the other constants both in the way it is defined and in the way it is usually used. Character string constants will probably be seen most often in PRINT and INPUT statements. String constants are defined by placing any ASCII character or group of characters (a string) between single or double quotes. A string may be of any length from 0 to 32767

characters long. Some examples of strings are:

```
Strings:        "WHAT IS YOUR NAME"
                "'FRIDAY'"    (string includes single quotes)
                '"REALLY"'    (string includes double quotes)
                "H"           (single element string)
                ""            (null string)
```

4.3 Variables

A variable is an item of data that may take on different values. For example, it can be assigned a value by the programmer and later be changed by the program during execution. The three types of variables are "floating point", "integer", and "character string". A "floating point" variable name can consist of a single alphabetic letter or a letter followed by another letter or single digit. Examples of names to define floating point variables are A, K, G9, E1, or XX.

The second type of variable is the integer variable. It is defined by following any "floating point" variable name with a percent sign (%). Using the same five variables as before, we could define them to be integer variables by writing them as A%, K%, G9%, E1%, or XX%.

The third type of variable is the string variable. It is defined by following any "floating point" variable name with a dollar sign ($). Using the same five variables as before, we could have defined them to be string variables by writing them as A$, K$, G9$, E1$, or XX$.

Any of the variable types may be subscripted (dimensioned) and this will be described in the next section. The same variable name can be used in a program for a floating point variable, a string variable or an integer variable. For example, the variables "A" and "A$" listed above could both be used in the same program because they are considered different variables since one is a floating point and the other is a string variable.

It should be noted that some combinations of double letters intended to be used as variables are not valid. They are keywords in BASIC and are reserved. They are: AS, FN, IF, ON, OR, PI, and TO.

4.4 Dimensioning

Any type of variable defined in section 3.3 can be subscripted (dimensioned) using the DIM statement. A variable is dimensioned in a DIM statement by following the variable name with an integer that is enclosed in parentheses or two integers that are also enclosed in parentheses but separated by a comma. To use a subscripted variable you

write it just as it appeared in the dimension statement except that you may use any legal expression to describe the subscripts. Suppose a one dimensional array has been created in a program with the statement:

```
100 DIM X(4)
```

This will create 5 new variables that are called:

X(0), X(1), X(2), X(3), and X(4)

They may be used in program operations just as they are written above or the "subscript" could be a variable such as X(A) where "A" has a value of 0, 1, 2, 3, or 4. The subscript could also be an expression such as X(A+2). When this is encountered in a program, the subscript expression is evaluated using the current value of "A". BASIC will also support two-dimensional arrays. A two-dimensional array defined by the statement:

```
30 DIM X(3,2)
```

specifies a four by three element matrix. This matrix has the following elements:

```
X(0,0) X(0,1) X(0,2)
X(1,0) X(1,1) X(1,2)
X(2,0) X(2,1) X(2,2)
X(3,0) X(3,1) X(3,2)
```

Just as with the one-dimensional array, this can also be used in a program with subscripts that are expressions.

The same variable name can be used to specify a non-dimensioned and a dimensioned array. BASIC will consider these to be separate but one-dimensional and two-dimensional arrays cannot share the same name. All dimensioned variables must be declared in a DIM statement before they can be referenced.

4.5 Assignment

Variables are assigned values by the LET, INPUT, or by the combination of the READ and DATA statements. These statements will be explained more thoroughly in sections 6.1 and 6.4 but they need to be introduced now to get a feel for the BASIC language.

Most variables are assigned numeric or string values using the LET statement. For example the statement:

```
10 LET X=2
```

assigns a floating point value of "2." to X even though the constant "2" is internally an integer.

An INPUT statement such as this one:

```
240 INPUT P1
```

causes the computer to print a question mark and then wait for the user to type in something. The data the user types will be assigned to the variable P1.

READ and DATA statements must be used together. Briefly, a READ statement such as:

```
100 READ K
```

will assign a new value to K each time it is encountered in the program. The first execution of this statement will assign to K the first piece of data from the first DATA statement of the program. The second execution of this READ statement will assign the second piece of data in the first DATA statement to K, and so on. After all the data has been read from the first DATA statement, reading continues at the next one and then goes through all those appearing in the program. If a READ is attempted after the program has read to the end (already read the last piece of data from the last DATA statement), then error number 31 will be issued to warn of this. A DATA statement such as this one might be used:

```
500 DATA 1,3.14,6.02
```

4.6 Operators

There are three different classes of operators available. The class of operators most familiar to everyone is that of the Mathematical Operators. This is comprised of addition, subtraction, multiplication, division, and exponentiation.

The second class is the Logical Operators. They are used to perform bit by bit operations on integer quantities and are used extensively in conditional tests and for masking. Since they operate on integer quantities the internal "floating point" representation of some variables and constants must first be converted to integer. Then the operation can be performed. These conversions are done automatically by the BASIC interpreter.

The remaining operators are called Relational Operators. They are also used in conditional tests. They may be used in an IF statement for example to determine if one quantity is greater than another. In the case of trying to "relate" a floating point number and an integer, the integer is first converted to floating point and then the relation is

performed. Each of the three classes of operators will be described separately.

4.6.1 Mathematical Operators

MATHEMATICAL OPERATORS:

SYMBOL	EXAMPLE	MEANING
+	X+Y	Add X and Y
-	X-Y	Subtract Y from X
*	X*Y	Multiply X and Y
/	X/Y	Divide X by Y
^	X^Y	X to the Yth power
^	X^Y%	X to the Yth power

When an arithmetic expression containing several of these symbols is to be evaluated, it is processed using the following priority scheme.

1. Exponentiation
2. Unary Minus
3. Multiplication and Division
4. Addition and Subtraction

This means that when BASIC is evaluating an expression containing a mixture of mathematical operators, it will first do the exponentiation, then take into account any unary minus signs (such as -3.4 or -A). Next it will do multiplications and divisions then, last of all, it does additions and subtractions. When signs of equal priority are encountered, it does the left one first since BASIC evaluates expressions from left to right. This order can be altered by the use of parentheses. BASIC evaluates quantities in parentheses first and, in the case of nested parentheses, it starts with the innermost set and works its way out. They can and should be used anytime there is a doubt about how the expression will be evaluated.

When two values of the same type are to be operated on, the appropriate operator is called. For example, two integers are to be multiplied, then the integer multiply routine is called. If two floating point values are to be subtracted, then the floating point subtract routine is called. When an integer and a floating point value are to be operated on, the integer is first converted to floating point and then the operation is performed. The exponentiation operator is an exception to these rules and one of two things can happen. First of all, if the "base" (base ^ power) is an integer, it is converted to

floating point. Note that the "^" operator always returns a floating point result. Next if the "power" is an integer then a fast algorithm is called that "essentially" performs repeated multiplication. On the other hand if the "power" is a floating point value then the following algorithm is used: X^Y = EXP(Y * LOG(X)). In the latter case, X cannot be negative because the logarithm of a negative number does not exist.

4.6.2 Logical Operators

When Logical Operators are used on one or two numbers they perform the desired operation on the corresponding bits of the number or numbers. If, for example, we assume A and B are equal to the following binary quantities:

```
A=(1100101111110110)
B=(0111010111100100)
```

Then:

```
NOT A  =(0011010000001001)
A AND B=(0100000111100100)
A OR B =(1111111111110110)
```

It can be seen that these are bit-by-bit operations. These operators, when used like this, operate on one or two numbers to give a single numerical result.

The logical operators have a totally different effect when they are used in an expression that is the test condition of an IF-THEN statement. In this case the expression is being logically evaluated (not arithmetically evaluated) to see if it is true or false. An expression that is evaluated and determined to be true has a non-zero value and one that is determined to be false has a value of "0". A statement such as:

```
22 IF A>0 AND A<10 THEN GO TO 40
```

will branch to statement 40 if and only if A is between 0 AND 10. The logical operator "AND" specifies that the condition "A>0" be true AND the condition "A<10" also be true. The following is a list of the available operators.

LOGICAL OPERATORS:

SYMBOL	EXAMPLE	MEANING
NOT	NOT X	When operating on integers, this operator simply switches each bit in the binary representation with its

		complement (1's are replaced with O's and O's are replaced with 1'S).
AND	X AND Y	The result of this is to assign each bit of the result a " 1 " if each of the corresponding bits of the two arguments is a "1". When "AND" is used in a conditional test, then both X AND Y in the example must be true for the logical AND of them to be true.
OR	A OR B	This leaves each bit of the resulting integer a "1" if either of the corresponding bits of A OR B is a one. When used in a conditional test, the test will yield true if A OR B is true. With each of these three operators, the integer is automatically converted back into a "real" number.

4.6.3 Relational Operators

As the name implies, this group of operators tests the relation of variables to other variables or constants. The six relational symbols recognized by BASIC are:

RELATIONAL OPERATORS:

SYMBOL	EXAMPLE	MEANING
=	X=Y	X is equal to Y
<>	X<>Y	X is not equal to Y
<	X<Y	X is less than Y
>	X>Y	X is greater that Y
<=	X<=y	X is less than or equal to Y
>=	X>=y	X is greater than or equal to Y

These are often combined with the Logical Operators to perform complex tests. The statement:

```
660 IF A=0 OR (C<127 AND D <> 0) GO TO 100
```

will cause a branch to statement 100 if A is equal to zero OR if both C

is less than 127 AND D is not equal to zero.

4.6.4 String Operators

The string operators consist of the concatenation operator ('+') and the relational operators. The '+' operator will concatenate two strings (join them together) to form a new string. The relational operators, when applied to string operands, indicate alphabetic sequence. If one string is "less than" another, it implies it would appear before the other if sorted into alphabetical order. In any string comparison, trailing blanks are ignored. If two strings of unequal length are compared, the shorter string is padded with trailing spaces to make it equal in length to the other. A string of zero length (null string) is considered to be completely blank and is less than any string of length greater than zero unless the string is all spaces, then the two are considered equal. All of the standard arithmetic relational operators may be used in connection with strings.

4.6.5 Operator Precedence

The overall operator precedence is shown in the table below. The operator at the top of the list has highest precedence, while the one at the bottom has lowest. Operators of equal precedence are evaluated left to right.

1.	()	Expressions enclosed in parenthesis
2.	^	Exponentiation
3.	-	Unary minus
4.	* /	Multiplication and division
5.	+ -	Addition and subtraction
6.		Relational operators
7.	NOT	The NOT operator
8.	AND	The AND operator
9.	OR	The OR operator

4.7 Mode

There are two different modes in which BASIC can function. The one referenced most often to this point is that in which you use the RUN command to execute a program that has been typed in. The other mode is the Immediate Mode. In the Immediate Mode you can type in a command or statement without a line number and the computer will immediately

execute it. In contrast to this, the normal running of a program starts at the statement with the smallest line number and progresses, executing statements, toward the largest numbered statement. BASIC distinguishes between these two types simply by the presence or absence of a line number. So, for instance, if you had typed in the statement:

```
100 PRINT "Sunday"
```

nothing would happen after you hit the carriage return. This is because BASIC assumes that this is part of a program you are writing and it will save and execute it only in response to a RUN command. If, on the other hand, you type:

```
PRINT "Monday"
```

this would be executed immediately because there is no statement number.

The two types of BASIC instructions (commands and statements) cross paths because of the immediate mode. Statements can be typed in without line numbers and used as if they were a command. This is done after BASIC has printed "READY" and is waiting for you to give it instructions.

4.8 Remarks

It is good programming practice to use remarks freely throughout your programs. This makes them easier for others to understand and will help you too when picking up one of your own programs that you haven't worked with for awhile. Remarks can be placed in a program by using the REMARK (can be typed REM for short) statement. When BASIC comes to a REM statement it ignores whatever follows. This statement must have a line number and the number can be referenced by the program (for example, by a GOTO statement).

5. COMMANDS

The following is a list of the commands that are available to the user. These commands are not used in the actual code of BASIC programs but are instructions to the computer. They are to be used when BASIC is at the command level, which is after it has printed "READY". When a command is typed into the computer, it causes action to be taken immediately.

NAME	EXAMPLE / EXPLANATION

CLEAR

CLEAR

The CLEAR command has the effect of setting all the variables in a program to zero. This operation is automatically performed when a RUN is executed.

COMPILE

COMPILE "LEDGER"

The COMPILE command is used to save a program on disk in a compiled form. The file name should be specified in quotes and should be in standard FLEX form (drive.name.ext). The drive will default to the working drive and the extension defaults to 'BAC' (BAsic Compiled). The resultant saved program will in most cases be smaller than the same program saved using the SAVE command. A compiled program can not be LOADed, LISTed, or edited. It can only be run by using the RUN "name" command (see RUN). The COMPILE feature should be used to save completed programs since they usually require less disk space and always load faster. Keep in mind that a compiled program can only be RUN, and any attempt to edit or LIST it once brought into memory will result in an error 64.

CONT

CONT

The CONTINUE command is used to restart a program after it has been stopped by either a STOP statement or a CONTROL/C. The STOP statement may have occurred anywhere in the program but the ^C would have had to been typed while the program was waiting for input at an INPUT statement. The command can not restart your program if you got an error during program execution or if you type in more program lines after you have stopped. If the program was stopped by a STOP statement, then CONT will cause he program to continue at

the first statement following the STOP. If the program has been stopped in an INPUT statement by a ^C, then the CONT command will cause execution to resume at the INPUT statement.

EXIT EXIT

This command is used to EXIT BASIC and enter the system monitor.

FLEX FLEX

The FLEX command is used to exit BASIC and return to the FLEX disk operating system. This is the normal BASIC exit method.

LIST
```
LIST              (list entire program)
LIST 10           (list line 10)
LIST 50-80        (list lines 50 through 80)
```

LIST can be used to display lines of a program. As the examples demonstrate, it can display all of the program or only specific lines. When a program is being listed the output may be terminated by typing ^C (control C). This is another way to display only part of a program.

LOAD LOAD "ORDER"

The LOAD command is used to load a text type file into BASIC from disk. The file name should be in quotes and in the standard FLEX form (drive.name.ext). The name defaults to the working drive and to a BAS (BAsic Source) extension. Standard FLEX text files (such as from the TSC Editor) may be loaded.

NEW NEW

When the NEW command is executed, it deletes the current program. After executing this command, you are ready to start typing in a "NEW" program.

RUN RUN

The RUN command instructs the computer to begin execution of the current program. When you RUN a program, all variables are initialized to zero and DATA statements are restored.

RUN RUN "LEDGER"

This form of the RUN command is used to load and execute a compiled BASIC program from disk. The file name must be in quotes and in standard FLEX form (drive.name.ext). The name defaults to the working drive and to a BAC extension. This is the only way a compiled program may be loaded from disk.

SAVE SAVE "ORDER"

SAVE is used to store the source form (text form) of a BASIC program on disk. The file name should be specified in quotes and in the standard FLEX form (drive.name.ext). The drive defaults to the working drive and the extension defaults to BAS. Any BASIC programs saved using the SAVE command may be manipulated by any FLEX program which works with text files (such a the EDITOR). IMPORTANT NOTE: The SAVE command will delete any existing file of the same name specified without any warning!

SCALE SCALE (print current scale factor)
SCALE 3 (set scale factor to 3)

The SCALE command specifies to BASIC the number of digits to the right of the decimal place that are to be preserved. A scale factor of zero turns off the scaling feature. The maximum scale factor is six (6).

With the scale factor set to non-zero, BASIC scales all floating point values by 10**scale factor and rounds to a whole number. In effect all floating point numbers become integers; the fractional parts disappear. The not-so-obvious advantage is after many floating point operations, round off error does not become significant because internally all numbers are whole. Perhaps a short example will help clear this up.

```
10 FOR I%=1 TO 10000
20   F = 0.01 + F
30 NEXT I%
40 PRINT 100.0 - F
99 END
```

```
RUN

 1. 77813319624E-1 2

READY
```

If we SAVE the program, set the SCALE factor to at least 2 (if we set it to 1 the constant 0.01 will be scaled to 0) LOAD the program back in and run.

```
RUN

 0

READY
```

Let's say the scale factor was set to 2, then the constant '0.01' was converted internally to 0.01 * 10^2 = 1.0. Repeatedly adding 1.0 does not generate any round off error since 1.0 can be EXACTLY represented in binary floating point notation.

BASIC converts all constants to their binary equivalent when the program is typed in or LOADed it is not possible to change the scale factor while the program is in memory. The same is true for any COMPILED files. When the binary is written to the disk, the current scale factor is also written out. When the program is later run the scale factor is read in automatically.

TROFF TROFF

TROFF turns the trace feature off.

TRON TRON

The TRON command turns trace on. Trace is used to debug programs and will print the line number in brackets of each line executed. TROFF or NEW turns trace off.

\+ +RENUMBER

The "+" command tells BASIC to send the rest of the command line to FLEX. This is a dangerous command in that some FLEX

utilities load in the same area of memory as BASIC. If a FLEX utility is executed from BASIC in this way, be certain that the utility loads into the utility command space in FLEX. The main use for this command is to invoke the RENUMBER utility which is supplied with BASIC. This utility loads into the utility command space in FLEX and will renumber BASIC programs in memory. See the section titled RENUMBER for more details.

6. STATEMENTS

All the BASIC statements listed below are arranged in groups that have similar functions or purposes. Appearing to the right of each statement is an expression showing its complete usage. This will be followed by one or more examples to demonstrate a typical use. Then, last of all, the definition and explanation of the statement appears.

6.1 Assignment

NAME — FORM / EXAMPLE / EXPLANATION

DATA

```
<line number> DATA <number> [,<number>,<number>,...]
                   <string> [,<string>,<string>,...]
```

```
50 DATA -3.556E-5,0,2,4.59E11
60 DATA APRIL, MAY, THATS ALL
70 DATA "   100"," 1000","10,000"
```

The DATA statement specifies information that will be read in by the program. The data is read in from left to right and, of course, begins with the first piece of data listed. Each time a READ statement is encountered in the program, the next item in the data list is read. The READ statements will begin taking data from the first DATA statement that appears in the program and, when it has read all that is in this statement, it will drop down to the next data statement and so on. If, for example, we assume we're running a program containing statement 50, the first time a READ statement is executed, the value read will be -3.556E-5. The next value read will be 0 and this continues through this statement and to the following DATA statements in the program. The DATA statement cannot appear in multiple statement lines. If a string is needed that contains an embedded comma or it is preceded by a space or spaces then it must be enclosed in quotes as in statement 70 above.

LET

```
<line number> LET <variable>=<expression>
```

```
10  LET X=3.5
25  LET H1=27.2*H1/(5.4E7-X)
70  LET DA$="MONDAY"
75  LET X(5,J)=0              (dimensioned variable)
80  Y=12.314                  (implied LET)
90  Y%=Y%+1
```

```
95 Z=Y%                    (type conversion)
96 Y%=EXP(F)
```

The LET statement assigns a value to a variable. Any variable can be assigned a value using this statement. The value can be a constant as in statement 10 or may be a complex expression as in statement 25. Notice that statements 80 - 95 are missing the word "LET". This is no accident but is an implied LET. This is a convenience feature of this BASIC interpreter, and any LET statement can be written this way. In line 95 note that the assignment variable is a different type than that of the resulting expression. In this particular example the integer expression is converted to floating point and the assignment is performed. In line 96 just the inverse is performed. The floating point expression is converted to an integer and then the assignment is performed.

READ <line number> READ <variable> [,<variable>,<variable>,...]

```
200 READ V,A1,CC
```

The READ statement is used to read data from a DATA statement. This has been explained in the above definition of the DATA statement for the case of a single variable argument. The READ statement can also be used with more than one argument. When a statement that is followed by several variables is executed, each of the variables is assigned the next available piece of data. Statement 200 for instance will read the next available piece of data and assign it to the variable "V". The next data entry will be assigned to "A1" and the next one will be assigned to "CC". It is important that the program be prevented from trying to read beyond the last data item in the last DATA statement because an error (number 31) will be issued if this is attempted.

RESTORE <line number> RESTORE [<line number>]

```
440 RESTORE
500 RESTORE 20    (restore to line 20)
```

When a BASIC program is run, the first execution of a READ statement causes the first element in the first DATA statement to be read (assuming of course that there is only one variable in the READ statement). The second execution of a READ causes the second available element to be read. Each time a READ is executed it reads the next available entry from the group of DATA statements that appears in the

program. When everything has been read from the first DATA statement the next READ will occur at the second data statement and so on. When a RESTORE is executed it causes the next available entry to be the first one appearing in the first DATA statement or, in other words, it resets the "next available entry" pointer to the beginning of the group of data that appears in the DATA statements of the program. Optionally one can specify a line number to restore to such that the next read will start at that line looking for a data statement instead of the first one.

6.2 Transfer of Control

NAME	FORM / EXAMPLE / EXPLANATION

GOSUB

<line number> GOSUB <line number>

5 GOSUB 250

This statement causes control to be transferred to the subroutine at the line number specified. The example will call the subroutine at line 250. All subroutines should end with a RETURN statement which will pass control back to the statement following the GOSUB which called the subroutine.

GOTO

<line number> GOTO <line number>

100 GOTO 50

The GOTO statement simply causes a branch to the specified line. When statement 100 (in the example) is executed, it forces a branch to line 50. The GOTO statement should only be used in the last position in lines containing multiple statements. This is because this statement always causes a branch and any statements following it would never be executed.

ON GOSUB

<line number> ON <expression> GOSUB <list of line numbers>

20 ON I GOSUB 30,40,50,60

The ON GOSUB statement allows calling one of several subroutines. The expression is evaluated and the integer portion of the result (any fractional portion will be truncated) determines where the jump will go. The "list of line numbers" has positions corresponding to 1,2,3,4,... So if the expression is evaluated to have a value of 1, then this statement will cause a subroutine call to the first line number in the list. A value of two for the expression will cause a subroutine call to the second line number in the list and so on for as many numbers as you have listed. If the expression evaluates to a number which is either less than or greater than the number of line numbers listed, an error message will result. The ON GOSUB statement should be the last statement on a line.

ON GOTO <line number> ON <expression> GOTO <list of line numbers>

200 ON I GOTO 500,600,700

This works exactly like the ON GOSUB statement except this causes a branch to one of several lines in the program. The ON GOTO statement is not used to branch to subroutines. Use the ON GOSUB statement instead. No statements should follow the ON GOTO in lines containing multiple statements.

ON ERROR GOTO <line number> ON ERROR GOTO [<line number>]

ON ERROR GOTO 1000

The ON ERROR GOTO statement allows user control of certain types of errors. All error numbers between 1 and 49 inclusive may be trapped and acted upon by the user. The ON ERROR statement tells BASIC where the user error routine is located and if an error occurs which is less than 50, control will be passed to the specified line number. See the section on "Using ON ERROR GOTO" for more details.

RESUME <line number> RESUME [<line number>]

RESUME 100

The resume statement is used to pass control back to the main program after an error routine has been executed. See the section "Using ON ERROR GOTO" for more details.

RETURN <line number> RETURN

34 RETURN

RETURN instructs the computer to return to the calling routine from the subroutine that is now being executed. This is how subroutines should be exited. When control returns to the routine that called the subroutine, execution resumes at the first statement following the statement that caused the branch to the subroutine. The statement that branched to the subroutine will normally be a GOSUB or an ON GOSUB statement. The RETURN statement, as with all the rest of the Branch statements, must be the last statement in lines containing more than one statement since statements following this (on the same line) would never be reached and executed.

6.3 Conditional

NAME — FORM / EXAMPLE / EXPLANATION

IF GOTO — <line number> IF <expression> GOTO <line number>

```
5 IF C=1 GOTO 110
1000 IF A>B AND F<>6.0 GOTO 300
```

The expression is evaluated and if it is true (has a nonzero value), the computer jumps to the line number following the GOTO. If the expression is not true, the next sequentially numbered statement after this "IF GOTO" will be executed. In other words, this statement would not cause BASIC to take any action if the expression is false.

IF THEN — <line number> IF <expression> THEN <line number>
< statement >

```
30 IF  A+B=10 THEN 50
80 IF  A=0 THEN PRINT "A=0"
99 IF  X=Y THEN IF X>Z GOTO 40
```

This works similar to the IF GOTO statement except that you don't have to branch to another line. If, as in the example, you follow the expression by "THEN 50" this has the same effect as GOTO 50. Each will cause a branch to line 50 if the expression is evaluated as being true. In line 80 of the example, we specified to execute the statement "PRINT "A=0"" if the expression is true. Executing another statement is our alternative to jumping to another line. The statement could even be another IF GOTO or IF THEN statement as in line 99 of the example.

IF THEN ELSE — <line #> IF <expr> THEN <line #> ELSE <line #>
<stment> <stment>

```
20 IF H/(Y+5)<8 THEN GOTO 50 ELSE PRINT "OUT OF RANGE"
```

This is identical to an IF THEN statement except that when an IF THEN statement evaluates an expression to be false it does nothing (doesn't execute the THEN part of the statement). Execution then passes to the next sequentially numbered statement. If the expression had been evaluated to be false and there had been an ELSE following what we just discussed, then whatever followed the ELSE would be

executed. To simplify the explanation slightly, the basic form for the IF THEN ELSE is:

<line > IF <expr> THEN <stment 1> ELSE <stment 2>

Considering this form, we will assume the expression has been evaluated. If it was true, then statement 1 will be executed; and if it wasn't, then statement 2 will be executed. Looking at the example, if the expression is true then the statement "GOTO 50" will be executed. If the expression is false, then the message "OUT OF RANGE" will be printed. Just as with the other conditional statements, IF THEN ELSE statements may be nested.

6.4 Input/Output

NAME — FORM — EXAMPLE / EXPLANATION

INPUT

FORM: <line number> INPUT ['string';] <variable list>
["string";]

```
556 INPUT "WHAT IS YOUR AGE";A
600 INPUT '"ENTER A NUMBER"';N
650 INPUT W,X,Y,Z
700 INPUT 'GIVE ME AN INTEGER';I%
```

The INPUT statement, when executed, prints out the character string enclosed in single or double quotes (if there is one). Then it prints a question mark and waits for the user to enter the value or values requested.

Either single or double quotes may be used. If one is to be part of the actual string, the other should be used to enclose the string as in line 600 above. In statements that request more than one variable, each entry from the keyboard must be separated by a comma and the last one is followed by a carriage return. If you don't enter as much data as the INPUT statement requests, BASIC will respond to your carriage return with "?" which is a prompt for more input. If too many entries are made then the extra entries will be ignored.

Note that the user may respond to this statement by typing a ^C. If this is done the program will be terminated and BASIC will be in a mode to accept new commands. The program will resume execution at the INPUT statement if the CONT command is typed.

INPUT LINE

FORM: <line number> INPUT LINE <string variable name>

```
10 INPUT LINE A$
20 INPUT LINE B1$(5)
```

The INPUT LINE statement is used to put an entire input line into a string variable. The entire line is accepted, including embedded spaces, punctuation characters, and quotes up to but not including the carriage return. No text string can be output as in the INPUT statement and only one variable name may be listed.

PRINT <line number> PRINT [variable, string;...] ,
[string ;variable,...] ;

```
10 PRINT                (only caused CRLF)
30 PRINT  "WHAT"        (prints "WHAT" on the terminal)
45 PRINT  "SPEED=";S
50 PRINT  A,B;X;Y
75 PRINT  "SOLUTION=";H*G/3.2   (expression)
```

The information following the PRINT statement can be left out or it can be an arbitrarily arranged string of constants, variables, expressions, and character strings. If the argument string is ended with a comma or semicolon, then a CRLF (carriage return line feed) will not be performed after the statement has been executed. Otherwise, a CRLF will be printed after the data. The individual items in the argument string are separated by either a comma or a semicolon but a short explanation of BASIC's printing format is necessary before the significance of each one can be understood.

The maximum length of a line is determined by the FLEX TTYSET "width" value. A width of zero means indefinite length. BASIC divides each output line into fields with each field containing 16 character positions. When arguments are separated by commas, the comma after one item will cause the printer to jump to the beginning of the next field before printing the next value or string. Thus, a comma will cause the printer to jump to column 16, 32, 48, 64, etc., depending on the current location of the printer. If the printer is already beyond the start of the last field, then the printer or CRT will do a carriage return/line feed and print the data in column 1 of the next line. You can type two or more commas next to each other. This will cause the printer to skip one field just as three commas would cause two fields to be skipped and so on.

When items are separated by a semicolon, they will be printed next to each other. There will still be a space or two between them if one or more of the adjacent arguments are numbers. This is because numbers are printed with one trailing space and a leading space if the number isn't negative. The last necessary bit of information is that character strings don't have leading or trailing spaces added. By knowing how to use commas and semicolons and by being aware of the format used to print numbers and character strings, you can tailor printed outputs any way needed.

PRINT USING <line number> PRINT USING <string>, <print list>

```
10 PRINT USING '####.##', 1234.56
20 PRINT USING '####.## IS THE SQUARE ROOT OF ###',SQR(X),X
30 PRINT USING '\  \','STRING FIELD'
40 PRINT USING A$, I, J, K%, B$
```

The PRINT USING is a highly flexible form of the print statement to give the user total control of the printing format. The <string> is an image of the output line except for special characters that are to be used for formatting instructions. The <print list> is the same as for a normal print statement where an expression is separated by either a comma (,) or a semicolon (;). Print using normally ignores the meaning of these separators UNLESS it is at the end of the print using string. Then and only then do the separators become meaningful. Because of this feature of print using, no literal characters can be between the last format field and the end of the print using string. Literal characters are defined to be any character that is not one of the following formatting characters and in some cases a comma. The special characters are:

EXCLAMATION MARK

The exclamation mark is used to denote a single character string field.

```
PRINT USING '! ! !', '01', 'AB', '()'
0 A (
```

BACK SLASH

A pair of back slashes (\) are used to denote a string field of 2 or more characters. The size of the field is determined by the total number of characters between the back slashes PLUS the two back slashes. Any character may be between the back slashes as they are ignored. It is recommended as a good programming practice to include a count or a string of numbers inside the slashes for programming documentation.

```
PRINT USING '\12345\', 'THIS IS A TEST'
THIS IS
```

POUND SIGN

The pound sign (#) is used to denote a number field.

```
PRINT USING '####.##', 124.555
124.56
```

If the number to be printed will not fit in the field defined a percent sign will precede the number and it will be printed as if no print using statement was used. Any character other than a comma, period, or up arrow will terminate the numeric field.

```
PRINT USING '#.#', 10.3
% 10.3

PRINT USING '#.#, #.#', 1,2
1.0, 2.0
```

If the fractional digits of the number does not fit into the field defined, the number will be rounded and then printed. If a number being rounded becomes too large to fit in the field a percent sign will be printed before the number.

```
PRINT USING '#.#       #.#', 1.99, 9.99
2.0      %10.0
```

DOLLAR SIGN

The dollar sign ($) is used when printing amounts of money. The field is defined the same as with the pound sign (#) except that the FIRST two characters of the field must be the dollar sign.

```
PRINT USING '$$###.##', 123.92
$123.92
```

Notice that the two dollar signs add an extra character to the size of the numeric field besides specifying the leading dollar. If there is not enough room in the field to insert the leading dollar, or if the number is negative, a percent sign will be printed then the number as if the dollar field had not been specified. It is not possible to print using the leading dollar and a negative amount at the same time. A trailing minus will have to be used.

```
PRINT USING '$$##.##', 1234
% 1234.00
PRINT USING '$$#.#', -1.23
```

```
%  -1.23
```

ASTERISK

The asterisk (*) is used to fill the leading blanks of any numeric field with asterisks. This is especially useful when printing a numeric field that should not be easily altered (ie. writing checks). The format is specified exactly as the leading dollar is and is used in much the same way. The two asterisks also specify an extra printable character. If room for at least one asterisk is not available a percent sign will be printed and then the number just as if an asterisk field was not defined.

The leading dollar and the asterisk field cannot both be defined for the same field. Instead a literal dollar sign can be used preceding the asterisk field.

```
PRINT USING '**##.##', 10.2
**10.20

PRINT USING '$**##.##', 1.15
$***1.15
```

COMMA

The comma (,) is used to insert commas in a numeric field very three places to the left of the decimal point. If at least one comma is embedded in a numeric field and before the decimal point then commas will be inserted appropriately. A comma before the numeric field or after the decimal point is considered to be a literal and will simple be printed. While filling the numeric field with commas, if BASIC runs out of field room, a percent sign will be printed followed by the number just as far as BASIC was able to fill it.

```
PRINT USING  '#,,,#.##', 1234.56
1,234.56

PRINT USING  '###,###,###', 1E6
1,000,000

    PRINT USING  '####,###', 1E6
    %1000,000
```

TRAILING MINUS

The trailing minus is used when printing negative numbers

and either a leading dollar field or an asterisk fill field. Negative sign cannot precede either of these fields so a trailing minus or debit is used.

```
PRINT USING '$$##.##- $$##.##', -10.23, -10.23
$10.23- %-10.23
```

UP ARROW

The up arrow (^) is used to denote scientific format for numeric fields. Four and only four up arrows are allowed and must trail the numeric field. The four up arrows (^^^^) are used to represent the 'E+XX' notation used in the scientific format. None of the other numeric formats may be used with the scientific format as the scientific format uses all possible printing positions. This format is particularly useful when printing tables of numbers that are quite large and have many decimal places.

```
PRINT USING  '#.############^^^^',  SIN(X)
2.55063602616E-01
```

6.5 Loops

NAME — FORM / EXAMPLE / EXPLANATION

FOR — <line #> FOR <variable>=<expr 1> TO <expr 2>[STEP<expr 3>]

```
 73 FOR H%=3 TO 600
 88 FOR B1=3.2-(X-7) TO 200+X STEP 5.5
116 FOR I%=10 TO -10 STEP -1
```

When BASIC executes a FOR statement, this will cause all the following instructions to be executed until a NEXT is reached. Execution then loops back to the FOR and if the condition specified in the FOR is true, then the statements will all be executed again. Once more control loops back to the FOR, and this cycle continues until the test finally fails, at which time execution resumes following the NEXT statement. The "variable" in the FOR statement is used to keep track of how many trips or loops have been made through the FOR-NEXT statements. The variable is initially assigned the value of "expr 1" and each time the NEXT statement loops back to the FOR statement, "expr 3" is added to the current value of the variable. Since "expr 3" can be positive or negative, this can have the effect of incrementing or decrementing the value of the variable. If no STEP is specified, then a value of positive one is assumed. When no step is specified (+1 assumed), or when a positive step is given, then the test that is specified in the FOR statement is determined to be true as long as the value of the variable is not greater than that of "expr 2". When a negative step is specified, the test yields true as long as the variable is not less than "expr 2". Once again, the statements between a FOR and NEXT will always be executed a first time but subsequent executions only occur while the index variable in the FOR statement is within the bounds of "expr 2".

FOR Statements may be nested (FOR statements located within other FOR statements), but they can't use the same index variable. The number of levels of nesting is limited only by the available memory space. FOR statements can be exited abnormally by using a GOTO instruction in the statements between the FOR and NEXT. They should not be entered like this unless they have previously been left abnormally. If you enter a FOR loop like this, the results are unpredictable and probably not what is desired because no initial value for the index variable has been specified.

NEXT <line number> NEXT <variable>

```
10 FOR I%=1 TO 10
20 PRINT "X"      (print 10 X's)
30 NEXT I%
```

The NEXT statement is only used in a program with a companion FOR statement. The variable specified in the NEXT statement is the same one that is in the FOR statement with which this NEXT is used. The FOR-NEXT pair specifies the boundaries of the loop. The NEXT statement is just a signal to the FOR statement that it has reached the end of the code it is to execute. NEXT causes a branch back to its corresponding FOR statement where the control variable is incremented as directed in that statement and then the test is repeated.

It is to the user's advantage to use integer FOR - NEXT loops whenever possible. Integer FOR -NEXT loops execute approximately 3 times faster than floating point and typically use 12 bytes less storage.

6.6 Termination

NAME	FORM / EXAMPLE / EXPLANATION

END

<line number> END

300 END

The END statement terminates program execution but its appearance in a program is optional. END can be placed anywhere in a program and it doesn't cause a break message to be printed. After a program has been terminated with an END statement, it can't be restarted with a CONTINUE command.

STOP

<line number> STOP

50 STOP

When a STOP statement is executed, program execution is terminated and a message is printed, informing the user where the break in execution occurred. If statement 50 above were executed it would halt the program and print the message:

```
STOP AT LINE 50
```

The program can be restarted with a CONT command and execution resumes following the STOP statement. In short, the STOP statement works just like the END statement except that STOP causes a break message to be printed.

6.7 Miscellaneous

NAME	FORM / EXAMPLE / EXPLANATION

DEF <line number>DEF FN<variable>(<dummy variable>)=<expression>

The user can define single line functions with the DEF statement. The function is defined (understood by the computer) as soon as the DEF statement has been executed. The same function may be redefined at any time in the program because only the most recent definition is used. The defined function may contain only one argument. The dummy variable used to represent this argument is local to the function definition and has no other meaning to the program.

A legal function name is formed by preceding any floating point variable name with the letters "FN". If, for example, we want to use the variables X, HI, and EE as function names, we would write them as FNX, FNHI, and FNEE respectively. A function is called just by the appearance of its name. When program execution does come across a function, execution branches to the "single line function definition" where the defining expression is evaluated and assigned to the function name. Execution then continues in the line where the function was called and the function name takes on the value of the function. At this point it is used in the expression in which it occurs just like it was an ordinary variable because it does have a specific value like a variable.

If we defined a function called "FNTT" which would multiply a variable passed to it by 10, we could use the following definition:

```
180 DEF FNTT(V)=10*V
```

Now suppose the definition has already been executed and the following two lines of code are executed:

```
300 X=3
310 Y=100+FNTT(X)
```

After they have been executed the value of Y will be 130. In statement 300, "X" is assigned a value of three. The next statement first calls the function FNTT. It is passed the variable "X" which is multiplied by ten as instructed by the definition. Control now returns to statement 310 where

the function name (which now has a value of 30) is added to 100 and this sum of 130 is assigned to the variable Y. Restrictions on the use of the DEF statement are that the definition must consist of only one line, only one argument is permitted, the returned value must be floating point, and string functions are not allowed.

DIM <line number> DIM <variable 1> (n) [,<variable 2> (m),..]
(k,l) (m,n)

20 DIM X(20),Y(30),Z(30,40)

The DIM statement was described in detail in section 4.4. In summary, the DIM statement is an instruction to the computer to reserve memory space for the variables that are listed. The memory space is allocated such that the variables can be referenced with subscripts. It is essential that all arrays be dimensioned with this statement before they are referenced in the program.

Whenever possible integer arrays should be used. For example, say you have a two dimensional array of 200 elements with dimensions (24,3). Remember all arrays in BASIC start with indices of 0. If the array is of type integer then it will need a total of 2*200 = 400 bytes of storage. On the other hand if the array is of type floating point then it will need a total of 8*200 = 1600 bytes of storage. One can easily see that a large floating point array can easily use all of memory and result in an error #80.

POKE <line number> POKE <address>,<data>
DPOKE <line number> DPOKE <address>,<data>

```
300 POKE 1000,33
670 POKE L,X
723 DPOKE HEX('0123'),496
800 DPOKE 0,0
```

Data may be passed to output ports or to machine language subroutines with the POKE (DPOKE) statement. These are two frequent uses. POKE (DPOKE) stores the byte (double byte) that is given as the second argument at the "address" that is specified by the first argument. The address and data that are used can be numbers or variables but they must have a numeric value that is subject to the following restrictions:

```
0 <= address <= 65535=(HEX("FFFF"))

0 <= byte <= 255
0 <= 2 bytes <= 65535
```

If either of these fall outside of the specified limits then the respective error numbers are issued.

Since this statement allows you to directly change the contents of memory, it should be used with care since you could write over your BASIC program or even over BASIC itself. It is safe, however, to write to non-existent memory locations.

REM <line number> REM [message....]

```
40 REM This could be the name of the program
50 REM
```

The REM statement is used to place remarks and comments throughout your program. Anything following the REM statement is ignored, including multiple statement per line characters, so remarks should be the last statement on multiple statement lines. Any character is legal following the REM and you can even leave the rest of the line blank as in statement 50. It's very likely you'll find REMARK statements at the beginning of programs to give the name of the program and what it does. Also they are often placed at different points in programs to explain what certain unclear or complicated sections of code do.

DIGITS <line number> DIGITS <total> [,<fractional>]

```
10 DIGITS 12        (set total digits to 12)
11 DIGITS 12,3      (set total digits to 12,
                     with 3 fractional digits)
```

The DIGITS statement specifies to BASIC how many digits to print independent of print using. The maximum number of digits is 17 and the minimum is 1. In a few cases, using the maximum of 17 can give an incorrect number in the 17th digit due to rounding performed in the binary to decimal conversion. Therefore, using the maximum of 17 is not recommended. The second argument specifies how many digits to the right of the decimal point to print. The number of fractional digits must be less than or equal to the total

number of digits or an error will be generated. If the second argument is left off then BASIC will print ALL fractional digits up to the total specified by the first argument. If a number is to be printed that is greater than the total number of digits specified, then the number will be printed in scientific notation.

```
DIGITS 4,3
PRINT PI
3.142

DIGITS 1,3
PRINT 10
 1E+1
```

SWAP <line number> SWAP <varl>, <var2>

```
10 SWAP A,B
20 SWAP A%(I%), A%(I%+I)
```

The SWAP statement does exactly what it sounds like. It swaps the the values of two variables. The variables must be of the same type and cannot be members of a virtual array. Its primary purpose is to decrease the time necessary to do a sort. In typical applications using the SWAP command can result in a savings of 20-30% in sort time. It is especially advantageous when sorting strings in that the strings are not actually moved, just the pointers pointing to them.

7. INTRINSIC FUNCTIONS

Since, in programming, there is often a need for many of the same functions to be used over and over again, several of the more common ones have been included as part of BASIC. The supplied functions are divided into five groups which are: MATHEMATICAL, TRIGONOMETRIC, CHARACTER, INPUT/OUTPUT, and MISCELLANEOUS.

Most of the functions require little explanation for what they do but a few words are necessary on the way that the functions in general are called. To call a function it is necessary only for its name and the required arguments to appear. For example, when the statement:

```
100 Y=SQR(9)
```

is executed, Y will have a value of three since SQR is the Square Root function.

There are also character functions such as CHR$(I) which return a character string.

7.1 Mathematical

The logarithmic and square root functions are generally accurate to sixteen decimal places.

FUNCTION	DEFINITION
EXP(X)	The mathematical operation e^X (e raised to the X'th power) is performed. Here "e" is the base of natural logarithms and is approximately equal to 2.718281828459045. The maximum allowable value of X is 88.02969193111306. An argument any greater than this will result in overflow and error message 102 will be issued.
LOG(X)	This is the natural logarithm (to the base "e") of the number X. This is the only logarithm supplied with BASIC, but it is the only one needed because logarithms to any other base can be calculated from it. The following formula is used to translate to logarithms of other bases where the LOG to the base B is desired. LOG of X to the base B = LOG(X)/LOG(B) Probably the most frequent use of this will be to convert to common logarithms (base 10). As an example suppose we wanted to find the common log of the number 327. The following expression will accomplish this. COMMON LOG OF 327 = LOG(327)/LOG(10)
SQR(X)	The Square Root function returns the square root of the argument X. If the argument is negative, error message 107 will be issued.

7.2 Trigonometric

For each of the trigonometric functions, the accuracy of the value returned is dependant on the magnitude of the argument passed to it. More accurate values will be returned by these functions when arguments with smaller magnitudes are used. In general, the trigonometric functions have an accuracy of thirteen and one-half digits.

FUNCTION	DEFINITION
ATN(X)	Returns the arctangent of X, in radians. The value returned will be between -pi/2 and pi/2, where pi/2 is approximately equal to 1.5707963267948966.
COS(X)	Determines the cosine of the angle X. The argument X is assumed to be in radians.
SIN(X)	This calculates the sine of the angle X where the argument is assumed to be in radians.
TAN(X)	The tangent of the argument X is calculated by this function. X is assumed to be in radians.

7.3 Character

FUNCTION	DEFINITION
ASC(X$)	The argument X$ is a string expression. The value returned by the function will be the ASCII numeric value of the first character in the string. Zero will be returned if the argument is the null string.
CHR$(I%)	This returns a single ASCII character (a one character string) whose ASCII value is the argument I%. The argument "I%" must be a number within the limits: 0 <= I% <= 255.
HEX(X$)	The HEX function converts a hexadecimal character string into its decimal equivalent. If the statement: 180 PRINT HEX("100") is executed, then the value printed would be "256" which is the decimal equivalent of the hexadecimal value of 100.
INCH$(I%)	The INCH$ functions inputs a single character from the file or device specified. The argument passed specifies the internal file channel. A channel of 0 is the users terminal.
INSTR(I%,S$,P$)	The INSTR function searches for sub-string P$ IN STRing S$. The first argument specifies the first character of string S$ at which to start the search. INSTR returns an integer value specifying at which character the sub-string started. If the sub-string was not found then zero is returned.
LEFT$(X$,I%)	Character function "LEFT$" returns a string that is the I% left-most characters of the string X$. The value of I% must be a positive number less than 32,767 to avoid

an error.

LEN(X$) LEN(X$) returns with the number of characters in the string X$. All characters in the string are counted, even spaces and non-printing characters.

MID$(X$,I%) This returns a character string that is a portion of the string X$. The returned string starts at position I% in the string X$ and includes everything until the end of the string. Position I% must be a positive number less than 32,768 or an error will result.

MID$(X$,I%,J%) With a three variable argument, MID$ functions the same as it does with a two variable argument except that the returned string only includes J% characters of the string X$.

RIGHT$(A$,I%) Returns a character string that is the rightmost I% elements of the string X$. If the value of I% is greater than or equal to the length of the string X$, then the entire string will be returned. Error number 74 will be returned if the value of I% is negative or greater than 32,767.

STR$(X) This will return a character string that represents the numerical expression X. In other words, it takes a number and transforms it to a character string. The string is constructed just as it would be output, honoring the DIGITS statement and including the leading space or minus sign and a trailing space.

VAL(X$) VAL(X$) does just the opposite of STR$(X). VAL(X$) takes a numerical character string (a string composed entirely of numbers, plus or minus sign, and decimal point) and converts it to its numerical VALue. Zero will be returned if the first non-space character is anything other than a digit or a decimal point, plus sign, or minus sign.

7.4 Input/Output

FUNCTION	DEFINITION
DPEEK(I) PEEK(I)	These functions do the opposite of the BASIC statements POKE and DPOKE. PEEK(I) and DPEEK(I) return the contents of memory location I (where I is the decimal address). The value returned will be >=0 and <=255 in the case of PEEK and will be >= 0 and <= 65535 in the case of DPEEK. If the current value of the argument is negative or greater than 65535, error number 104 will be printed.
POS(I%)	The value returned will be the current column position of channel I%. Numbering starts at zero, so if the value returned is zero, then the cursor is in the first position (column) of a line.
SPC(I%)	This function may only be used in a print statement. It causes "I%" spaces to be printed. BASIC will respond with error number 74 if the value of I% is negative or greater than 255.
TAB(I%)	This function also must only be used in print statements. It moves the cursor to column I% on the CRT or would move the print head to this column if a printer is being used. If the cursor is already past this position, then no action is taken. As usual, the argument must be positive and less than 256.

7.5 Miscellaneous

FUNCTION	DEFINITION
ABS(X)	The absolute value of X is returned by this function. If X is positive, then the value returned is X, and if it is negative, the value returned is -X.
INT(X)	The value returned is the largest integer that is not greater than X. This function is often called the "Floor" of X. Several examples of the values returned are shown below. INT(70) =70 INT(-.01)=-1 INT(5.7)=5 INT(-3) =-3 INT(0) =0 INT(-4.2)=-5
PI	This returns the value of "PI" = 3.1415926535897933.
PTR(<var name>)	The PTR function returns the address of the variable named as the argument. If the variable is a floating point type, the address returned will be the actual storage location of the number. Floating point numbers are stored as eight bytes, the first seven being the mantissa (sign plus magnitude, with the sign in the most significant bit position) and the last byte is the exponent (biased by hex 80). The mantissa is kept in hidden bit, normalized form. If the variable is an integer, the value returned will be the actual storage location of value of the variable. Integers are stored in 16 bit two's complement notation. If the argument is a string variable name, the value returned is the address of a four byte "string descriptor". The first two bytes of the string descriptor contain the actual address of the string, while the second two bytes indicate the strings length. It should be noted that no special string termination characters are stored with strings and any 8 bit combination is

valid in a string.

RND(X) The function returns a random number that has a value between zero and one. The programmer can use this to generate random numbers between any desired limits using the formula:

Random Number=(ML-MS)*RND(0)+MS

Where ML is the larger number and MS is the smaller number. The resulting number that is generated will range from MS to ML.

The argument X has an effect on the number that is generated according to the following rules.

X<0 A new series of random numbers is started. For different negative values of X, a different sequence is started each time, but if the argument retains the same value, the function will keep starting the same random sequence so the value returned will be the same each time the function is called.

X=0 Causes the function to generate a new random number when it is called. This is the argument that will normally be used with the RND function.

X>0 This returns the last random number that was generated.

SGN(X) This is the sign function. It returns "1" if the argument is greater than zero. A zero is returned if the argument is zero and a minus one is returned if negative.

FRE(0) FRE(0) returns the current number of free bytes available. The numeric argument is meaningless, but is syntactically necessary.

DATE$ DATE$ returns the current date in FLEX's date register in 'DD-MON-YY' format.

8. SEQUENTIAL FILE I/O

The simplest form of file I/O in BASIC is sequential I/O. Regular text files may be read and created sequentially. All disk data files are created with a default extension of "DAT" for DATA, and the default drive is the working drive. These defaults may be over-ridden at any time by simply specifying the desired extension or drive in the file name, as in any standard FLEX file specification.

8.1 The OPEN Statement

Before any file may be used in BASIC, it must first be opened. The OPEN statement is used for this purpose. For sequential file manipulations, there are two forms of the OPEN statement. The syntax is as follows:

```
OPEN OLD <string expression> AS <file expression>
OPEN NEW <string expression> AS <file expression>
```

Examples:

```
OPEN OLD "TEST" AS 1
OPEN NEW A$ AS F
```

The OPEN OLD statement will open the file specified in the string expression for read. The defaults are the working drive and DAT for the extension. The AS clause tells BASIC which I/O channel to use for the file I/O. The file channel may be 1 to 12, which means there is a limit of 12 open files at any one time (memory permitting). If the file is not found, an error will result (error #4). This error may be handled by using ON ERROR GOTO. Once opened, the file is ready for reading. The first example will open the file "TEST.DAT" on the working drive on channel 1 for read.

The OPEN NEW statement is used for creating a new file and preparing it for writing. The file specified in the string expression will be created if it does not already exist. If it does exist, the original file WILL BE DELETED and a new file with the same name will be created! The defaults for the file name are as in OPEN OLD. The second example above will open the file whose name is contained in the string variable A$ for write. The I/O channel used will be the value of the variable F.

It should be noted that the OPEN statement will not actually open the file on the disk, but will only make the preparations to do an open file operation. Only when actual file I/O is required will the file truly be opened on the disk. This implies that an OPEN OLD statement which tries to open a non-existent file will not generate an error, but when actual input is attempted, the error #4 will be generated.

8.2 Sequential Output

A modified version of the PRINT statement is used to output sequential data to a disk file. Its syntax is as follows:

<line number> PRINT #<expression>, [USING <string>,] <print list>

where the expression is the internal channel number specified in the OPEN statement. The list is the list of information to be output and follows all rules of the normal PRINT statement. An example will demonstrate its use.

```
10 OPEN NEW "TESTFILE" AS 3
20 PRINT #3, "THIS IS A TEST FILE"
30 PRINT #3, "THIS IS THE SECOND LINE"
```

The above lines create a new file called "TESTFILE.DAT" on the working drive. Internal channel number 3 is used. The resultant file will have two lines, the first will be THIS IS A TEST FILE and the second will be THIS IS THE SECOND LINE.

Another example will demonstrate a method of creating a table of the integers 1 through 5 and their values squared. The program might look as follows:

```
100 OPEN NEW "SQUARES" AS 1
110 FOR I=1 TO 5
120 PRINT #1, I, I^2
130 NEXT I
```

The file that is created is a pure text file and may be edited, listed, etc. with any of the standard FLEX utilities. Listing the file SQUARES.DAT created above would display the following:

```
1             1
2             4
3             9
4             16
5             25
```

Notice that since commas were used in the PRINT statements, the resultant lines have the numbers spaced accordingly.

8.3 Sequential Input

Like the PRINT statement, the INPUT statement may also communicate with a file. The INPUT syntax is:

```
<line number> INPUT #<expression>, <list>
<line number> INPUT LINE #<expression>, <string variable>
```

where the expression is the internal channel number referenced in the OPEN statement. The list is the same as the normal INPUT statement. No question mark prompt is output when inputting from a channel. The INPUT LINE will input an entire line from the disk file and put it in the string variable specified. An example will demonstrate input from a file.

```
10 OPEN OLD "NUMBERS" AS 2
20 INPUT #2, A, B
```

This will cause the variables A and B to be read from the disk file NUMBERS.DAT on the working drive. The file must provide ASCII data exactly as it would be typed if the input were coming from the terminal. Since two variables need data, the numbers in the file must be separated by a comma and terminated by a carriage return, or both may be terminated by a carriage return. If a file is being created to be read later by an input statement such as shown above, commas need to be manually inserted in the data file between the data items. For example:

```
100 OPEN NEW "NUMBERS" AS 6
110 PRINT #6, A; ","; B
120 CLOSE 6
130 OPEN OLD "NUMBERS" AS 6
140 INPUT #6, A, B
150 PRINT A, B
```

This sequence of lines will create a file NUMBERS.DAT on disk, output the values of A and B separated by a comma, close the file (see below), open the file for read, read in the values of A and B, and print the results on the terminal. It should be noted that reopening file for read after closing it will position the pointer back to the beginning of the file so all of the data becomes available for sequential read.

The maximum size of a line that may be read from a file is 255 characters.

8.4 The CLOSE Statement

The CLOSE statement is used to terminate I/O between BASIC and a file. Closing a file also frees up the internal channel it was using allowing another file to be opened on the same channel. The syntax of the CLOSE statement is:

<line number> CLOSE <expression> [,<expression>...]

The expression indicated should have the same value as that used in the OPEN statement for the file and indicates the internal channel number of the file to close. Any number of files may be closed with one CLOSE statement. Some examples will demonstrate its use.

```
200 CLOSE 3
520 CLOSE 1,6
```

Line 200 will close the file on I/O channel 3 and line 520 will close the files open on channel 1 and channel 6. An error will result if the file had not been previously opened.

8.5 INPUT on Channel 0

Some times it is desirable to request input from the terminal but not output a question mark prompt as the INPUT statement does. This can be accomplished by inputting from channel 0. Even though it is illegal to open a file on channel 0, it is possible to reference channel 0 with the INPUT statement, just for the purpose of eliminating the input prompt (the question mark). An example follows.

```
10 INPUT #0, B$
```

will request input from the terminal just as a normal INPUT statement, but no question mark prompt is output. After entering the data and typing the return key, a carriage return line feed will not be echoed as is done with the standard INPUT. This mechanism will allow precise cursor control in programs requiring fancy input prompts. Remember that files may not be opened for input on channel 0, but referencing channel 0 in an INPUT statement is allowed for the above stated purpose.

8.6 Output to Other Devices

It is often necessary to have a BASIC program output to a device other than the terminal. Channel 0 has special meaning for the PRINT statement just as it does for the INPUT statement. Using channel 0 without an OPEN statement when PRINTing will print on the terminal just as if the channel 0 had not been specified. As an example:

```
200 PRINT #0, "THIS IS A TEST"
```

will print THIS IS A TEST on the terminal, exactly as if the "#0," were not present.

Opening channel 0 allows you to send output to some device (such as a line printer) rather than the terminal. This is done in much the same way as the "P" command works in FLEX. Using OPEN and specifying channel 0 will tell BASIC to read the file name specified as a PRINT.SYS type file and use this new output routine whenever PRINT #0 is specified. An example will demonstrate its use.

```
10 OPEN "0.PRINT" AS 0
20 FOR I=1 TO 10
30 PRINT #0, USING  '##.######', I, SQR(I)
40 NEXT I
50 CLOSE 0
```

Line 10 tells BASIC to read in the file PRINT.SYS (SYS is the default extension when channel 0 is referenced). Notice that it was not necessary to say OPEN OLD. The PRINT.SYS file format is described in the FLEX User's and Advanced Programmer's Manual. Once the PRINT.SYS file has been read, all output through PRINT #0 statements will use the driver routines which are contained in the print file loaded. If the PRINT.SYS contained drivers to output to a parallel printer port on port 7, then the output from the above sample program would have gone to that printer. All output with PRINT statements not containing the #0 would still go to the terminal. The CLOSE 0 statement would tell BASIC to send all PRINT #0 output data back to the terminal again until another OPEN AS 0 statement is executed.

One last example will demonstrate the print to channel 0 use. This program allows output to be sent to either the terminal or the printer by user request.

```
10 INPUT "TERMINAL OR PRINTER (T OR P)",R$
20 IF R$="P" THEN OPEN "0.PRINT" AS 0
30 FOR I=1 TO 10
40 PRINT #0, I, SQR(I)
50 NEXT I
60 CLOSE 0
70 END
```

Line 10 asks if the output should go to the printer or the terminal. Only if the response is P for printer will the program open channel 0 (line 20). If channel 0 is opened, the output from line 40 will go through the printer drivers supplied from PRINT.SYS and on to the printer. If it is not opened, the output will go to the terminal. It is very important that the file specified in the OPEN AS 0 statement is actually a printer system file. If it is not, unpredictable results will occur, possibly crashing BASIC!

8.7 The KILL Statement

The KILL statement is used to delete an existing disk file. Its syntax is of the form:

<line number> KILL <string expression>

where the string expression is used as the file name to be deleted. The default extension is BAS and the default drive is the working drive. This statement may also be used in the immediate mode. There is no prompting with this command as in the FLEX delete command, so caution is advised! An example will demonstrate its use.

```
100 KILL "LEDGER"
```

This line will delete the file named LEDGER.BAS from the working drive. Again, there is no prompt with this delete so be careful!

8.8 The RENAME Command

The RENAME statement is used to rename a disk file. It may be used in a program or in the immediate mode. Its syntax is:

<line number> RENAME <string expression>, <string expression>

where the first expression specifies the name of the file to be renamed and the second expression is the new name it will have. The name defaults to a BAS extension and to the working drive. As an example:

```
225 RENAME "TEST", "OLDTEST"
```

This line will rename the file TEST.BAS on the working drive to OLDTEST.BAS. If TEST does not exist, an error #4 will be issued.

8.9 The CHAIN Statement

When a program is too large to be loaded into memory and run by BASIC it must be divided into several smaller segments. The CHAIN statement is used to load and run these program segments from a running program. The program segments may be either BAS type source files or BAC type compiled files. Each segment has its own file name and any program on disk may be passed control (similar to a GOTO) by CHAIN. The syntax is:

<line number> CHAIN <string expression> [<expression>]

The file name referenced by the string expression will be loaded and executed. The name will default to the working drive and BAC (compiled form) extension. The second optional expression designates the line number at which the program should be started. Without the line number specification, the program will begin execution at the lowest numbered line, just as with the RUN command. An example will demonstrate:

```
1300 CHAIN "BALANCE2" 100
```

This line will cause the file named BALANCE2.BAC to be loaded and run starting at line 100.

Chaining to compiled programs (BAC type) will be much more efficient than chaining to source type files (BAS). The only way a source file can be chained is if it has a BAS extension and if the extension is explicitly stated in the file name. All other extensions will be assumed to be compiled type files. Communication between chained programs must be performed through disk files. When the CHAIN statement is executed, all open files are closed, the new program is loaded, and execution continues. Any files to be used in common by several programs should be opened in each of the programs run.

9. ERROR HANDLING

There are two classes of errors which can happen while executing a BASIC program. The first class consists of I/O errors, both disk and terminal related, while the second class deals with computational and syntax type errors. The complete list of error numbers and their respective meanings are in a following section. The error numbers between 1 and 49 are all I/O type errors while 50 and above fall into the second class. It should be noted that error numbers 1 through 28 are disk errors and are the same as those generated by FLEX. Normally BASIC will print the error number as the error occurs and the program will come to a halt. Many times it is desirable to continue execution of a program after an error occurs, especially if the error is I/O related. The ON ERROR statement is used for this situation and may be used to control the result of any I/O related error.

9.1 The ON ERROR GOTO Statement

The ON ERROR GOTO statement is used to tell BASIC that there is a user supplied error handling routine. Anytime an error occurs BASIC will check to see if an ON ERROR statement has been executed. If it has, control is transferred to the line specified, otherwise, the program will halt and print the error message in the usual manner. The syntax for the ON ERROR statement is:

<line number> ON ERROR GOTO [<line number>]

This statement should be placed in the program before any lines which may cause an I/O error for which the error routine deals. If an error does occur, control will be transferred to the line number specified in the ON ERROR statement. The system variables ERR and ERL will also be set to the value of the error number and to the line number which caused the error. More about these variables to follow.

In addition to errors, the ON ERROR GOTO statement can be used to process a break (CONTROL C) and the FLEX "escape return". The "escape return" occurs when the ESCAPE key is used to stop output from printing, followed by pressing the RETURN key (carriage return). Typing either a CONTROL C or an "escape return" sequence will cause control to go to the error handling routine. CONTROL C may be detected by checking for an error 34, and the "escape return" sequence is detected by checking for an error 37. If there is no error handling routine, control will go to the READY prompt.

9.2 The RESUME Statement

The RESUME statement is used to pass control back to the main BASIC program after the error handling routine has completed. During program execution, if an I/O error occurs, and an ON ERROR statement has been executed, BASIC will go to the first line of the error handling routine. When the error routine is finished it must pass control back to the main program, the function of RESUME. The syntax for the RESUME statement is:

<line number> RESUME [<line number>]

If a line number is specified after RESUME, BASIC will restart the main program execution at that time. If no line is specified (or 0 is specified), BASIC will resume by re-executing the line which caused the error. If this second method is used, note that the entire line will be re-executed, and not just the statement which caused the error. Thus, if the offending statement was not the first statement on the line, the proceeding statements on that line will also be repeated. Two examples follow:

```
1000 RESUME
2000 RESUME 200
```

The first example will restart the main program on the line which caused the error (this is equivalent to RESUME 0). Line 2000 would cause the program to resume at line 200. A RESUME statement should always be included in error handling routines. No other form of return is valid.

9.3 ERR and ERL System Variables

When any error occurs, the two variables named ERR and ERL are updated. The ERR variable will contain the error number and ERL will contain the line number of the line which was executing at the time the error happened. These variables may not be set by the user but may be read at anytime. As an example:

```
130 IF ERR=4 THEN PRINT "NO SUCH FILE"
```

This line will cause BASIC to print NO SUCH FILE if the last error was an error number 4. The variable ERL is used in a similar manner. It should be noted that programs using the ERL variable may need alteration after a RENUMBER operation since the line referenced may change!

9.4 Disabling ON ERROR

Many times there are only sections of a program which require a user supplied error routine. It is possible to disable the ON ERROR feature once it has been disabled by using one of the following:

```
<line number> ON ERROR GOTO 0
<line number> ON ERROR GOTO
```

Both of these are equivalent. After execution of this statement, BASIC will handle all errors, printing the error number and halting. Another user error handler may be setup at anytime by executing another ON ERROR GOTO and specifying the line number of the routine.

The ON ERROR GOTO 0 statement may also be placed inside a user error handling routine. If BASIC comes across an ON ERROR GOTO 0 statement while executing a users error routine, the user routine immediately exits, and BASIC will print the error number on the terminal, just as if the user routine had never been activated.

9.5 Error Handling Examples

Following are three examples which demonstrate typical applications of the ON ERROR and related statements. The first example deals with "data type mismatch" errors (error number 30).

```
  10 ON ERROR GOTO 1000
  20 INPUT "PLEASE TYPE THREE NUMBERS",A,B,C
  30 PRINT "THE SUM IS";A+B+C
  40 GOTO 20
1000 IF ERR<>30 THEN ON ERROR GOTO 0
1010 PRINT "PLEASE TYPE NUMBERS ONLY!"
1020 RESUME
```

Line 10 tells BASIC where the error routine is located (line 1000). When the INPUT statement is executed on line 20, it expects numeric data only to be input. If the user enters a letter instead of a number, BASIC would normally stop and print ERROR #30 AT LINE 20 at the terminal. Since the ON ERROR statement has been executed, BASIC would transfer control to line 1000 if an error occurred. Line 1000 checks to see if the error is number 30. If ERR is not 30, the ON ERROR GOTO 0 is executed which disables the error handling routine and causes BASIC to print the error number on the terminal and stop. If it is error 30, the message on line 1010 is printed. Line 1020 will cause the program to resume execution at the line which caused the error (line 20), and the input prompt will be reissued.

The second example demonstrates disk I/O error handling. A common situation when working with disk files is the detection of the end of file. The following short program will list a disk text file on the terminal.

```
 10  INPUT "PLEASE TYPE FILE NAME (WITH EXTENSION)",F$
 20  ON ERROR GOTO 1000
 30  OPEN OLD F$ AS 1
 40  INPUT LINE #1, L$
 50  PRINT L$
 60  GOTO 40
100  CLOSE 1
110  END
1000  IF ERR<>8 THEN ON ERROR GOTO
1010  RESUME 100
```

After the file name is input on line 10 the error routine is identified to BASIC in line 20. Line 30 opens the file for sequential input on internal channel 1. Lines 40 and 50 read the file a line at a time and prints each line at the terminal. Eventually the end of the file will be reached and an error will occur at line 40 from attempting to read past the end of the file. At this time control will be passed to line 1000 which tests to see if the error was an error number 8 (end of file). If it is, then the program is resumed at line 100 which closes the file and ends the program. If the error was not 8, then the error number will be printed and the program stopped.

The third example deals with detecting control-C traps while a program is running. Some times it is necessary to cleanly exit a program, to close files, print a summary of action taken, etc. If the user types a control-C before the program has finished, the termination procedures will not be executed. By using the "ON ERROR GOTO" mechanism a program can detect just this thing. Error number 34 is control-C trap and when any ^C is typed, control is passed to the error handling routine. One warning when using this mechanism, if the program disables the ^C trap, some other means of exiting the program must be available. Either by a STOP or END statement or by the ESCAPE key on output. If the program goes into an infinite loop, and some other means of stopping it can not be used, then a machine RESET will be the only way to break the loop.

```
 10  ON ERROR GOTO 100
 20  INPUT "GIVE ME A NUMBER", C
 30  IF C=0 THEN END
  .
  .
  .
100  IF ERR<>34 THEN ON ERROR GOTO 0
110  PRINT "YOU CAN'T QUIT ON ME NOW !"
120  RESUME 20
```

10. ADVANCED DISK CAPABILITIES

10.1 The EXEC Statement

The EXEC statement is used to execute any FLEX utility which loads into the FLEX utility command space ($A100 for 6800, $C100 for 6809). Its syntax is:

<line number> EXEC, <string expression>

where the string expression is used as a FLEX command, just as it would be typed into FLEX. As an example, suppose it was necessary to set a TTYSET parameter during a BASIC program execution, such as the line width. The line might look as follows:

300 EXEC, "TTYSET WD=64"

The string in quotes would be sent to FLEX and executed, just as if it had been typed directly into FLEX. Remember that only commands which reside in the utility command space should be called. Programs which load into low memory will kill BASIC! BASIC does not check for a low memory utility, so it is up to the user.

10.2 Virtual Arrays

The simplest form of random file I/O is called virtual arrays. The virtual array mechanism allows the user to specify that a data array be stored on a disk file rather than in memory. The two advantages of this feature are that the array may be much larger than what would fit in the available memory and that the data in the array remains after program execution and may be used at a later date from another program. Virtual array data is referenced exactly like standard array data which makes the mechanism quite powerful and easy to use!

The sequential I/O methods previously described only allow the next sequential data item to be accessed or stored at any one time. The virtual array storage method allows a random data item to be accessed or stored, no matter where in the file the item resides. Before a data matrix can exist in a virtual array, the array must be declared using a special form of the DIM statement. Its syntax is:

<line number> DIM #<expression>, <variable>(<dimension>)

The expression designates an internal channel number and must be between 1 and 12. This is the channel number on which the file will be opened. Only one variable may be associated with each virtual array channel number. The variable may either be single or double dimension. As an example:

```
20 DIM #3, A(100,50)
30 DIM #4, B%(100)
```

would define the matrix A to be 101 by 51 in size and be associated with channel 3; and define B% to be a vector with 101 elements associated with channel 4. Virtual arrays may be floating point, integer, or string. All data items are stored in the file in "internal format", 2 byte binary for integers, 8 byte binary for floating point numbers, and ASCII characters for strings.

For the most part, virtual array and standard memory array manipulations are identical. One difference is in the way string storage is performed. In a standard type string array, the data items may be any length and change in size as the program executes. A virtual string array requires each string in the array to have the same length. The maximum length allowed is 252 characters but may be defined to be anywhere from 1 to 252. Each string element stored into a virtual array will either have enough spaces attached to the right side of the string to make it equal in length to the defined string size (if it is shorter than the defined length), or it will be truncated from the right if it is longer than the defined length. To define the string length for a particular virtual string array, the following form should be used.

<line number> DIM #<expr>, <string var and dimension>=<expr>

Example:

```
100 DIM #7, A$(100)=63
```

The equals sign and expression define the string length to be used. Again, the maximum length is 252 characters. The example defines the actual string array A$, which has 101 elements (including the 0 element), each of which are 63 characters in length. If a string length is not specified in the DIM statement, a default length of 18 is used.

The length of a virtual string array data item can greatly affect the efficiency of data storage in a disk file. The system requires that a string be completely contained in one disk sector and that the string may not cross a sector boundary (thus the limit of 252 since there are 252 bytes of storage available in one sector). To avoid wasting disk space, the defined string length should be an even divisor of 252. A few examples will demonstrate this.

```
10 DIM #1, A$(100)
20 DIM #2, B$(100)=63
30 DIM #3, C$(20)=130
```

Line 10 would default to a string length of 18, creating 14 strings per sector. No disk space is wasted since 14 times 18 is 252. Line 20 would create strings of length 63, again not wasting disk space since there would be 4 strings per sector and 4 times 63 is 252. Line 30 is extremely wasteful since only one string of length 130 will fit in a sector, so 252-130 bytes (122 bytes) are wasted in each sector of the File.

10.3 Opening a Random File

Before any random file may be referenced, it first must be opened. Opening a random file is a little different than opening a sequential file. There are three forms of the OPEN statement for random file use.

```
<line number> OPEN OLD <string exp> AS <expression>
<line number> OPEN NEW <string exp> AS <expression>
<line number> OPEN <string exp> AS <expression>
```

The OPEN statement used determines whether an existing disk file is to be used or if a new file should be created. The open statement does not determine if the file is going to be used for input or output as in the sequential file use since random file operations allow both input and output to a particular file. Each OPEN type will be described separately.

The OPEN OLD statement tells BASIC to search for a file which already exists (an "old" file). If the file is not found, an error number 4 will be issued. As an example:

```
10 OPEN OLD  "TEST" AS 8
```

will cause BASIC to search the working drive (by default) for the file named TEST.DAT. DAT is the default extension. Either of the defaults may be overridden, just as any FLEX file specification. If the file is not found, error 4 will be generated.

The OPEN NEW statement tells BASIC to create a "new" file. If the file name specified already exists, IT WILL BE DELETED, and a new file of the same name will be created. If the file does not exist, it will be created. As an example:

```
10 OPEN NEW "INVENT" AS 4
```

will cause a new file named INVENT.DAT to be created on the working drive. The same file name defaults apply as stated above.

Finally, the OPEN statement, without the NEW or OLD modifier will first attempt to open an existing file (just like OPEN OLD). If the file name is not found, one will be created and no error will be issued.

```
10 OPEN "NAMES" AS 1
```

This line would cause BASIC to first search for a file named NAMES.DAT on the working drive (the above stated file name defaults apply). If the file is found it is opened. If the file is not found, one is created with the specified name.

As in the sequential OPEN statement, the file is not actually opened at the time the OPEN statement is executed, but is only prepared for the opening process. When the first I/O attempt is executed, the disk file will actually get opened, so any file related errors will be delayed until this first 1/O operation is performed.

10.4 Using Virtual Arrays

Before a virtual array may be used, the corresponding disk file must be opened using the OPEN statement described above, and the array must be defined using the special DIM statement. Once this has been done, the virtual array may be used in assignments and expressions exactly like any standard array. A short example will demonstrate virtual array use.

```
 10 OPEN "TESTFILE" AS 1
 20 DIM #1, A(100)
 30 INPUT "TYPE ARRAY ELEMENT, NEW VALUE",E,V
 40 PRINT "THE CURRENT VALUE IS";A(E)
 50 A(E)=V
 60 PRINT "THE NEW VALUE IS";A(E)
 70 CLOSE 1
 80 END
```

Line 10 opens the file named TESTFILE.DAT on the working drive. If the file does not already exist, one will be created (because of the type of OPEN statement used). Line 20 defines the virtual array A which is a floating point array containing 101 elements (counting element 0). From

this point on, the program treats the array A just as if it were a standard array. Line 30 requests which element in the array is to receive a new value and what that value should be. Line 40 prints the current value of that data item, line 50 gives it the new value, and line 60 prints the new value of the selected array item. Notice that the array reference is purely random, and that it was not necessary to access the file sequentially. Line 70 closes the file (as is necessary after completing use of any disk file). The file TESTFILE now exists on the disk with the selected data item altered to reflect its new value.

10.5 Notes on Virtual Arrays

A file created or referenced as a virtual array has no information contained in it which specifies its dimension or data type. If the file was created as a floating point virtual array, it consists of 8 byte binary numbers, one right after the other, with 31 numbers per sector (31 times 8 is 248) with 4 bytes left over. If it was created as a string array, it consists of one string after the other, of length specified in the DIM statement. If the array is integer, it consists of 126 integers (2 times 126 is 252), one right after the other. A double dimensioned virtual array is stored in row form. This means that row 0 is closest to the beginning of the file, followed by row 1, etc. This information is important in two respects. First of all, sequential access of virtual array data items in a 2 dimension array will be faster if accessed by row than if accessed by column. The following example demonstrates this point.

Program #l

```
10 OPEN OLD "DATA" AS 1
20 DIM #1, A(20,20)
30 FOR I=0 TO 20
40 FOR J=0 TO 20
50 PRINT A(I,J)
60 NEXT J
70 NEXT I
80 CLOSE 1
```

Program #2

```
10 OPEN OLD "DATA" AS 1
20 DIM #1, A(20,20)
30 FOR I=0 TO 20
40 FOR J=0 TO 20
50 PRINT A(J,I)
60 NEXT J
70 NEXT I
80 CLOSE 1
```

The two programs are identical except for line 50. In program 1 line 50 accesses the array by row (the row index advances the slowest) while in program 2 line 50 accesses the array by column (the column index advances the slowest). Program I will run much faster since it is progressing through the file sequentially, while program 2 is having to randomly thrash about the file which makes it run slower.

The second point concerning 2 dimension arrays (which also applies to 1 dimension arrays) is the dimensioning method. Since a virtual array data file is just a collection of data items in the file, one right after the other, the file may be dimensioned in any way desired.

It usually only makes sense to dimension the array in the same way it was dimensioned when it was created, but this is not necessary. As an example, suppose a virtual array was created and dimensioned as (50,50). The array would potentially contain (50+1) times (50+1) data items or a total of 2601 elements. Normally, when referencing this array file at a later date, the same dimensioning would be used. We could however, just as easily dimension this existing array file as a single dimension array of 2060 (2061 elements). It could also be dimensioned as a one dimension array containing only 100 items, which would yield the remaining elements inaccessible.

When a virtual array file is closed in a program, the array becomes "undimensioned". If the virtual array file is reopened, the array must be dimensioned again.

10.6 Extending Virtual Array Disk Files

When a random file is created it contains only one usable sector. If a virtual array element is referenced (read) which lies beyond the end of the file, an error number 24 will be issued (non-existent record number referenced). This is similar to an end of file type error with sequential file reading. A random file may be extended in size at anytime by assigning to an array element which lies beyond the end of file. Anytime an extension is necessary, BASIC will extend the file by a few more sectors than actually needed, in anticipation of another extension assignment. File extending can take considerable time and the actual amount of time required is directly proportional to the number of sectors by which the file is being extended. As an example:

```
10 OPEN NEW "DATA" AS 1
20 DIM #1, A(250)
30 PRINT A(250)
```

will cause an error 24 at line 30 since a new file was just created and no data existed at the element referenced. To make this program run without an error, it is necessary to extend the file before the reference is made.

```
10 OPEN NEW "DATA" AS 1
20 DIM #1, A(250)
25 A(250)=0
30 PRINT A(250)
```

The addition of line 25 causes the file to immediately be extended to contain all 250 data items. Now line 30 will run without error. It is not necessary to extend a file to its final size before use, but it must be extended to the size necessary to accommodate all data items which will be referenced. All new data items created at the time a file is extended will be zero if the array is floating point, and will be null strings if it is a string array.

10.7 Record I/O

Up to this point, two methods of disk I/O have been described, sequential I/O using PRINT # and INPUT #, and virtual arrays for random file access. The sequential I/O is simple and easy to use but restricts data access to sequential methods. Virtual arrays provide high speed random access but do not allow the mixing of ASCII and numeric data in one disk file. There exists a third type of I/O called record I/O. Record I/O is the most flexible form of disk I/O but is also the most complex. Data may be accessed randomly and both ASCII and numeric data may be freely mixed in one disk file.

The basic idea behind record I/O is that data is stored on the disk in fixed length records. These records are each 252 bytes or characters in length and reside in one physical disk sector. Any record in a file may be read or written upon request and the data in each record is easily defined to be ASCII or numeric data. Strings may be stored on disk as characters and numbers may be stored in eight byte binary form as floating point or as 2 byte binary integers, eliminating the need for I/O conversions. Record I/O is a very efficient and quick method of saving data on a disk file.

10.8 Opening and Closing Record I/O Files

Since record I/O files are random files, they are opened and closed exactly like virtual arrays. See section 11.3 for details. It is important to remember that random files are special and may be read in either a random or sequential manner. Sequential files however, may only be read sequentially. An attempt to open a sequential file for random operations will result in error number 48. The user will find the plain OPEN statement (without the NEW or OLD modifier) to be the most useful since it will work with existing files, or create a new file if the file referenced does not exist, without causing an error. Closing a record I/O file is exactly like closing any file and all of the same rules apply, as stated earlier.

10.9 The GET and PUT Statement

All record I/O is performed by record (252 byte blocks). Records may be accessed randomly or sequentially. Once a file has been opened on a particular channel, the GET and PUT statements may be used to transfer data to and from the file. Their syntax is:

<line number> GET #<expression> [,RECORD <expression>]

```
<line number> PUT #<expression> [,RECORD <expression>]
```

The first expression in each line represents the internal channel number referenced in the OPEN statement and must be between 1 and 12. The RECORD portion of each line is optional, and if used, the expression following it designates which record number of the file should be used. If the RECORD option is not specified, the next sequential record will be read or written. The GET statement will read the appropriate record from the disk file into the I/O buffer associated with the channel number used. The PUT statement will write the contents of the I/O buffer to the specified record of the file.

Records in a random file are numbered from 1 to n, where n is the size of the file in records. Trying to GET a record which is larger than n will result in an error number 24. Using PUT to write a record which is larger than n will automatically extend the file. There will be more information on disk file extension following. After performing a random access GET or PUT on a disk file, the next GET or PUT on that channel will access the next sequential record. As an example:

```
10 OPEN "TEST5" AS 2
20 GET #2, RECORD 25
30 PUT #2
```

The PUT statement in line 30 will write record number 26 to the disk file since it is the next sequential after record 25.

10.10 The FIELD Statement

So far, methods of opening and closing record I/O files have been described, as well as methods of reading and writing the 1/0 buffer associated with the file. This section and the following deal with the manipulation of the data in the I/O buffer.

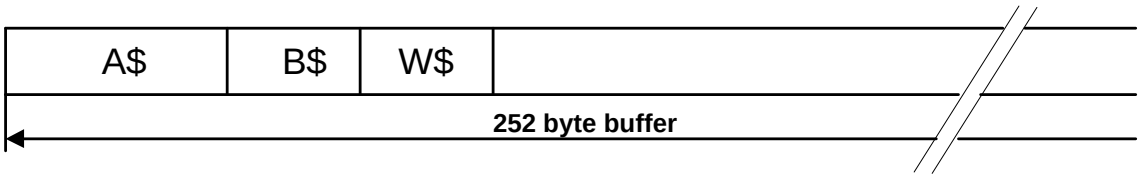
For each record I/O file opened, there exists an I/O buffer designated by the internal channel number used with the OPEN statement. Each I/O buffer is 252 characters in length and is used for temporary storage of each disk file record as it is being operated on. The FIELD statement is used to associate string names with various parts of the I/O buffer. Its syntax is:

```
<line number> FIELD #<expr>, <expr1> AS <string var1>
                          [, <expr2> AS <string var2> ...]
```

where the expression designates the internal channel number used in the OPEN statement. Expression 1 is used to designate the length, in characters, of the associated string variable and string var1 is a unique string variable name for this part of the buffer. As many expressions and names as desired may be listed and are ssociated left

to right in the I/O buffer assigned to the channel number referenced. As an example:

```
100 FIELD #1, 20 AS A$, 10 AS B$, 6 AS W$
```



As shown in the diagram, line 100 would associate the string A$ with the first 20 character positions in the I/O buffer, B$ with the next 10 character positions, and W$ with the next 6 positions. The remaining 216 characters of the buffer are left undefined. The total number of characters positions associated with an I/O buffer must be less than or equal to 252. Each time a FIELD statement is executed, it will start the string association with the first character position in the buffer, regardless of how the buffer has been previously defined with prior FIELD statements. Once a variable name has been fielded by using the FIELD statement, its value will be whatever is currently in the I/O buffer it is associated. If the contents of the buffer change (by executing a GET statement) the contents of the string will also change. The string length will be the length allocated for that string in the FIELD statement.

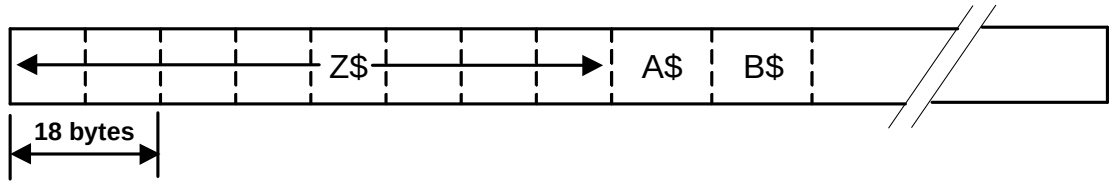
The FIELD statement does not move any data between variables and the I/O buffer but simply sets up a field definition for later use using LSET and RSET. Using a FIELD statement to associate a string variable with an I/O buffer is temporary and the definition is nullified by any attempt to assign a value to the string variable by using LET or the implied LET. As an example:

```
10 OPEN "TEST" AS 1
20 FIELD #1, 50 AS B$
30 B$="TEST STRING"
```

Line 30 does not put the string "TEST STRING" into the I/O buffer but instead removes B$'s association with the I/O buffer giving it new storage area for the string. The result is that line 30 nullifies the FIELD definition setup in line 20.

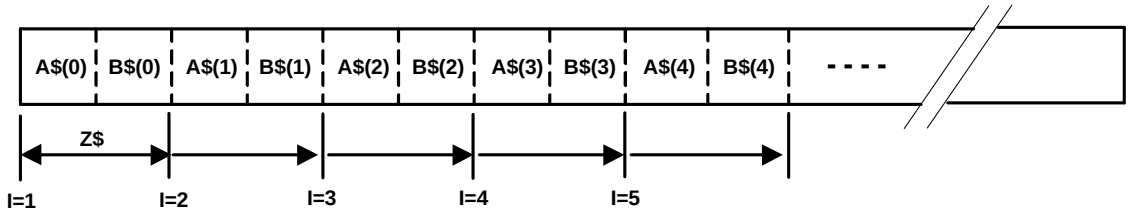
It is possible to break the I/O buffer up into several smaller records if a record size of 252 is too large. As an example, suppose that each record of a disk file contains 14 sub-records, each 18 characters long (14 times 18 is 252). Each sub-record is further divided into one 10 character field and one 8 character field. To access the fifth sub-record of the I/O buffer, the following statement could be used:

```
120 FIELD #1, 72 AS Z$, 10 AS A$, 8 AS B$
```



This line would cause A$ and B$ to point to the desired sub-record. The string Z$ is used as a "dummy string". Its purpose is to skip the first four records (4 times 18 is 72). A more general statement may be used which will allow accessing any one of the 14 records in the I/O buffer.

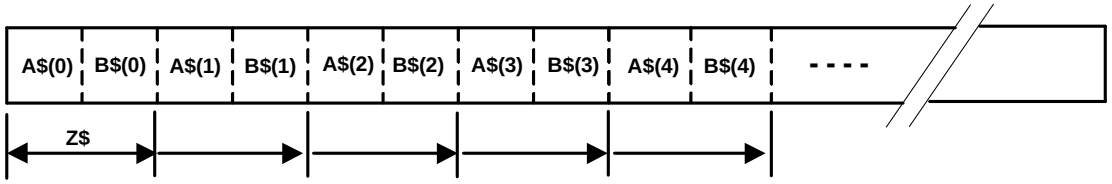
```
150 FIELD #1, (I-1)*18 AS Z$, 10 AS A$, 8 AS B$
```



When executing the above statement, the variable I should contain the desired sub-record number (a number between 1 and 14). As an example, if I is equal to 3, the dummy string Z$ will be assigned the first 36 character positions, allowing A$ and B$ to point to the third sub-record. If I is 1, A$ and B$ will point to the first sub-record since Z$ was given zero character positions in the buffer.

It is also possible to use subscripted string variables in a FIELD statement (except for virtual strings). As an example:

```
100 DIM A$(13), B$(13)
110 FOR I%=0 TO 13
120 FIELD #1, I%*18 AS Z$, 10 AS A$(I%), 8 AS B$(I%)
130 NEXT I%
```



This set of statements will associate each element of the string arrays A$ and B$ with a sub-record in the I/O buffer. A$(0) and B$(0) will be associated with the first sub-record, A$(1) and B$(1) with the second, and so on.

10.11 The LSET and RSET Statements

The FIELD statement has been used to associate a string with an I/O buffer. Once fielded, the strings contents may be altered by using RSET or LSET. These statements are similar to the LET statement except the string storage location is not changed as it is with LET. The syntax for these two statements is as follows:

<line number> LSET <string variable> = <string expression>

<line number> RSET <string variable> = <string expression>

Where string variable represents any legal string variable name, including subscripted variables. These statements store the result of the string expression into the previously defined space, the original string being overwritten. The old length of the string is not changed. If the new string is longer than the old one, the new one will be truncated to the same length as the old. If the new string is shorter, LSET will left justify the string, padding to the right with spaces until the lengths are equal, while RSET will right justify the string, padding with spaces on the left. The normal use of LSET and RSET is with FIELDed string variables but they may also be used with regular strings. They can be used to assign a value to any legal string variable in BASIC, following the above rules for padding and truncation.

A short example will demonstrate the record I/O tools described so far.

```
10 OPEN "DATA5" AS 1
20 FIELD #1, 10 AS A$, 40 AS B$, 70 AS C$
30 GET #1, RECORD 15
40 PRINT A$
50 PRINT B$
60 PRINT C$
70 LSET A$="NEW A$"
80 LSET B$="NEW VALUE FOR B$"
90 RSET C$="NEW RIGHT JUSTIFIED C$"
100 PUT #1, RECORD 15
110 CLOSE 1
```

Line 10 opens the file DATA5.DAT on the working drive. Line 20 sets A$, B$, and C$ to the channel 1 I/O buffer. Line 30 reads record number 15 into the I/O buffer. The fielded string variables now contain the values from that record, so the print statements on lines 40 through 60 will print their values. The strings printed reflect what is contained in those character positions of record 15. Lines 70 through 90 assign new values to these strings using LSET and RSET. Remember that the new strings are actually being stored in the I/O buffer. Line 100 writes the I/O buffer back into record number 15 of the data file. If line 100 were omitted from this program, the file would remain unchanged. The file is finally closed in line 110.

10.12 The CVT Conversion Functions

The statements described above allow string data to be stored and retrieved from the I/O buffer. It is often desirable to store numeric data in a record I/O file. Four conversion functions exist which allow floating point and integer numbers to be converted to strings and strings to floating point and integer numbers. Examples of each follow:

```
A$ = CVTF$(X)
B$ = CVT%$(X%)

X  = CVT$F(A$)
X% = CVT$%(B$)
```

The first pair of functions map their respective arguments (variable, constant or expression) into an eight (two for integers) character string. Each character in the new string is one of the eight (two) bytes of the floating point (integer) number. The second pair of functions does just the opposite. It maps the first eight (two) characters of a string into a floating point (integer) number. If the string has fewer than eight (two) characters, null characters are appended. These functions should not be confused with the STR$ and VAL functions. STR$ and VAL work with ASCII numbers, not internal binary and as a consequence, must perform a very time consuming ASCII to binary conversion. The CVT functions provide a compact and fast way of storing numeric data in a record I/O file. As an example, suppose it was necessary to store the items of a floating point array in a record I/O file. The following program demonstrates a way of accomplishing this.

```
10 DIM A(30), A$(30)
20 OPEN "FPDATA" AS 1
30 FOR I%=0 TO 30
40 FIELD #1, 8*I% AS D$, 8 AS A$(I%)
30 NEXT I%
.
.
.
200 FOR I%=0 TO 30
210 LSET A$(I%) = CVTF$(A(I%))
220 NEXT I%
230 PUT #1
240 CLOSE 1
230 END
```

After the variables are dimensioned and the file is opened, lines 30 through 50 setup the FIELD definition. Each element of the string array A$ is associated with a four byte section of the I/O buffer. It is assumed that the array A is assigned values between lines 50 and 200. Lines 200 through 220 assign the new buffer values by assigning to each element of the previously fielded array A$. Notice that the CVTF$ function is used to store the floating point elements of the array A

into the buffer. Line 230 writes the record to the disk file and line 240 closes the file.

10.13 Extending Record I/O Files

As stated earlier, a random file is created with only one sector (record). If a GET statement is used to read a record of a file, and the file does not contain that many records, an error number 24 will be issued. This is similar to en end of file error with sequential files. A record I/O file may be extended at anytime by simply using a PUT statement to write a new record which lies beyond the end of the file. Anytime a file is extended, BASIC will extend the file by a few more records than is actually needed. This will speed up future attempts to extend the same file by one or two records. File extending can take a considerable amount of time and is directly proportional to the number of records the file is being increased. As an example:

```
10 OPEN NEW "TESTD" AS 1
20 FIELD #1, 100 AS G$
30 GET #1, RECORD 20
```

Line 30 in the above program will cause an error number 24 since the file opened was just created (OPEN NEW) and only contains one record. To make this program run without an error, it is necessary to extend the file before the reference is made.

```
10 OPEN NEW "TESTD" AS 1
20 FIELD #1, 100 AS G$
30 PUT #1, RECORD 20
40 GET #1, RECORD 20
```

This program will run without error since line 30 extends the file to contain at least 20 records. It is not necessary to extend a file to its final value before use, since it can be extended at anytime, but it must be extended to the size required to accommodate all record references in the program being run. All new records created in a file while the extending process is operating will contain null characters (0's). Keep in mind that the file extension process can take a long time to complete.

10.14 Record I/O Example

The following short program demonstrates some of the record I/O statements. It works with an existing employee file which contains information about each employee. Each record of the file contains information about one employee and there are 100 records total. Each

employee has a number which corresponds to the record number in the file. This program is used to print a selected employee's phone number and change it if desired. The employee's name is stored in the first 20 characters of each record and the phone number is in character positions 90 through 104.

```
 10 OPEN "EMPLOYEE" AS 1
 20 FIELD #1, 20 AS N$, 69 AS D$, 15 AS P$
 30 INPUT "ENTER EMPLOYEE NUMBER",E%
 40 IF E%>100 THEN 30 ELSE IF E%<=0 THEN END
 50 GET #1, RECORD E%
 60 PRINT N$,P$
 70 INPUT "CHANGE NUMBER",R$
 80 IF LEFT$(R$,1)<>"Y" THEN 30
 90 INPUT "NEW NUMBER",A$
100 LSET P$=A$
110 PUT #1, RECORD E%
120 PRINT "NUMBER CHANGED"
130 GOTO 30
```

Line 10 opens the employee file on channel 1. Line 20 fields the variables such that N$ points to the employee name and P$ points to the phone number. The variable D$ is used as a dummy variable so P$ is positioned correctly in the buffer (we have skipped data in the buffer). line 30 prompts the user to enter the employee number which should be between 1 and 100. If the number entered is 0 or negative, the program will end. If the number is greater than 100, the user will be prompted for the number again. Line 50 reads in the selected employee record and prints his name and phone number in line 60. If the number does not need changed, the program prompts for another employee number. If it does need changed, line 90 inputs the new number, line 100 puts it in the I/O buffer using LSET, and line 110 writes the updated record back to the disk file. This is a simple program but does demonstrate the versatility of record I/O type operations.

11. THE 6800 USR FUNCTION

The USR function allows the programmer to call a machine language subroutine. A 16 bit value is passed to the routine and a 16 bit value may be returned to the BASIC program. To save space in memory for a user supplied subroutine, set the memory end pointer to save space between the end of BASIC and the actual end of memory (see Section 12).

When BASIC encounters a USR function in an expression it evaluates the argument, converts the result to a 16 bit 2's complement integer, and places the result in memory locations hex 26 and 27. Next, BASIC gets a 16 bit address from memory locations hex 24 and 25. This address tells BASIC where the user supplied subroutine is located. If these bytes are zero, an error will be issued. If they are non-zero, a JSR to the address specified will be executed. This address may either be set manually or by using the DPOKE statement.

In the user's routine, it is only necessary to access the locations hex 26 and 27 to retrieve the argument passed from BASIC. To return a value from the subroutine, simply write a 16 bit value into locations hex 26 and 27. When the subroutine is finished, do an RTS to return. It is very important that the executing subroutine returns with the stack pointer at the same location as when it was entered. The subroutine should also not use more than 256 bytes of stack space during execution.

After the subroutine returns control to BASIC the USR function call will assume the value returned in locations hex 26 and 27. The following example demonstrates the mechanics of using USR without performing any specific task.

```
300 DPOKE HEX("24"), HEX("5000")
310 A1 = 10 * USR(6)
```

The machine code may look like this:

```
ORG     $5000
LDX     $0026         GET PASSED ARGUMENT
***                   ACTUAL CODE HERE
LDX     #VALUE        GET VALUE TO BE RETURNED
STX     $0026         SAVE FOR BASIC TO USE
RTS                   SHOULD END WITH RTS
```

Line 300 in the BASIC code sets up the address of the machine language routine ($5000) and puts it in location $24 (the USR vector). Line 310 will pass the value 6 to the machine language routine. After returning from the routine, the value returned will be multiplied by 10 and put in the variable A1. The assembly language code shown demonstrates a method of passing the parameter to and from the user subroutine.

11.1 Calling multiple 6800 USR routines.

While there is only one USR command setup in BASIC, there is no reason not to have more than one actual routine. The following hints will show you how to call as many assembly language routines as you want from BASIC.

There are two ways to accomplish such a task. The first is to pass a parameter to the USR routine which could be decoded by the actual assembly routine to determine which of any other routines to jump to. The second method, which is much simpler, is to change the USR routine start address as desired. This is done via the DPOKE command.

As an example, let's assume we have two assembly language user routines, one at hex 6E00 and one at hex 6FF0. To call these routines in a row (passing zero parameters to them) we would use the following lines.

```
190 DPOKE HEX("24"), HEX("6E00")
200 A = USR(0)
210 DPOKE HEX("24"), HEX("6FF0")
220 B = USR(0)
```

You might want to consider line 190 as part of the USR call for the routine at $6E00 and line 210 as part of the USR call for $6FF0. In fact it might make sense to put the two steps into a single line using the multiple statements per line character. This is done in the following example.

The main purpose of the following example, however, is to show how you might save yourself some typing and chances for mistakes by assigning addresses to variable names. We will setup the address of the USR routine address vector as variable UL and the various user routines as X and Y. The result is as follows.

```
340  UL = HEX("24")
350  X = HEX("6E00")
360  Y = HEX("6FF0")
365  REM
370  REM   NOW USING MULTIPLE STATEMENTS PER LINE,
380  REM   THE CALLS LOOK LIKE THE FOLLOWING.
385  REM
390  DPOKE UL, X :  A = USR(0)
400  DPOKE UL, Y :  B = USR(0)
```

You can see that if there were to be many calls to the USR routines this method would make things much simpler and cleaner.

11.2 THE USR FUNCTION FOR 6809

The USR function allows the programmer to call a machine language subroutine. A 16 bit value is passed to the routine and a 16 bit value may be returned to the BASIC program. When BASIC encounters a USR function in an expression it evaluates the argument, converts the result to a 16 bit 2's complement integer, and places the result in memory at MEMEND-4. Next, BASIC gets a 16 bit address from memory at MEMEND-2. This address tells BASIC where the user supplied subroutine is located. If these bytes are zero, an error will be issued. If they are non-zero, a JSR to the address specified will be executed. This address may either be set manually or by using the POKE statement. MEMEND represents the FLEX memory end pointer. If memory end is $7FFF, then the USR argument will be at $7FFF-4 or $7FFB, and the USR vector will be at $7FFF-2 or $7FFD. You must know where the FLEX memory end is to use USR! The FLEX memory end is stored at $CC2B and $CC2C. The following explanation assumes a memory end of $7FFF.

In the user's routine, it is only necessary to access the locations hex 7FFB and 7FFC to retrieve the argument passed from BASIC. To return a value from the subroutine, simply write a 16 bit value into locations hex 7FFB and 7FFC. When the subroutine is finished, do an RTS to return. It is very important that the executing subroutine returns with the stack pointers at the same location as when it was entered. The subroutine should also not use more than 256 bytes of stack space during execution.

After the subroutine returns control to BASIC the USR function call will assume the value returned in locations hex 7FFB and 7FFC. The following example demonstrates the mechanics of using USR without performing any specific task.

```
300 DPOKE HEX("7FFD"),HEX("C100")
310 A1 = 10 * USR(6)
```

The machine code may look like this:

```
ORG     $C100
LDX     $7FFB          GET PASSED ARGUMENT
***                    ACTUAL CODE HERE
LDX     #VALUE         GET VALUE TO BE RETURNED
STX     $7FFB          SAVE FOR BASIC TO USE
RTS                    SHOULD END WITH RTS
```

Line 300 in the BASIC code sets up the address of the machine language routine ($C100) and puts it in location $7FFD (the USR vector). Line 310 will pass the value 6 to the machine language routine. After returning from the routine, the value returned will be multiplied by 10 and put in the variable A1. The assembly language code shown demonstrates a method of passing the parameter to and from the user subroutine.

11.3 Calling multiple 6809 USR routines.

While there is only one USR command setup in BASIC, there is no reason not to have more than one actual routine. The following hints will show you how to call as many assembly language routines as you want from BASIC.

There are two ways to accomplish such a task. The first is to pass a parameter to the USR routine which could be decoded by the actual assembly routine to determine which of any other routines to jump to. The second method, which is much simpler, is to change the USR routine start address as desired. This is done via the DPOKE command.

As an example, let's assume we have two assembly language user routines, one at hex AE00 and one at hex AFF0. Also assume memory end is at $7FFF as before. To call these routines in a row (passing zero parameters to them) we would use the following lines.

```
190 DPOKE HEX("7FFD"), HEX("AE00")
200 A = USR(0)
210 DPOKE HEX("7FFD"), HEX("AFF0")
220 B = USR(0)
```

The main purpose of the following example, however, is to show how you might save yourself some typing and chances for mistakes by assigning addresses to variable names. We will setup the address of the USR routine address vector as variable U and the various user routines as X and Y. The result is as follows.

```
340 LET U = HEX("7FFD")
350 LET X = HEX("AE00")
360 LET Y  = HEX("AFF0")
365 REM
370 REM    NOW USING MULTIPLE STATEMENTS PER LINE,
380 REM    THE CALLS LOOK LIKE THE FOLLOWING.
385 REM
390 DPOKE  U, X : A = USR(0)
400 DPOKE  U, Y : B = USR(0)
```

You can see that if there were to be many calls to the USR routines this method would make things much simpler and cleaner.

12. GETTING BASIC RUNNING

Before BASIC can be run, FLEX should be running and the three plus sign prompt (+++) should be present. Insert the disk containing BASIC into the system drive and type XBASIC. After a few seconds, BASIC should respond with "READY". BASIC is ready to go at this time. If BASIC is ever left by using RESET or the EXIT command, the WARM start entry point should be used. Entry at COLD start will clear out any program you may have entered while entry at WARM start will preserve it.

There is an alternative way of calling BASIC from FLEX. It has the following form:

```
+++XBASIC, <file name>
```

where <file name> is the name of a BASIC program. If this name issupplied on the calling line, once FLEX has loaded BASIC, BASIC will load and execute the program named. The file name defaults to the working drive but may be over-ridden in the file spec. The extension defaults to BAC for a BASIC compiled type program. A BAS extension may be specified with the file name which will allow BASIC to load a BASIC source type file and execute it. This mechanism may be used in the STARTUP file in FLEX so BASIC will load and execute a program when your disk is booted!

13. RENUMBER

BASIC does not contain an internal renumbering routine but does contain a disk resident one. To use it you must be in BASIC and have the program you want renumbered in memory. To initiate the renumbering process, type the following:

```
+RENUMBER,<1st Line>, <Increment>
```

where the + sign tells BASIC to send the following command to FLEX which gets the renumber command. The first number represents the number which should be assigned to the first line of the program. The increment indicates what value should be added to each successive line. Both of these values default to 10 if no numbers are specified. For example:

```
+RENUMBER
+RENUMBER,100,20
```

The first example will renumber the program with line number increments of 10 and the first line number will be 10. The second line will renumber it with the first line being 100 and a line increment of 20. Long source files may require a long time to renumber.

14. ADAPTING TO YOUR 6800 SYSTEM

There are several key locations in BASIC which will help you adapt it to your particular hardware configuration. If you are running a 6800 system, has an ACIA for terminal I/O at location hex 8004, and is using FLEX, no adaptations need be performed. It is recommended that this section is read whether or not it is necessary to make adaptations since other useful information is contained. After making any necessary changes, save BASIC back onto disk from location hex 20 through hex 4BFF. The transfer address should be hex 100.

14.1 User Noted Storage

MEMEND — The end of user memory that BASIC uses is defined in FLEX at location $AC2B. It may be desirable to set the end of memory lower than the actual end of memory to save space for a USER supplied subroutine to be called with the USR function. To do so, SAVE any program currently in BASIC, change the MEMEND vector in FLEX and jump back into BASIC at the COLD start ($100) address.

ACIA — $20-21. These bytes contain the base location of the ACIA being used for terminal I/O. If yours is different from $8004, set accordingly. This location is used by the routine which tests for a control C. If your system does not use an ACIA for terminal input, see section 14.2 below.

COLD — $100. This the cold start address used to initialize BASIC when first bringing up BASIC.

WARM — $103. This is the warm start address normally used to enter BASIC after doing an EXIT or FLEX command. It preserves any program currently in BASIC, and consequently does not reset the stack pointer. If the stack pointer has been changed since exiting BASIC, unexpected results can occur.

EXIT — $106. This is used by BASIC when the EXIT command is typed and should be set to jump to the entry point in the monitor ROM being used. It is currently $E0D0 for MIKBUG compatibility.

14.2 User Supplied Break Routine

If your system does not use an ACIA for terminal input, you will need to supply a routine which checks to see if a character has been received from the keyboard. If you do not need the control C break capability, set the ACIA address described above to point to a zero byte in ROM. This will disable this feature. The user supplied routine should check if a key has been typed, and return the zero status bit cleared (NE status) if so. The character should not be input! Make the following patches:

```
at $024C put BD AD 15 01
at $0298 put BD xx xx 01
at $02F9 put BD xx xx 01
```

where 'xx xx' represents the address of your check key typed routine.

14.3 ADAPTING TO YOUR 6809 SYSTEM

There are several key locations in BASIC which will help you adapt it to your particular hardware configuration. If you are running a 6809 system which has an ACIA for terminal 1/O at location hex E004, no adaptations need be performed. It is recommended that this section is read whether or not it is necessary to make adaptations since other useful information is contained. After making any necessary changes, save BASIC back onto disk from location hex 0 through hex 4BFF. The transfer address should be hex 0000.

14.4 User Noted Storage

MEMEND $CC2B-CC2C. These two bytes specify to BASIC what the end of memory should be. These bytes are part of the FLEX operating system and are set by FLEX. These may be changed by the user if so desired (to a lower value only). It may be desirable to set these lower than the actual end of memory to save space for a USER supplied subroutine to be called with the USR function. If the memory end is changed, you must enter BASIC through the cold start entry point at location 0. Entry through warm start after changing memory end will cause program errors.

ACIA $4D-4E. These bytes contain the base location of the ACIA being used for terminal I/O. If yours is different from $E004, set accordingly. This location is used by the routine which tests for a control C. If your system does not use an ACIA for terminal input, see section 14.5 below. After changing these bytes, the cold start entry point should be used (location 0).

COLD $0000. This is the COLD start address used to initialize BASIC when first bringing up BASIC.

WARM $0003. This is the warm start address normally used to enter BASIC after doing an EXIT or FLEX command. It preserves any program currently in BASIC.

EXIT $0006. This is used by BASIC when the EXIT command is typed and should be set to jump to the entry point in the monitor ROM being used. It is currently $F814, the SBUG entry point.

14.5 User Supplied Break Routine

If your system does not use an ACIA for terminal input, you will need to supply a routine which checks to see if a character has been received from the keyboard. If you do not need the control C break capability, set the ACIA address described above to point to a zero byte in ROM. This will disable this feature. The user supplied routine should check if a key has been typed, and return the zero status bit cleared (NE status) if so. The character should not be input! Make the following patches:

```
at $0022 put 8D E8
at $01C5 put BD xx xx 12 12 12
at $0220 put BD xx xx 12 12 12
```

where 'xx xx' represents the address of your check key typed routine.

15. ASCII CHARACTER CHART

Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec
NUL	00	000	+	2B	043	V	56	086
SOH	01	001	,	2C	044	W	57	087
STX	02	002	-	2D	045	X	58	088
ETX	03	003	.	2E	046	Y	59	089
EOT	04	004	/	2F	047	Z	5A	090
END	05	005	0	30	048	[	5B	091
ACK	06	006	1	31	049	\	5C	092
BEL	07	007	2	32	050	]	5D	093
BS	08	008	3	33	051	^	5E	094
HT	09	009	4	34	052	_	5F	095
LF	0A	010	5	35	053	`	60	096
VT	0B	011	6	36	054	a	61	097
FF	0C	012	7	37	055	b	62	098
CR	0D	013	8	38	056	c	63	099
SO	0E	014	9	39	057	d	64	100
SI	0F	015	:	3A	058	e	65	101
DLE	10	016	;	3B	059	f	66	102
DC1	11	017	<	3C	060	g	67	103
DC2	12	018	=	3D	061	h	68	104
DC3	13	019	>	3E	062	I	69	105
DC4	14	020	?	3F	063	j	6A	106
NAK	15	021	@	40	064	k	6B	107
SYN	16	022	A	41	065	1	6C	108
ETB	17	023	B	42	066	m	6D	109
CAN	18	024	C	43	067	n	6E	110
EM	19	025	D	44	068	0	6F	111
SUB	1A	026	E	45	069	p	70	112
ESC	1B	027	F	46	070	q	71	113
FS	1C	028	G	47	071	r	72	114
GS	1D	029	H	48	072	S	73	115
RS	1E	030	1	49	073	t	74	116
US	1F	031	J	4A	074	u	75	117
SP	20	032	K	4B	075	v	76	118
!	21	033	L	4C	076	w	77	119
"	22	034	M	4D	077	x	78	120
#	23	035	N	4E	078	y	79	121
$	24	036	0	4F	079	z	7A	122
%	25	037	P	50	080	{	7B	123
&	26	038	Q	51	081	/	7C	124
'	27	039	R	52	082	}	7D	125
(	28	040	S	53	083	~	7E	126
)	29	041	T	54	084	DEL	7F	127
*	2A	042	U	55	085			

16. INDEX TO STATEMENTS AND COMMANDS

17. ERROR SUMMARY

Any time a program that is being executed encounters an error, of any kind, execution will be halted immediately and an ERROR MESSAGE will be printed (except when an ON ERROR is in effect). The message contains an ERROR NUMBER which can be looked up in the following table and also the line number in which the error occurred. The table provides a brief explanation of what type of error the number represents. An example of an error message that you could receive is:

ERROR 50 AT LINE 100

Looking in the ERROR TABLE, we see that error number 50 represents an "unrecognizable statement". The message tells us that it occurred at line 100 so we could do a "LIST 100" to display this line and we will more than likely find a typing error in it. The error should be corrected then the program can be run again.

All errors are assigned numbers below 100 except the arithmetic errors which range from 101 through 109 and error number 255. Error 255 informs you that an illegal token has been encountered. This error should never be encountered during normal program debugging. Its occurrence indicates the presence of a bad memory location or other serious problems.

The errors are divided into two tables. Table one contains all of the I/O related errors and are numbered 1 through 49. It is this set of errors which may be acted upon by using the ON ERROR statement. It should be noted that all errors below error number 30 are FLEX errors and their numbers are identical to the FLEX error numbers. Errors 50 through 99 are related to syntax or computational type errors.

NUMBER	MEANING
1	ILLEGAL FMS FUNCTION CODE
2	THE REQUESTED FILE IS IN USE
3	THE FILE ALREADY EXISTS
4	THE FILE COULD NOT BE FOUND
7	ALL DISK SPACE HAS BEEN USED
8	END OF FILE ERROR
9	DISK FILE READ ERROR
10	DISK FILE WRITE ERROR
11	THE FILE OR DISK IS WRITE PROTECTED
12	THE FILE IS PROTECTED
15	ILLEGAL DRIVE NUMBER SPECIFIED
16	DRIVES NOT READY
21	ILLEGAL FILE SPECIFICATION
22	FILE CLOSE ERROR
23	SECTOR MAP OVERFLOW
24	NON-EXISTENT RECORD NUMBER SPECIFIED
25	RECORD NUMBER MATCH ERROR - FILE DAMAGED
26	FLEX COMMAND ERROR
30	DATA TYPE MISMATCH
31	OUT OF DATA IN "READ"
32	BAD ARGUMENT IN "ON" STATEMENT
34	PROGRAMMABLE BREAK (CONTROL-C) TRAP
37	FLEX "ESCAPE RETURN" SEQUENCE TRAP
40	BAD FILE NUMBER USED
41	FILE ALREADY OPEN
42	MUST OPEN FILE AS "NEW" OR "OLD"
43	FILE HAS NOT BEEN OPENED
44	FILE STATUS ERROR
45	FIELD SIZE ERROR (>252 OR <0)
46	CAN'T EXTEND A SEQUENTIAL FILE
47	RECORD 0 NOT ALLOWED
48	MUST USE RANDOM TYPE FILE
50	UNRECOGNIZABLE STATEMENT
51	ILLEGAL CHARACTER IN LINE
52	SYNTAX ERROR
53	ILLEGAL LINE TERMINATION
54	LINE NUMBER 0 NOT ALLOWED
55	UNBALANCED PARENTHESES
56	ILLEGAL FUNCTION REFERENCE
57	MISSING QUOTE IN STRING CONSTANT
58	MISSING "THEN" IN AN "IF" STATEMENT

NUMBER	MEANING
60	LINE NOT FOUND
61	RETURN WITHOUT "GOSUB"
62	"FOR-NEXT" NEST ERROR
63	CAN'T CONTINUE
64	SOURCE NOT PRESENT
65	BAD FILE - WON'T LOAD
66	"RESUME" NOT IN ERROR ROUTINE
67	CAN'T CHANGE SCALE FACTOR
70	DATA TYPE MISMATCH IN "PRINT USING"
71	ILLEGAL FORMAT IN "PRINT USING"
72	MIXED MODE IN AN EXPRESSION
73	ILLEGAL EXPRESSION
74	ARGUMENT <0 OR >255
75	ARGUMENT >32,767
76	ILLEGAL VARIABLE TYPE
77	ARRAY REFERENCE OUT OF RANGE
78	UNDIMENSIONED ARRAY REFERENCE
79	BAD ARGUMENT IN "SWAP" STATEMENT
80	MEMORY OVERFLOW
81	ARRAY OVERFLOW
83	STRING TOO LONG
90	UNDEFINED USER FUNCTION
91	UNDEFINED USER CALL
94	BAD STRING LENGTH SPECIFIED
100	EXPRESSION TOO COMPLEX
101	OVERFLOW OR UNDERFLOW IN FLOATING POINT OP.
102	ARGUMENT TOO LARGE
103	DIVISION BY ZERO
104	NUMBER TOO LARGE TO CONVERT TO INTEGER
105	NEGATIVE OR ZERO ARGUMENT FOR "LOG"
106	CONVERSION ERROR IN INTEGER "INPUT"
107	IMAGINARY SQUARE ROOT
108	CONVERSION ERROR (NUMBER TOO LARGE)
109	OVERFLOW/UNDERFLOW IN INTEGER OPERATION
255	ILLEGAL TOKEN ENCOUNTERED